



**AFRL-RI-RS-TR-2017-021**

**HARDWARE SUPPORT FOR MALWARE DEFENSE AND END-TO-END TRUST**

---

**INTERNATIONAL BUSINESS MACHINES CORPORATION (IBM)**

*FEBRUARY 2017*

**FINAL TECHNICAL REPORT**

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED*

**STINFO COPY**

**AIR FORCE RESEARCH LABORATORY  
INFORMATION DIRECTORATE**

## NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (<http://www.dtic.mil>).

AFRL-RI-RS-TR-2017-021 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /

CARL THOMAS  
Work Unit Manager

/ S /

JOHN D. MATYJAS, Technical Advisor  
Computing and Communications Division  
Information Directorate

This material is based on research sponsored by the Department of Homeland Security (OHS) Science and Technology Directorate, Cyber Security Division (OHS S & T CSD) via BAA 11-02; the Department of National Defence of Canada, Defence Research and Development Canada (DRDC); and Air Force Research Laboratory Information Directorate via contract FA8750-12-C-0243. The U.S. Government and the Department of National Defence of Canada, Defence Research and Development Canada (DRDC) are authorized to reproduce and distribute this report for Government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security; Air Force Research Laboratory; the U.S. Government; or the Department of National Defence of Canada, Defense Research and Development Canada (DRDC).

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.  
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE (DD-MM-YYYY) <b>FEBRUARY 2017</b>			2. REPORT TYPE <b>FINAL TECHNICAL REPORT</b>		3. DATES COVERED (From - To) <b>SEP 2012 - JUN 2016</b>	
4. TITLE AND SUBTITLE  <b>HARDWARE SUPPORT FOR MALWARE DEFENSE AND END-TO-END TRUST</b>					5a. CONTRACT NUMBER	
					5b. GRANT NUMBER <b>FA8750-12-2-0243</b>	
					5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S)  <b>Richard Boivie, Ek Ekanadham, Bhushan Jain, Eric Hall, Guerney Hunt, Mohit Kapur, Mehmet Kayaalp, Elaine Palmer, Dimitrios Pendarakis, David Safford, and Ray Valdez</b>					5d. PROJECT NUMBER <b>DHS2</b>	
					5e. TASK NUMBER <b>IB</b>	
					5f. WORK UNIT NUMBER <b>M2</b>	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) <b>International Business Machines Corporation T.J. Watson Research Center 1101 Kitchawan Rd Yorktown Heights, NY 10598-0218</b>					8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)  <b>Air Force Research Laboratory/RITA                      Department of Homeland Security 525 Brooks Road    1120 Vermont Ave NW Rome NY 13441-4505    Washington DC 20005</b>					10. SPONSOR/MONITOR'S ACRONYM(S) <b>AFRL/RI</b>	
					11. SPONSOR/MONITOR'S REPORT NUMBER <b>AFRL-RI-RS-TR-2017-021</b>	
12. DISTRIBUTION AVAILABILITY STATEMENT <b>Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.</b>						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT <b>This report describes an end-to-end architecture for establishing end-to-end trust. Including computing platforms, Internet of Things (IoT) sensors and actuators, mobile devices and servers; cloud based, stand alone, and traditional mainframes. The prototype developed demonstrated that hardware extensions, along with corresponding firmware can provide strong isolation for secure virtual machines and be transparent to unmodified virtual machines. For mobile platforms we developed and prototyped an architecture supporting separation of personalities on the same platform, safeguarding enterprise from personal data in a bi-directional manner. Lastly we demonstrated IoT sensor and actuator security using trusted security.</b>						
15. SUBJECT TERMS <b>End-to-End Trust, Secure, Hyper-Visor, Super-Visor, IoT devices, Servers, Virtual Machines</b>						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			<b>CARL THOMAS</b>	
<b>U</b>	<b>U</b>	<b>U</b>	<b>UU</b>	<b>81</b>	19b. TELEPHONE NUMBER (Include area code) <b>N/A</b>	

## Table of Contents

<b>TABLE OF FIGURES.....</b>	<b>iii</b>
<b>ACKNOWLEDGMENTS.....</b>	<b>iv</b>
<b>1. SUMMARY.....</b>	<b>1</b>
<b>1.1. SUMMARY OF PROTOTYPES DEVELOPED.....</b>	<b>1</b>
<b>1.2. SUMMARY OF TECHNOLOGY TRANSITION ACTIVITIES.....</b>	<b>2</b>
<b>2. INTRODUCTION.....</b>	<b>3</b>
<b>3. METHODS ASSUMPTIONS AND PROCEDURES.....</b>	<b>4</b>
<b>3.1. MOBILE.....</b>	<b>4</b>
<b>3.1.1. GOALS.....</b>	<b>4</b>
<b>3.1.2. HARDWARE.....</b>	<b>4</b>
<b>3.1.3. SOFTWARE ARCHITECTURE.....</b>	<b>5</b>
<b>3.1.4. INTEGRITY COMPONENT DESCRIPTIONS.....</b>	<b>5</b>
<b>3.1.5. APPROACH TO IMPLEMENTATION.....</b>	<b>7</b>
<b>3.2. SERVER.....</b>	<b>8</b>
<b>3.2.1. BASIC PRINCIPLES.....</b>	<b>8</b>
<b>3.2.2. BASIC COMPONENTS.....</b>	<b>8</b>
<b>3.2.3. BUILDING THE ACM BLUESPEC FPGA MODEL OF A POWER PC SERVER PROCESSOR.....</b>	<b>9</b>
<b>3.2.4. MODIFYING AN EXISTING POWER PROCESSOR FOR ACM.....</b>	<b>11</b>
<b>4. RESULTS AND DISCUSSION.....</b>	<b>18</b>

<b>4.1. MOBILE PLATFORM RESULTS .....</b>	<b>18</b>
<b>4.1.1. IMA-APPRAISAL-IMASIG TEMPLATE PATCHES AND POLICIES.....</b>	<b>18</b>
<b>4.1.2. OAT ANALYSIS AND INTEGRATION.....</b>	<b>18</b>
<b>4.1.3. QEMU INTEGRATION WITH CUSE BASED SWTPM .....</b>	<b>18</b>
<b>4.1.4. CHANGES TO THE ORIGINAL PLAN .....</b>	<b>19</b>
<b>4.1.5. STATUS .....</b>	<b>20</b>
<b>4.2. SERVER RESULTS.....</b>	<b>20</b>
<b>4.2.1. ACM BLUESPEC FPGA MODEL.....</b>	<b>20</b>
<b>4.2.2. MODIFYING THE MODEL OF AN EXISTING POWER PROCESSOR .....</b>	<b>33</b>
<b>4.2.3. PERFORMANCE OF BARE BONES ACM FIRMWARE .....</b>	<b>38</b>
<b>4.3. CUSTOMER NEEDS ADDRESSED.....</b>	<b>39</b>
<b>4.4. COMPARISON WITH COMPETITION.....</b>	<b>39</b>
<b>4.5. TECHNOLOGY TRANSITION AND TRANSFER.....</b>	<b>39</b>
<b>5. CONCLUSION .....</b>	<b>41</b>
<b>6. REFERENCES .....</b>	<b>42</b>
<b>APPENDIX A: PUBLICATIONS .....</b>	<b>43</b>
<b>LIST OF SYMBOLS ABBREVIATIONS AND ACRONYMS .....</b>	<b>73</b>

## TABLE OF FIGURES

Figure 1 Mobile Architecture.....	6
Figure 2 Samsung verified boot.....	8
Figure 3 Architectural block diagram of the ACM Bluespec FPGA model.....	10
Figure 4 Design flow of ACM Bluespec FPGA model.....	11
Figure 5 ACM concept: confidentiality and integrity by isolation.....	12
Figure 6 ACM demonstration hardware preparation.....	12
Figure 7 ACM demonstration: program loaded into memory.....	13
Figure 8 FPGA based emulation platform.....	14
Figure 9 Overview of the Linux/KVM environment on Powre systems.....	16
Figure 10 QEMU Process and guest system relationship.....	16
Figure 11 Conceptual view of SVM.....	17
Figure 12 ACM demonstration: step 1 - creation of initial ACM.....	21
Figure 13 ACM demonstration: step 2 - domain D2 requested.....	22
Figure 14 ACM demonstration: step 3 domain D3 completed.....	23
Figure 15 ACM demonstration: Step 4 - OS attempts invalid access.....	24
Figure 16 Process isolation OS jumps to secure code.....	24
Figure 17 Process isolation secure domain invalid access.....	25
Figure 18 Process hierarchy illustration.....	28
Figure 19 Functional overview of ACM firmware.....	36
Figure 20 Components required for bare bones ACM firmware.....	37

## ACKNOWLEDGMENTS

The following staff members at IBM Research have contributed during the duration of this project:

Rick Boivie, Ek Ekanadham, Kenneth Goldman, Bhushan Jain, Eric Hall, Guerney Hunt, Mohit Kapur, Mehmet Kayaalp, Elaine Palmer, Dimitrios Pendarakis, David Safford (now with GE Research), Ed Suh (Cornell University, while an academic visitor with IBM Research) and Ray Valdez.

Additionally, we would like to acknowledge a number of IBM employees, across both Research and business units, in particular IBM Systems, for the extensive and close collaboration we had while pursuing commercialization of the technologies developed throughout this project. Specifically, we would like to thank Jens Leenstra, Pete Sandon (now retired), Utz Bacher, Stefan Berger, Jonathan Bradbury, Reinhard Buendgen, John Cohn, John Dayka, Donna Dillenberger, Brad Frey, Joefon Jann, Christian Jacobi, Ronald Kalla, Jeb Linton, Angel Nunez Mencias, Jose Moreira, Pratap Pattnaik, JR Rao, Balaram Sinharoy, Bill Starke, Charles Webb, George Wilson Christian Zoellin and Mimi Zohar.

Finally, we would like to acknowledge the valuable feedback and guidance we received throughout the duration of this project from our government sponsors from the Department of Homeland Security (DHS) Science and Technology Directorate (S&T) and the Air Force Research Laboratory (AFRL), Rome, New York. Specifically, we would like to thank Edward Rhyne, Matthew Billone, John Drake and Douglas Maughan from DHS S&T and Carl Thomas from AFRL.

## 1. SUMMARY

The objective of this project was to develop and prototype an end-to-end architecture for defending against “malware” and establishing end-to-end trust. The performers defined “end-to-end” as spanning computing platforms from Internet of Things (IoT) sensors and actuators or mobile devices to servers which could be cloud based, stand alone or traditional mainframes. In this environment it was observed “trust” cannot be established if the software components of any part of the Trusted Computing Base (TCB) were unknown or unverified. This definition of trust is similar to the one used by the Trusted Computing Group (TCG) for the definition of the Trusted Platform Module (TPM). It is noted that for long running systems, establishing trust at boot may be insufficient for continuation of trust at an arbitrary point in the future. If the design of the system permits undetectable attacks against memory, continuous trust must be established. Our conclusion is that for the smallest devices where the dynamic memory is typically inaccessible to attackers and the software operating the device may be replaceable or upgradeable, secure boot is necessary and trusted boot is sufficient. As devices become more capable these technologies combined with the work described herein become necessary.

### 1.1. SUMMARY OF PROTOTYPES DEVELOPED

This project developed a number of different prototypes, targeted for the different computing platforms, as we outline below.

- Mobile/Embedded prototypes with security enhancements
  - Secure firmware enhancements for representative embedded Linux devices
  - Mobile platform with integrity management and verification for different “personalities”.
  - Trust Dust: Secure Cyber Physical/IoT platform with embedded hardware root of Trust — demonstrated at the 2013 PI Meeting
- Access Control Monitor ACM a server side architecture
  - We build two simulation environments for enhanced general-purpose processor architectures, targeted for secure server platforms. Please note that a summary of our proposed hardware-enabled security architecture for server platforms and its evaluation is provided in the draft paper included in 6.
    - A PowerPC Book III compliant processor in the Bluespec language, emulated on a field programmable gate array (FPGA) platform
    - An IBM Power Server central processing unit (CPU) software simulation platform, based on a commercial IBM Power Systems processor
  - ACM Firmware (previously referred to as Ultravisor): we have developed a prototype firmware component which manages the ACM hardware extensions and which is in the process of being transferred to our IBM Systems Group partners.
    - We tested and validated this prototype by booting the Linux/KVM hypervisor on the ACM hardware model and then booting secure virtual machines on top of Linux/KVM.
    - We demonstrated that the ACM firmware function is transparent to any Virtual Machine that is not exploiting the ACM/SB++ functionality.
    - We started our prototype from a “barebones ACM firmware” and have been improving it and adding functionality as the project evolved.
  - Tools for building and deploying secure virtual machines (SVMs)



- Created Linux command line utilities for building SVMs from existing VMs in Power Linux
- Designed and developed a secure bootstrap loader for launching SVMs in PowerKVM
- Made appropriate modifications to the Open Firmware boot process and Petitboot ([git://ozlabs.org/jk/petitboot](https://ozlabs.org/jk/petitboot)) to create the initial bootstrap loader for booting SVMs

## 1.2. SUMMARY OF TECHNOLOGY TRANSITION ACTIVITIES

- Commercialization: The processor hardware and firmware security architecture developed is pursued for commercialization with the IBM Systems Group and specifically IBM servers. In particular, a derivative of the ACM hardware extensions and associated ACM firmware are planned for release in a next generation Power processor. The planned feature was referred to as “trusted execution enforced by hardware” in a recent presentation by IBM, titled “POWER9: Processor for the Cognitive Era”, at the Hot Chips Conference [7]. The relevant slides from this presentation are extracted and included in Appendix A. The embedded/mobile and cyber-physical security assets (including “Trust Dust”) are pursued within IBM, in particular the recently formed IoT division, and with selected partners.
- Standards: “Trust Dust”, the early prototype of the value of embedded TPM for cyber-physical systems influenced the creation of the Trusted Computing Group (TCG) IoT Subgroup (referred to as a root of trust for measurement (RTM)).
- Open Source Software Contributions: As mentioned in 3.2.1 our approach leverages and builds on the Trusted Computing model. In particular, our work on the secure mobile prototype motivated additional requirements for the virtual Trusted Platform Module (vTPM) and led to a new vTPM implementation (tpm server cuse) that was open sourced, along with the requisite extensions to quick emulator (QEMU) and libvirt.

## **2. INTRODUCTION**

This report is organized as follows. In Section 3 we describe our approach to this project, including the key assumptions, methods and procedures we employed in the course of our research. We break these down across the different platforms where we pursued hardware-enabled security; mobile, FPGA-emulator and server class processor. In Section 4 we outline the key results, again along the different platforms, and discuss their benefits, comparison with competitive offerings and status of commercialization. Next, we present the conclusions of our project in Section 5 and include references in Section 6. Additionally, the report includes an appendix which lists academic publications generated by this project. Finally, at the end there is a list of all symbols, abbreviations and acronyms used in this report.

### 3. METHODS ASSUMPTIONS AND PROCEDURES

As described earlier we developed an end-to-end architecture, consequently this project worked on the one end on client devices, such as IoT and mobile platforms, and on the other end on higher performance servers. First, we will describe how we approached the IoT sensor, actuator and mobile device platforms, followed by a description of our work our methodology and assumption on the server end.

#### 3.1. MOBILE

In this section, we describe our approach to developing a secure mobile platform. Our objective is to develop an architecture that supports separation between multiple personalities on the same platform. As an example, a personal mobile phone or tablet is used within an enterprise to access corporate data and applications, in addition to personal applications. The different personalities separate and safeguard enterprise from personal data and applications in a bi-directional manner.

While some prior work has focused on how to provide “sandboxing” and isolation between different applications and/or containers running on the same mobile platform, less emphasis has been placed on verifying the integrity of these applications and/or containers. In this work, the initial focus is on mobile containers of Virtual Machine granularity, trying to leverage the capabilities available through virtualization technologies like those present in servers and desktops. This was possible for this project as new mobile processors with hardware virtualization support became available. The technologies that were developed verified the integrity and trustworthiness of different virtual machines, which correspond to different personalities. This project also leveraged secure and trusted boot capabilities that were developed for embedded Linux devices.

**3.1.1. GOALS** The mobile prototype is a key component of our end-to-end security architecture, offering a practical demonstration realized in an actual mobile device. It demonstrates the combination of isolation, integrity measurement (trusted boot), integrity appraisal (secure boot), and remote attestation, and it was implemented on hardware representative of the mobile space (as described below), including phones, tablets, and books (ebooks, netbooks and Chromebooks). The specific goals were:

- Mobile Compatible Hardware
- Multiple VMs (Personalities) to isolate domains
- Measured/Appraised/Attested Native and VMs
- Trusted and Secure boot with hardware root of trust
- Field Deployable

**3.1.2. HARDWARE** For the prototype we selected the Samsung ARM Chromebook. This is a representative platform for the mobile environment, since it has a hardware architecture, which is compatible across the phone/tablet/book spectrum, while supporting the project requirements for isolation and integrity management. In particular, it has:

- The same base hardware as the Galaxy S4 phone
- Acorn RISC machine (ARM) A15 cores (with hardware virtualization support)
- 1GB random access memory (RAM), 16GB Flash
- Trusted Boot with Hardware TPM
- Secure Boot with SPI Hardware Protected Mode (HPM)
- Perfect prototype for Phone/Tablet/Book use cases

The ARM A15 cores are the first mobile cores capable of supporting virtualization, and kernel virtual machine (KVM) in particular is supported as of Linux Kernel 3.11. As this is the same SoC (Samsung Exynos 5250), with the same flash and RAM sizes as the international version of the Galaxy S4 phone, this hardware is representative of a significant portion of the mobile market <sup>1</sup>.

The Chromebook also contains a TPM chip, which provides the needed hardware root of trust for measurement and attestation. In addition, the Chromebook has a hardware root of trust for secure boot, based on hardware write-protection of the bootstrap SPI flash, and the u-boot boot code implements secure boot (Samsung calls it “verified boot”) based on this hardware write-protection. In fact, this verified boot makes it difficult to install our own new kernel that supports KVM, as we do not have Google's private key for signing the new kernel. The solution for this problem is described in section 3.1.5.

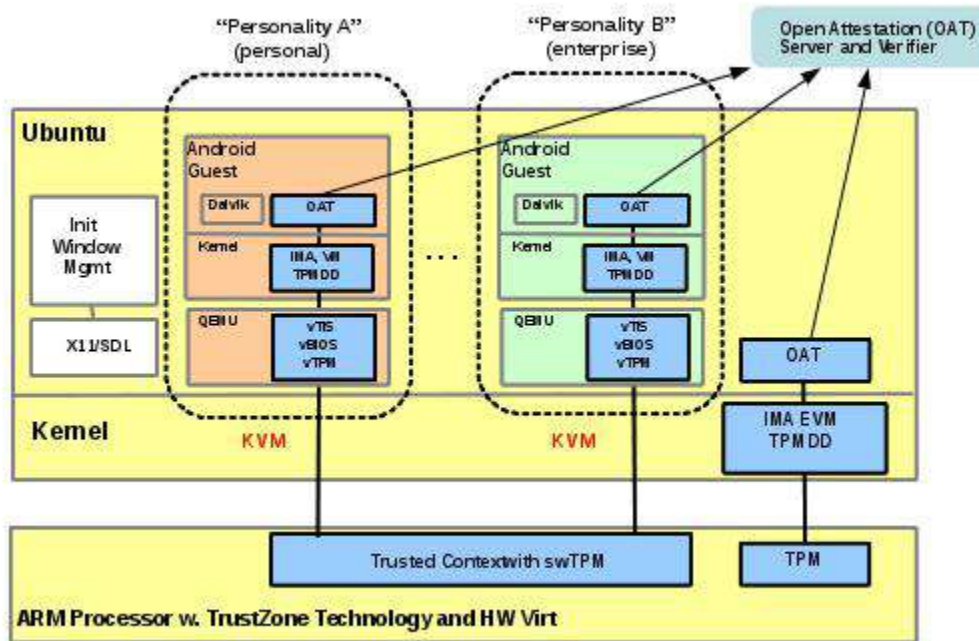
**3.1.3. SOFTWARE ARCHITECTURE** Starting from the Chromebook hardware platform, this project removed the base Chromium OS software that comes pre-installed, and installed our own version of Linux (based on Ubuntu), which supports running multiple isolated personalities (KVM Virtual Machines) of Android, while including our overall integrity architecture with integrity measurement, appraisal, and attestation, both for the native Linux, and for the Android guests. The overall architecture developed in this project is shown in Figure 1.

**3.1.4. INTEGRITY COMPONENT DESCRIPTIONS** Figure 1 shows the overall software architecture, with the base Linux kernel and window management, and Android guests running on KVM. In addition, it shows the integrity components added. These components have been developed separately by IBM and other open source community contributors, initially targeted for desktop and server environments. These integrity components are:

- TPM - this is the hardware TPM, which is used as the root of trust for measurement and attestation for the native Linux system (native kernel, and stripped down Ubuntu based user space.) This is a standard part of the Chromebook, although the standard Chromium OS does not take advantage of it as a measurement root of trust.
- swTPM - this is the software based TPM emulator [1]. This is used to provide emulated TPM service to the KVM guests. This code is added to the native Linux system, and is measured/appraised/attested as part of the native system. It provides TPM services to the guests, with guarantees similar to a physical TPM, as this software is outside of the running guests, so they cannot tamper with it, just as a native OS cannot tamper with the physical TPM.
- IMA - (Integrity Measurement Architecture [2]) has several components, which provide various integrity services. The base IMA maintains a kernel-based measurement list of all files accessed, including all measurements of the boot chain. This list is anchored in PCR 10 of the TPM, and the TPM can sign PCR 10 with a private key known only to the TPM, so this signature (called a TPM Quote) cannot be forged by the possibly corrupt OS. This attestation is created and verified by the OpenAttestation (OAT) [3] client and server components.

In addition, IMA has an appraisal module, which validates RSA signatures on all files

<sup>1</sup> Even though most mobile phones do not contain a TPM, there is an active standardization effort, for example in the Trusted Computing Group (TCG), focused on defining a “TPM” for phones and smaller devices.



**Figure 1 Mobile Architecture**

accessed. If a file's signature is invalid, access to that file is denied, even to root, thus protecting the integrity of the running system. If a file has a valid signature, IMA appraisal adds the measurement and signature information to the measurement list. The signature data makes analysis of the measurement list much simpler and more scalable, as it provides file provenance for all measurements (the signature includes the key fingerprint, which identifies the signer of the file.) With the signature extensions, attestation verification reduces to verifying the small number of signing keys used, rather than maintaining a large list of all “good” file hashes.

IMA and IMA-Appraisal are upstreamed in the current Linux Kernel.

- EVM - The extended verification module [2] module signs or HMAC's file metadata (the inode and all of its security extended attributes, including the selinux label and IMA signature) to prevent off-line attacks on the file's data or metadata.
- TPMDD - this is the Linux kernel's TPM device driver for the native (physical) TPM. This is already a standard kernel component.
- OAT - OpenAttestation [3] includes a monitoring agent on the mobile device and on its guest Android systems. The client agent provides the measurement list and the corresponding TPM Quote, for remote attestation. The OAT server/verifier collects the integrity reports, verifies them, and displays the results for all monitored systems on the OAT portal.

The OAT client and server are extended to verify the IMA measurement list, and to provide details and summary to the OAT server display.

The remaining three components are patches to QEMU to support emulation of a TPM to KVM guests [4]:

- eTPM - This is a the swTPM emulation, modified to fit into QEMU as a library, so that QEMU can provide an emulated TPM service to its guests.
- eBIOS - This is an extension to the normal SEA BIOS used by QEMU as the virtual machine's BIOS, to perform the necessary standard boot-time measurements for the guest.
- eTIS - This is the emulation of the hardware TPM interface added to QEMU to support the guest. (TIS is the Trusted Interface Specification, which gives the details of the TPM chip hardware interface on a nominal low pin count (LPC) bus.

An earlier implementation of this architecture on a server platform, applicable to trusted cloud environments was presented and demonstrated in [6]. The work on these components in the open source community has continued since then.

**3.1.5. APPROACH TO IMPLEMENTATION** The basic ARM/KVM installation on the Chromebook, as enabled in kernel 3.11, is described in [5]. Though insightful, this paper has serious limitations. It describes how to install KVM only on an external SD card and turns off the Chromebook's verified boot. It does not automate startup of guests or hide the native Ubuntu OS at all. The instructions are incomplete or incorrect in several places. In addition to installing our own Linux kernel natively (not only on an external SD card), we also have to add our desired integrity components: IMA for the native and guest kernels, IMA-Appraisal for the native and guest images, OAT clients for the native and guest systems, signing of all files, vTIS, vBIOS, and vTPM support in the native QEMU.

**Installation** The Samsung Arm Chromebook has restricted boot (it is called verified boot or VB.) The architecture of verified boot is illustrated in Figure 2.

In this architecture, the Google root key is locked in the SPI flash with HPM that prevents any modification unless the device is disassembled, and a washer removed. Even then, there are no instructions or scripts that support changing the root key.

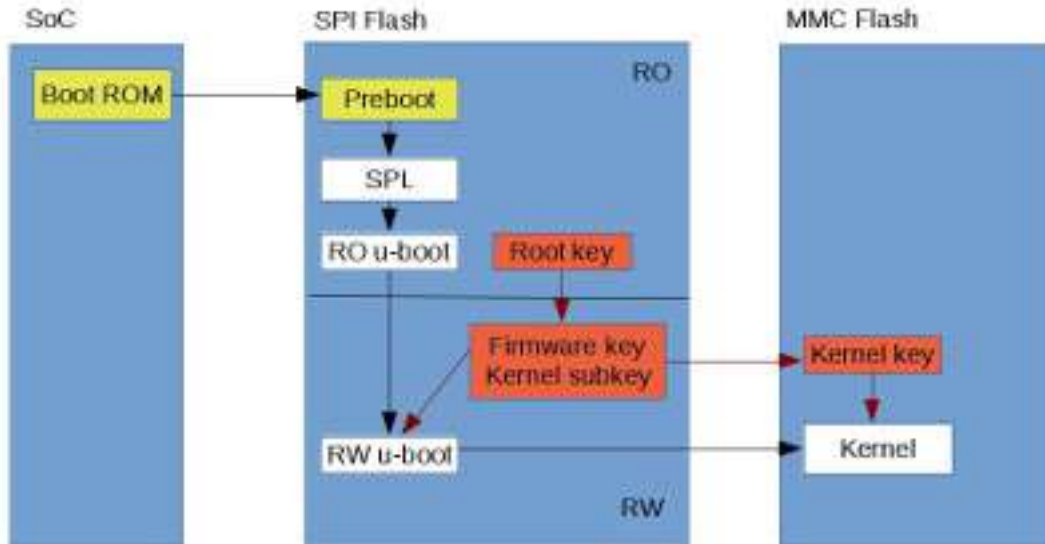
The Chromebook's U-boot is locked in a 4MB serial programming interface (SPI) flash, so all of the tools used in our prior embedded Linux security project are available to help read and write the u-boot, once the chip is physically unlocked (and they can help recover the Chromebook, even if it is accidentally bricked).

In order to achieve our objectives, the u-boot flash had to be unlocked so that the root key could be replaced with our key. This enabled our project to sign the new KVM capable kernel, while retaining secure and trusted boot supported by the hardware. Three scripts were developed to automate this key replacement and kernel signing: make new keys, sign firmware and sign kernel.

**Taking Control Instructions** This is the process that was used to take control of the Chromebook:

- Disassemble Chromebook and remove !WP washer
- Enter Developer mode (esc-refresh-power)
- Copy scripts to somewhere executable (/usr/local/takeown)
- Press (ctl-alt->), login as chronos, sudo -i
- ./makekey (makes all new key pairs)

- ./takeown firmware.sh (signs RW u-boots and keys)
- ./takeown kernel.sh (signs kernels and keys)
- dev debug vboot (verifies all keys/signatures)
- Modify RO u-boot to set power cycle protection
- Save keys to usb, reboot, and follow prompts for normal mode



**Figure 2 Samsung verified boot**

### 3.2. SERVER

This project worked on two variants of the server architecture that are related to one another. The architecture was first developed as a formal model, which was proven. That model fulfilled the basic principles listed in section 3.2.1. We worked on realizing this architecture with two different but complementary approaches. The first was to modify an existing Bluespec model for a server class PowerPC processor. The second was to modify an existing mambo model for an IBM Power 8 processor. These two approaches are reported on separately below.

**3.2.1. BASIC PRINCIPLES** The following are the basic principles of our approach.

- Protect integrity and confidentiality of both code and data
- Minimize the Trusted Computing Base (TCB) hardware and software that needs to be trusted. In our case the TCB consists of the processor and a small amount of firmware that manages hardware extensions
- Minimize changes on existing hardware and systems software to maximize commercialization opportunity
- Ability to apply transparently to existing software
- Leverage as much as possible the existing TCG architecture for hardware root of trust
  - we aim to build on the TCG model; not to replace it

**3.2.2. BASIC COMPONENTS** To realize our objectives and basic principles, we derived our approach from the following basic components.

- SecureBlue++: technology to create:
  - Cryptographically protected enclaves corresponding to a process

- Build and distribute protected software, which may be optionally encrypted using a system key to protect distribution with secrets
- Leverage complementary trusted computing capabilities to verify capabilities of target systems and manage system keys
- Initial focus on Linux/KVM hypervisor and Linux OS
- A Bluespec model of a server class PowerPC processor
- A mambo model of an existing IBM server

**3.2.3. BUILDING THE ACM BLUESPEC FPGA MODEL OF A POWER PC SERVER PROCESSOR** This approach was done by modifying an existing Bluespec PowerPC server class processor model. A demonstration was run on a Verilog simulator and was shown at the DHS S&T PI meeting in December 2014. The architecture of the ACM Bluespec model is shown in Figure 3.

This figure depicts the delineation between hardware and software. The hardware model consists of a processor, connected to main memory over a memory bus and external devices over an I/O bus. The ACM hardware component is shown as connected to the processor. The software components consist of the “ACM Software”, an operating system and two secure processes. Each of the software components is labeled with a different color, which is the same as the color of the memory that they are accessing.

Figure 4 shows the development flow that was employed in the construction of the ACM model and the demonstration components. The processor architecture changes are coded in the Bluespec language and translated into Verilog. The partition and synthesis tools, with input from the infrastructure logic, generate FPGA images that are loaded onto our FPGA hardware platform, shown in the icon. The software components that are part of the demonstration scenario (ACM Software, OS and secure processes) are loaded onto the platform and a set of simulated events are triggered to test how the ACM protects against unauthorized accesses. A trace is generated and by inspecting it, we can confirm that the correct ACM operation took place. The detailed steps are shown in the following sequence of figures, a subset of which were shown at the public demonstration event organized by DHS S&T CSD R&D Showcase in December 2014.

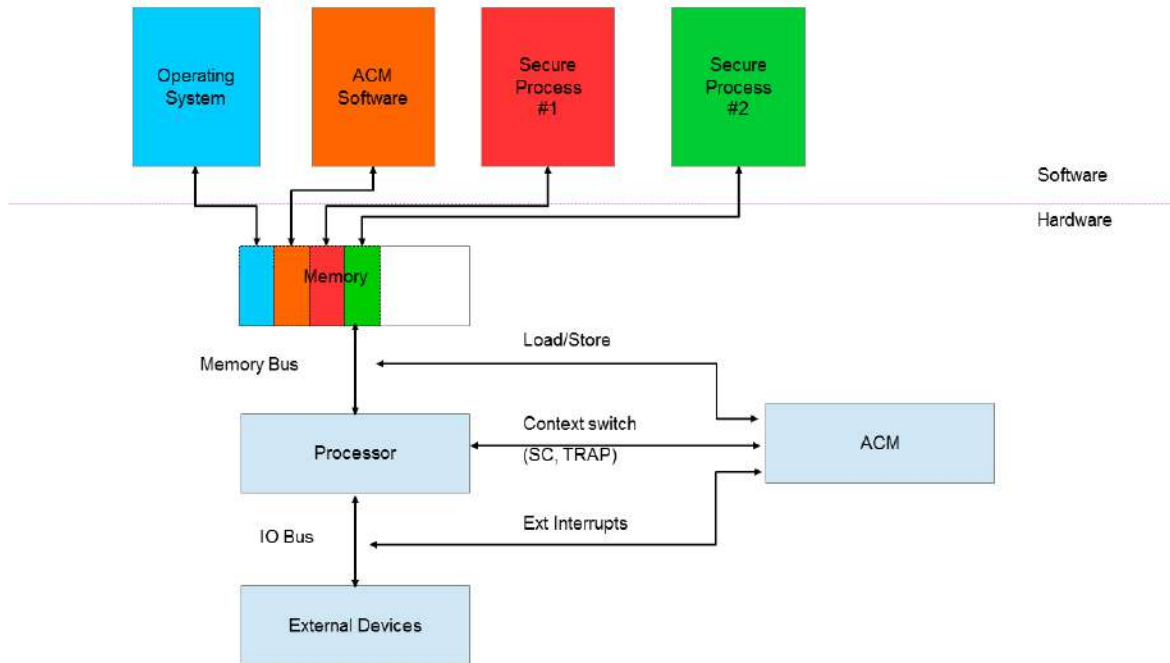
Figure 5 provides a review of the key ACM concept and objectives.

Figure 6 shows how the ACM is integrated with the PowerPC (PPC) processor model through the "glue logic" and the steps taken to prepare the hardware for the demonstration.

Figure 7 presents a synopsis of how the program is loaded onto memory.

Figure 8 shows our FPGA based emulation platform. It comprises of 28 daughter cards populated with Xilinx Virtex-5 and Virtex-6 devices. Each daughter card has on board 32 MB of static RAM





**Figure 3 Architectural block diagram of the ACM Bluespec FPGA model**

(SRAM) or 2GB of dynamic RAM (DRAM). These 28 daughter cards are mounted on a motherboard. The motherboard has a board controller FPGA and four more FPGAs which are primarily used for routing and logic between the 28 cards. All daughter cards as well as the motherboard are connected to a host machine via 1Gb Ethernet cables running UDP. The host machine runs a Server on top of Linux. This Server provides the tool control language (TCL) user interface to the FPGA hardware. It is used to configure the FPGAs, load files directly into DRAM or SRAM and issue commands to the board controller for single stepping or waveform extraction.

To produce configuration files for the FPGA devices we followed the following procedure:

1. Bluespec models of PPC + ACM were compiled to generate Verilog models.
2. VHDL models of clock controller, double data rate (DDR) DRAM controller were then added to the above Verilog models.
3. The combined model was then synthesized using Xilinx tool chain to generate bit files for configuration.
4. To load programs into the PPC memory, a haskell based custom compiler was built to generate the object code. This object code was then directly loaded into the DRAM.

*We observed a six percent increase in FPGA LUT utilization due to the addition of ACM to the PPC core.*

### Software setup process isolation demonstration

The software is primarily divided into three code segments:

1. Operating system (OS) code. The OS segment is further divided into:
  - a) Code to invoke ACM functions. We choose the location of this code at 0x0 address.
  - b) Code to handle ACM exceptions. We chose the location of this code at 0xf00 address.
  - c) Code to handle System calls. We chose the location of this code at 0xb00 address.
  - d) A simple scheduler was coded to schedule various tasks.

2. Code performing some of the ACM functions (ACM firmware).
3. Secure domain code.

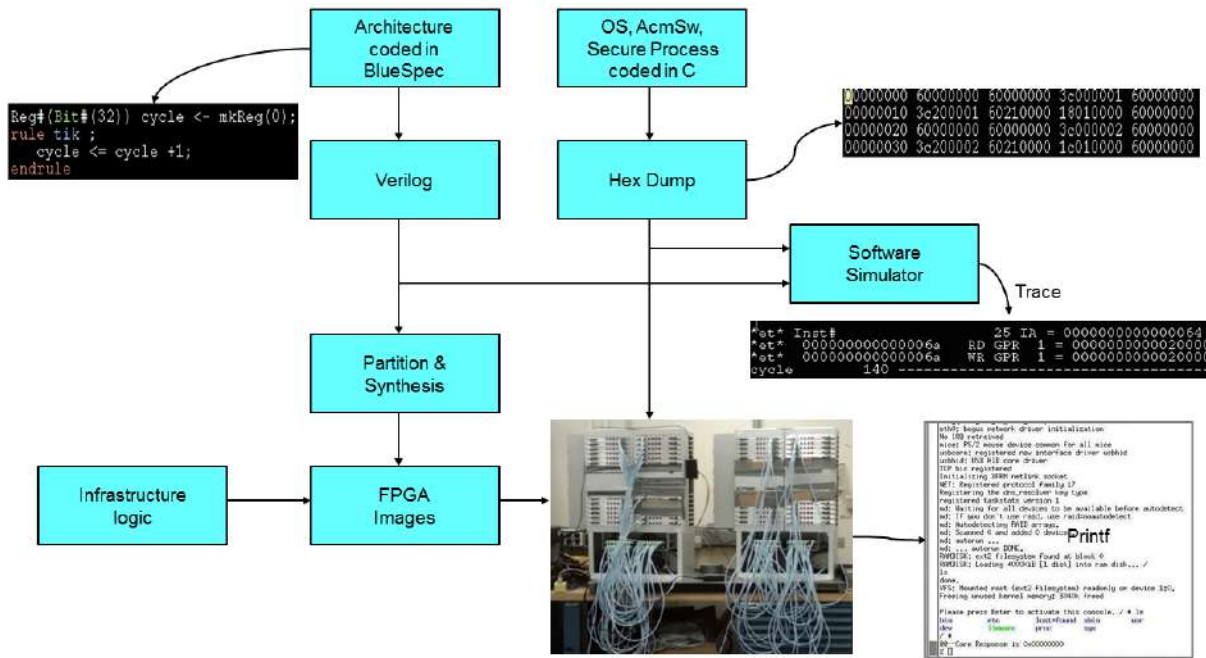
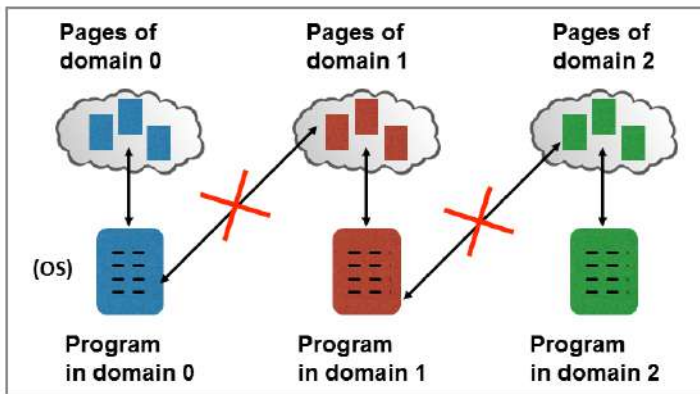


Figure 4 Design flow of ACM Bluespec FPGA model

**3.2.4. MODIFYING AN EXISTING POWER PROCESSOR FOR ACM** Our second approach to ACM involves modifying a model of an existing IBM server. These modifications introduce a new higher privileged mode into the architecture called ultravisor mode. In order to do this we had to add new registers, interrupts, and instructions. A more detailed but still high level introduction to these modifications can be found in the draft ASPLOS paper in Appendix A. This paper describes these new features in the context of the existing Power ISA™ Version 2.07 B architecture (P8). ACM also adds some new control bits in other parts of the architecture.

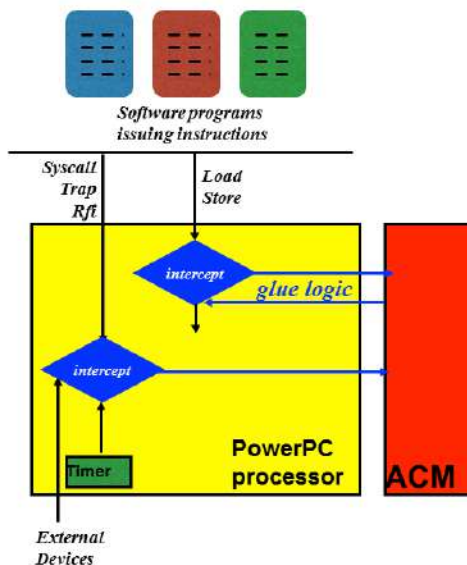
The Access Control Monitor (ACM) facility provides secure isolation of virtual machines and applications from one another and from system software. ACM functionality is implemented using a combination of hardware facilities and firmware that runs at a privilege level above the hypervisor. ACM targets a threat model in which the hypervisor or operating system can be compromised such that its inherent isolation capabilities can no longer be counted on.

The ACM protection mechanism is based on an assignment of virtual machines (VMs) and their data to security domains. The hypervisor is in one security domain, along with all the VMs that do not take advantage of the ACM security capabilities called normal virtual machines (or NVMs). Each of the secure virtual machines (SVMs) is assigned to its own secure domain so that its data (and state) can be protected from the others. Secure entity identifiers (SEIDs) are used to keep track of the security domain to which a VM or page of memory belongs. Hardware enforces the isolation boundaries associated with security domains based on the SEIDs.



- A domain should not be able to access contents of pages of another domain
- Pages for all domains are allocated/mapped/unmapped by OS
- In particular, conventional OS can access pages of all domains
- We will demonstrate that, with ACM, even OS cannot access contents of pages of other domains
- When OS must handle a page of another domain, it is encrypted; it is decrypted when that page is returned to the owning domain

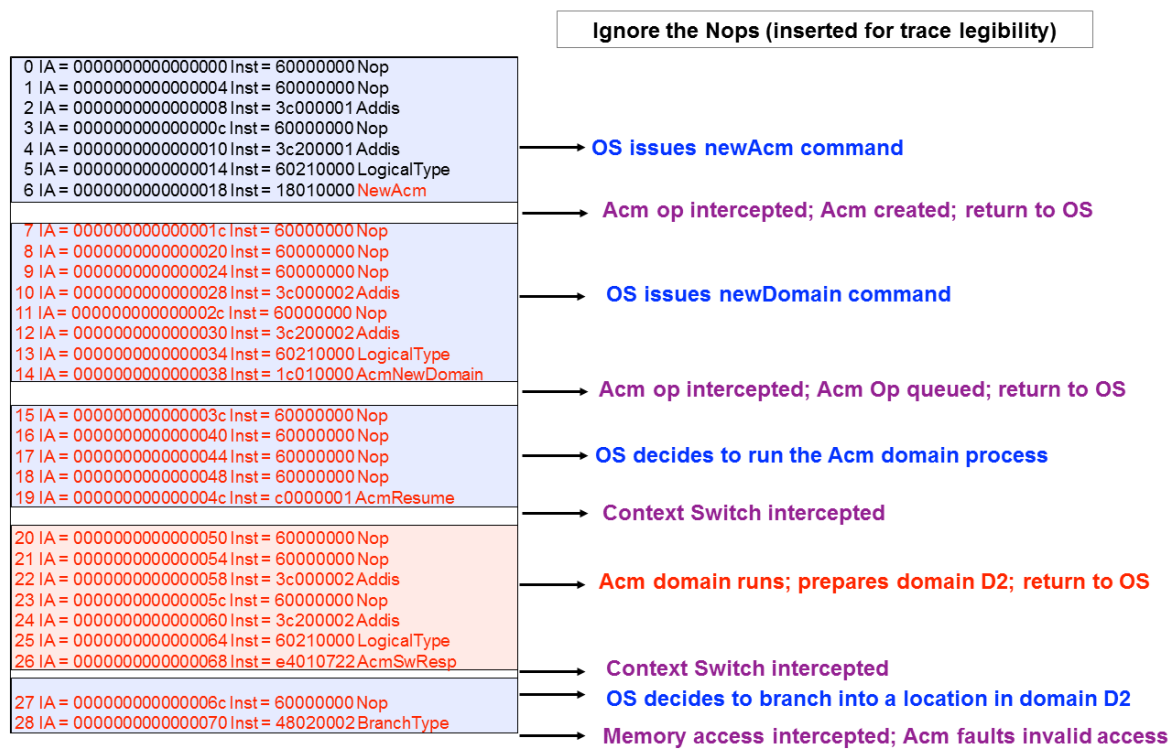
Figure 5 ACM concept: confidentiality and integrity by isolation



- Processor Bluespec/Verilog is available
- Acm Bluespec/Verilog is made ready
- Glue logic Bluespec/Verilog is made ready
  - to intercept load/store operations
  - to intercept events causing context switch including ACM ops
  - 7 New instructions added to ISA
- All the above components are stitched together, compiled and resulting verilog is simulated to run any program in memory
- We write a program in assembly language and run it on the simulator and show how the domains are switched and how access faults are detected

Figure 6 ACM demonstration hardware preparation

ACM firmware runs in Ultravisor mode, which is a privilege level above that of the hypervisor. This firmware, along with the ACM hardware, is responsible for maintaining SEIDs associated



**Figure 7 ACM demonstration: program loaded into memory**

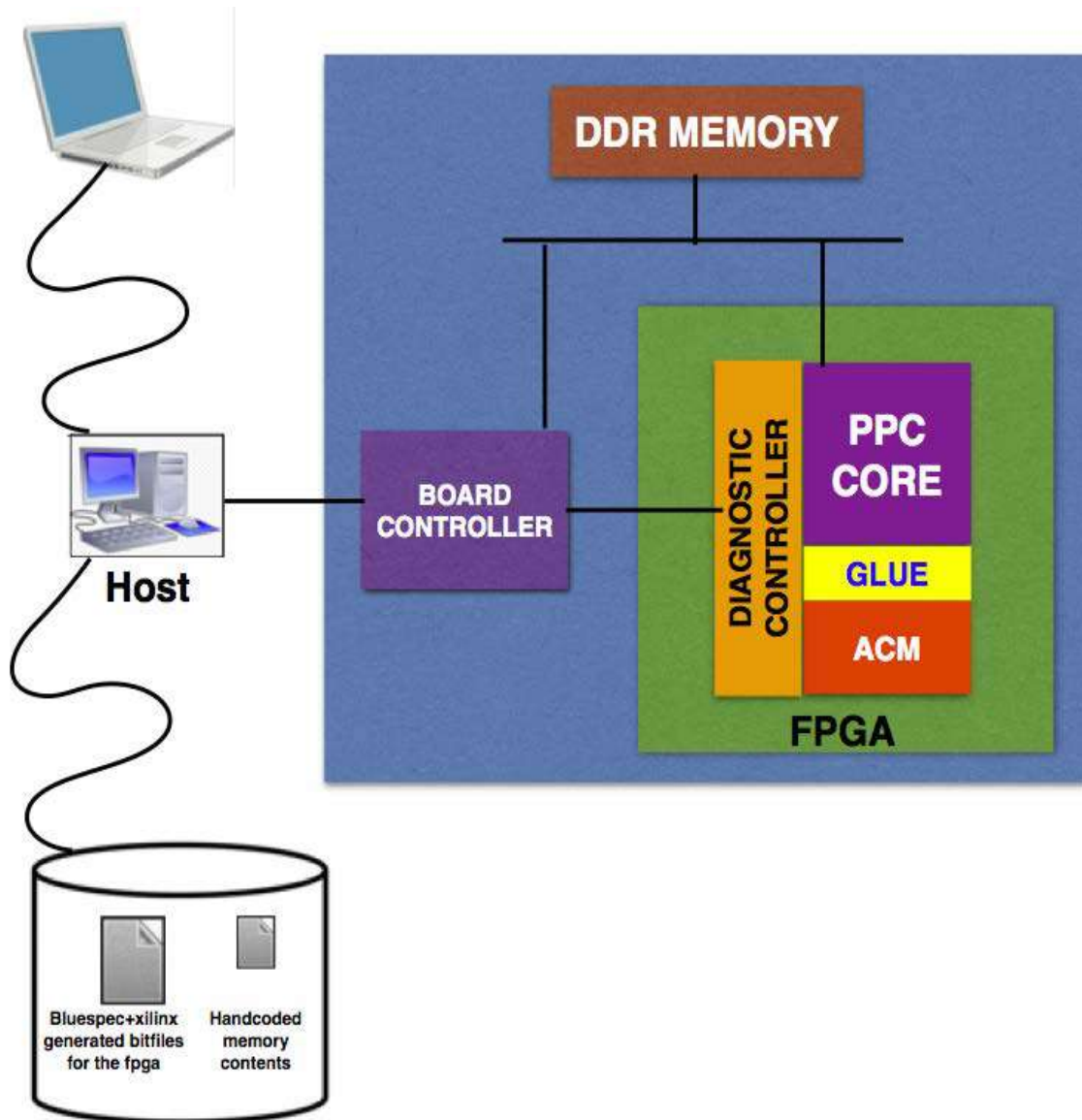
with processes and memory and enforcing the corresponding access restrictions. In addition, this firmware provides oversight of hypervisor services, such as page table management, that must be coordinated with SEID management. Finally, hardware mechanisms are in place to invoke ACM firmware when a thread transitions between security domains, so that the state of the process in one domain is not available to the process in the other domain.

In order to test the architecture modifications, it was necessary to build tooling to construct secure virtual machines as well as ACM firmware to support the modifications.

**ACM firmware Key Objectives, Requirements and Principles** There are multiple objectives that motivate the development of the ACM firmware software architecture.

- We want to further study how the different software components that comprise the ACM firmware will interact with all the components of an ACM enabled system. This includes low-level firmware, hypervisor, operating system, various libraries (including cryptographic) and applications. This is especially important to support ACM commercialization in IBM server platforms and to help IBM product and service groups build offerings around the prototype research technologies.
- A detailed software architecture is essential for performing a security evaluation of the system, which could be desirable.

- In addition to a security evaluation, detailed documentation of our software would be important for acceptance by the open source community, if and when parts of our overall architecture are open sourced. We note that IBM and IBM Research has been a leading contributor to open source security solutions, for example in the area of Linux security (Linux Integrity) and Trusted Computing.



**Figure 8 FPGA based emulation platform**

The ACM firmware provides ultravisor calls that manage SVMs, provides “shim” code for intercepted interrupts, handles protection violations, and provides oversight of hypervisor functions. Our objective is to make the ACM firmware as transparent as possible to other software running on a system and to have minimal performance overhead.

The following assumptions/requirements guided the initial development of the ACM firmware prototype. They were motivated by a desire to integrate with the IBM OpenPOWER and Power architectures.

- In steady state, (after it has started) the ACM firmware only responds to interrupts and it has no timer. The ACM firmware functions are time limited and no function should take longer than the hypervisor interrupt time. Similarly, the ACM firmware has no idle loop.
- The ACM firmware must be thread safe: it operates on the thread of the process that was active when it got control.
- When ultravisor mode is active, even when there is no ESM (Enter Secure Mode) instruction or ultravisor call, it must handle SEID faults and perform allowable operations on restricted registers. The ACM firmware provides oversight of hypervisor services, such as page table management, that must be coordinated with SEID management.
- The SEID table is an ultravisor mode resource, and therefore must be placed in storage to which only the ACM firmware has write access. Furthermore, the contents of the SEID Table must be such that non-ultravisor mode software cannot modify storage that contains ACM firmware programs or data.
- The ACM firmware is responsible for measurement and attestation of the Hypervisor (or the next program to execute) in order to verify its integrity. This function may also be performed by Host boot.

**Tooling for Building SVMs** Tooling needs to enable and support the following objectives:

- Provide protection for virtual machines running on Linux/KVM on Power (Open-POWER platform). This implies that virtual machines are protected from a potentially compromised Linux/LVM hypervisor/host and at the same time the Linux/KVM host is protected from the VMs.
- As envisioned by the SecureBlue++ model, the architecture should protect the virtual machine no matter where it is. That means unauthorized software cannot read or undetectably write the disk image or read/tamper with the VM while it is executing.
- Minimize changes to QEMU, Linux and KVM.
- Create secure virtual machines that run as guests, with one or more secure disks. The secure virtual machine (SVM) has all the capabilities of a regular virtual machine. First targets are Linux virtual machines.

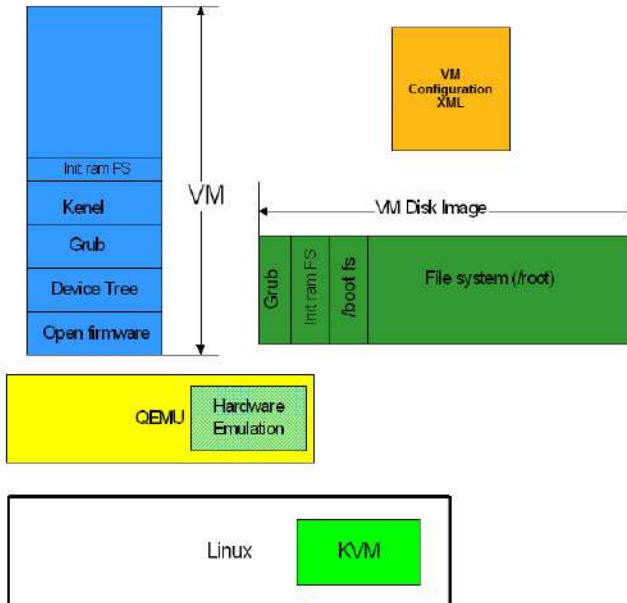
Figure 9 shows the Linux/KVM environment on POWER Systems and the steps involved in booting a Linux Virtual Machine. Furthermore, Figure 10 shows the interaction between the QEMU process and guest VMs.

Given this environment of Linux/KVM/QEMU and guest VMs on POWER Systems, we proposed the conceptual view of SVMs (Secure VMs) in Figure 11.

Our proposed SVM is a modified VM image that consists of the following:

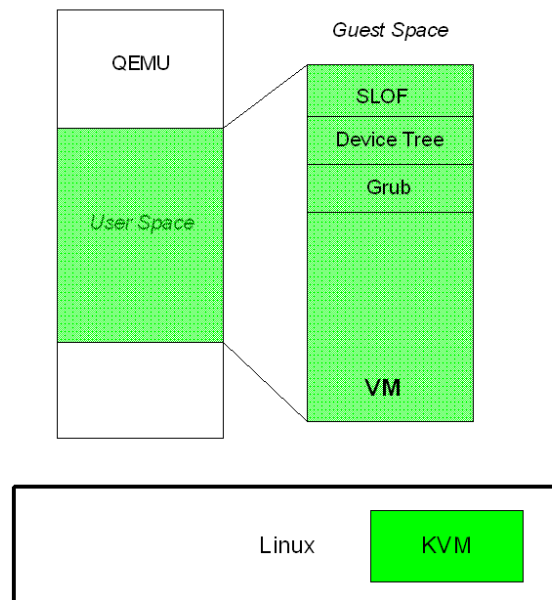
- Customized bootloader containing:
  - Boot wrapping code instrumented with ultracalls, e.g. ESM, and embedded Linux kernel with Petitboot application
  - Secure Object - encrypted with the ACM public key and decrypted and interpreted by the ACM firmware
- Boot partition integrity protected w. digital signatures

- **Linux/KVM runs natively on the bare-hardware**
  - KVM is a kernel module, exploiting hardware virtualization capabilities (e.g., virtual CPUs)
- **QEMU hosts guest VMs on KVM**
  - runs as a user process on top of Linux with KVM
  - manages VMs (create, stop/start, delete, etc.)
  - handles IO between guest and KVM
- **Guest consists of configuration file and image file containing an OS**
- **To start a virtual machine QEMU**
  - builds device tree
  - loads SLOF (open firmware)
- **Execution of the virtual machine starts in SLOF and then moves to Grub**
- Grub uses SLOF to load the Linux kernel and the initram from the disk Image, and then transfers control to the kernel
- **The kernel runs and mounts the file system**
  - Open firmware is discarded at some point
  - Device tree is internalized so the initial device tree is discarded
  - Grub is discarded once the kernel is up
- **User-space VM management/control is via libvirt tools**
  - virsh, virt-install, etc.



**Figure 9 Overview of the Linux/KVM environment on Power systems**

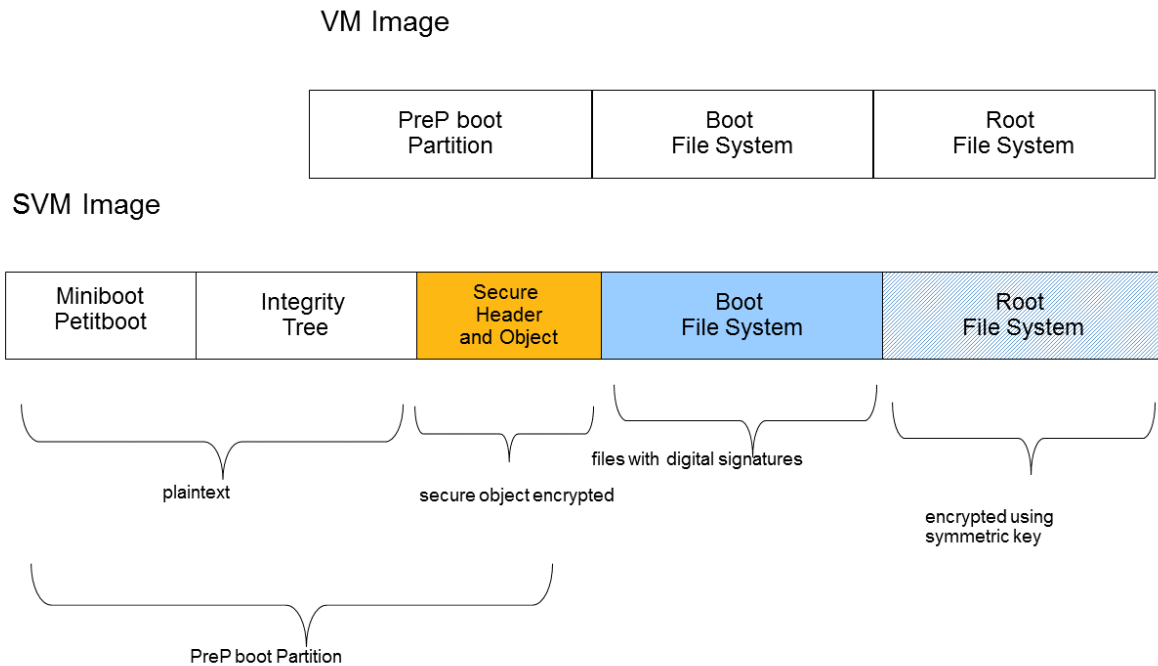
- QEMU/KVM is a type 2 hypervisor
- QEMU functions by mapping the guest address space inside of it's space
- QEMU starts with full addressability to the entire guest
- Once execution passes to SLOF or Grub, QEMU no longer needs free access into the VM
  - We should not trust QEMU
- The interface between the guest VM and QEMU is PAPR (hcalls, as implemented by KVM & QEMU)
- QEMU uses a syscall interface to talk to Linux/KVM
- We propose using the coarse-grained version of SB++/ACM to protect virtual machines



**Figure 10 QEMU Process and guest system relationship**

We studied the Open Firmware boot process and created a bootloader for booting VMs in the PowerKVM environment. The bootloader is the first step toward creating a SVM. We also identified steps for deriving a SVM from an existing VM image at rest; experimented with Buildroot ([git://git.buildroot.net/buildroot](http://git://git.buildroot.net/buildroot)) and Petitboot ([git://ozlabs.org/jk/petitboot](http://git://ozlabs.org/jk/petitboot)) open

source tools for creating a bootstrap loader for VM images; and reviewed documentation and source code on the Open Firmware boot process.



**Figure 11 Conceptual view of SVM**

Using the Buildroot tool, a tool to automate the creation of Linux images for embedded systems, we created our initial bootloader for booting an SVM. This bootloader consisted of custom Linux kernel with an embedded initramfs containing the Petitboot application, a user level application bootloader based on kexec. We employed Petitboot to boot the VMs from their image files since it is the bootloader used to natively boot Linux on Power Systems. We demonstrated that our custom Linux can boot VM images from the Linux command line using the `qemu-system-ppc64` command. We also tested booting the VM image by writing a zImage version of the bootloader into the VMs PreP boot partition. A zImage is a compressed kernel image wrapped with bootstrapping code.

We created a bash script called `writerep.sh`. This script writes an ELF file into the VM images PReP partition by associating a loop back device with the image and using the `dd` command to write the ELF file into the image. The script requires the following input parameters: VM image, bootloader (ELF file), and VM name. The `writerep.sh` script uses the zImage version of our bootloader as an input parameter.

We reviewed the IEEE Standard for Boot Firmware (1275-1994) and the Standard for Embedded Power Architecture Platform Requirements (ePAPR) Version 1.1 documentation for information on device tree format and the client program interface for accessing Open Firmware services. Additionally, we examined the source code of the Slimline Open Firmware (SLOF, <http://git.qemu.org/SLOF.git>) which is an open source implementation of the IEEE 1275-1994 standard.



## 4. RESULTS AND DISCUSSION

### 4.1. MOBILE PLATFORM RESULTS

This project built the components required to realize the architecture illustrated in Figure 1. This involved working with the previously existing technologies identified in section 3.1. The primary work involved extensions to OAT, adaptation of the IMA policy and integration with QEMU in a mobile platform. During the course of this project our objectives were changed based on technology developments described in section 4.1.4 below.

**4.1.1. IMA-APPRAISAL-IMASIG TEMPLATE PATCHES AND POLICIES** The Linux integrity work provided two distinct, but separate integrity models: one “trusted” computing model, rooted in a hardware TPM, and the second “secure” computing model, rooted in secure boot. The trusted model records hashes of files in a measurement list, which is anchored in platform configuration register (PCR) 10 of a hardware TPM (or virtualized TPM for guest virtual machines).

Prior to our project work was submitted and accepted into the Linux kernel 3.13, that allows integration of the two models, by creating a new measurement list template. This template adds the “secure” computing signatures to the measurement list, so that they too can be securely attested in the “trusted” computing list.

This project gains the benefit of hardware based attestation and the benefit of signature based verification, which provides authenticated provenance for all signed files.

**4.1.2. OAT ANALYSIS AND INTEGRATION** This project developed an extension to OpenAttestation (OAT) to support the verification based on the new integrated IMA attestation model (see section 4.1.4).

The existing OAT base validates only the boot aggregate PCR values (PCR 0-7). It has a client program that runs on the host or VM to be monitored, which signs these PCR values with a TPM Quote, and forwards the signed values to the OAT server, which stores the reports in a report database, and can verify and display the reports on an integrity web portal. The database stores a single “good” value for each PCR, based on the value at client registration, and creates an alert if a new submitted report has PCR values that are different, or if the TPM signature does not validate.

This project extended the OAT data to include all of the IMA measurements and signatures. A new verification program was written, which validated all signatures against the corresponding public keys (collected from the client at registration). Errors were noted if:

- The overall list did not validate to the TPM Quote
- A signature did not verify against the indicated key
- A signature verified, but against an untrusted key
- A file was not signed

These OAT extensions have been demonstrated and tested in a prototype environment.

**4.1.3. QEMU INTEGRATION WITH CUSE BASED SWTPM** For virtual machines on QEMU/KVM to take advantage of the Linux integrity model, QEMU has to provide at least a

software emulation of a per-VM TPM. Our original plan was to integrate our software emulation of a TPM (swTPM - <http://ibmswtpm.sourceforge.net>) directly with QEMU as a shared library (libtpms).

However, the QEMU maintainer has not accepted our submitted patches to do this, but did accept a simple patch that provides a “passthrough” driver linking a single VM to the native hardware (via /dev/tpm0).

For now, we have developed a work around which requires no changes to the guest or native kernels, QEMU, or libtpms. The basic approach is to use the existing “cuse” (character driver in user space) kernel module in the native kernel, to connect a user space application (“tpm server cuse”) to virtual devices which appear as /dev/vtpm\*. The passthrough driver does accept a pathname to the host TPM, so we can redirect it to /dev/vtpm\*, so long as the cuse driver duplicates the functionality of the /dev/tpm\* device driver.

For full support of all TPM functionality, we add a few ioctls to the cuse driver, so that in the future the QEMU passthrough driver will be able to support hardware level TPM features, such as TPM Init, for more complete emulation.

A backend script for virt-manager/libvirt was developed, to startup the cuse drivers for each VM instance, and to pass the needed passthrough driver command line parameters to QEMU, without having to modify libvirt for now.

**4.1.4. CHANGES TO THE ORIGINAL PLANOAT** This work was done with members of the OpenAttestation community, who completed version 1.7 of OAT (released 3/25/2014) which includes a framework for OAT verification and reporting of IMA measurements. This framework provides for the necessary external verification programs,

Originally, we had planned on integrating this support into the latest OAT version 2.1, but the 2.1 design and implementation is so different, that any port from 1.7 to 2.1 will require a significant effort. Consequently, our work remained on 1.7 for the mobile prototype.

**Containers** During the course of this project, containers including Docker and Linux containers (LXC) have become increasingly popular, due to security improvements, most notably the completion and upstreaming of user namespaces in the Linux Kernel. A separate project at IBM, which members of this team led, did a detailed security analysis of containers (particularly Docker) as candidates to replace virtualization as a secure isolation technology. Modern containers, particularly LXC are a reasonably secure alternative to virtualization. Because containers have such significantly lower space and performance overhead compared to virtualization, we view them as a prime alternative for mobile isolation.

This project completed a prototype for IMA measurement and appraisal for LXC containers, with extensions to OAT to verify the attestations by container. This prototype successfully demonstrated the ability of the OAT system to verify attestations at all levels, including native, VM, container, and containers in VMs. The work included patches to IMA to include container mount point information in every measurement list entry, and modifications to the OAT appraiser

to verify each container independently. The IMA patches have been posted for review, and we anticipate they will be accepted.

**IMA** A more complete IMA policy was developed that guarantees integrity of the entire TCB with digital signatures, while avoiding the problem of false alarms from changing files. This avoids having to maintain a database of measurements of files that are read (but not executed) by root processes.

This new design involves locking of "mutable" TCB files. The TCB was mapped by logging of IMA-appraisal data. As files were opened, mapped, or executed, we logged details on who made the request, the owner of the file, and which process was making the request. With this data we mapped the files that were (and were not) in the TCB. Most of the TCB files (roughly 3000) were immutable files, such as ELF executables, shared libraries, and interpreted executables, and were thus easy to lock down with an IMA policy. There were a smaller set (roughly 1000 files), which were read by root, which were more problematic. Some of these, such as interpreted code being read into an interpreter could be made immutable. Others need to remain mutable to root, or the system cannot boot or perform updates.

A patch to IMA was developed and tested to enable IMA to boot in a mode which allows mutable files to be updated while maintaining trusted appraisal hashes. Then, after the system is booted and updated, the systemd scripts can "lock" IMA-appraisal not to allow any further updates to any appraised files, whether signed or hashed. In addition, this new "locked" mode blocks renaming, blocks changing the security.ima xattr, blocks unlinking of TCB files, and locks all root owned directories, so that TCB files cannot be replaced or changed in any way, even by a root privileged attacker. All of this locking is under IMA policy control.

This design was implemented. The demonstration showed that it successfully boots and updates a full Fedora 20 system, and then the system can be locked against root attack, without affecting any user level processes or applications.

**4.1.5. STATUS** During this portion of the project, we completed and made available the following:

- Upstreamed the OAT extensions for IMA and IMA-sig.
- Upstreamed modifications to the QEMU passthrough driver.
- Hosting of a tpm server cuse application that became available in distributions such as Fedora and Redhat Enterprise Linux (RHEL).
- Completed "locking" extensions to IMA.
- Posted tools for signing .deb and rpm packages.
- Developed, tested, and posted IMA and OAT extensions for containers.

## **4.2. SERVER RESULTS**

### **4.2.1. ACM BLUESPEC FPGA MODEL**

**Isolation** This project built a demonstration of our Bluespec approach to ACM. Figure 12, Figure 1, Figure 14 and Figure 15 depict a 4 demonstration of the isolation capabilities of acm . The full

demonstration of this part of the technology was shown at the public demonstration event organized by DHS S&T CSD R&D Showcase in December 2014.

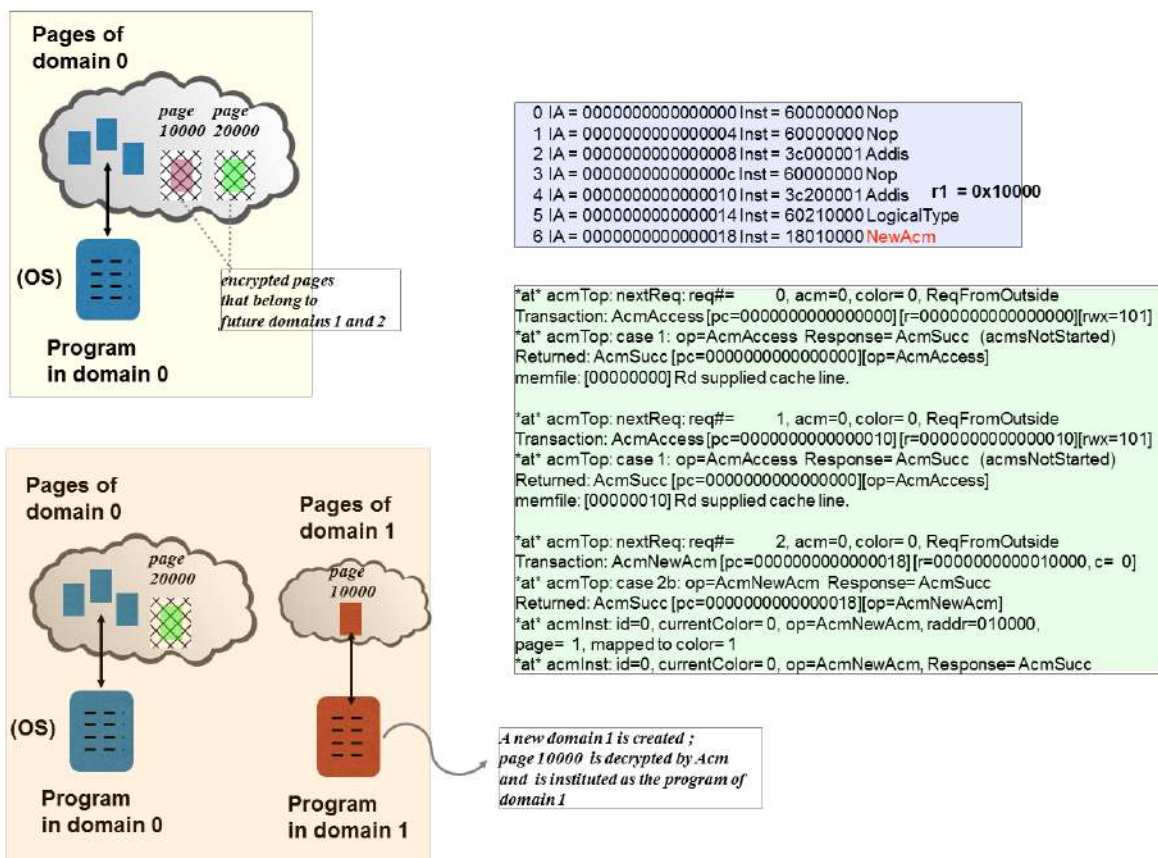


Figure 12 ACM demonstration: step 1 - creation of initial ACM

**FPGA implementation of PPC core with Access Control Monitor** In paragraph 4.2.1 we showed how ACM enforces secure process isolation on a Verilog simulator. In this section we will show how the Verilog models were mapped to our FPGA emulation platform.

Figure 16 shows the process isolation demo code in which the OS tries to branch into the location owned by a secure application or domain and is stopped by the ACM by redirecting the execution to an ACM fault handler. Figure 17 shows the process isolation demo code in which one secure domain tries to access the contents of another secure domain and is denied by the ACM. Once again, execution control is transferred to an ACM fault handler. These two figures illustrate the process isolation capability of an ACM enhanced PPC core.

In the next sections we will describe and demonstrate the advanced features of ACM model on FPGA hardware, including support for hierarchical ACMs and routing of interrupts from various hierarchies.

**Access Control Monitor enforced process hierarchy on FPGA platform** In this paragraph we show how we extended our ACM model and the Power core to support secure process hierarchies.

Secure process hierarchy is a unique and innovative concept, which we have brought to the secure processor design.

Hierarchical ACM allows a secure process to create child secure processes with parent secure process having no access to memory pages owned by child and vice versa. To further illustrate the advantage of this feature, a Secure Virtual Machine (SVM) could be run on processor with hierarchical ACMs. This SVM could then run an untrusted Operating System (OS) and several secure as well as un-secure processes on top of this OS. The hierarchical ACM will dynamically track pages owned by each process entity at various levels of the software stack and enforce strict hardware based isolation rules between them. In order to accommodate hierarchical ACMs several changes had to be made in the processor pipeline.

1. Instruction format: Power architecture has fixed length instructions. The new instructions added to support ACM now required more than six arguments. We decided to pass these arguments through General Purpose Registers (GPRs). Seeding specific GPRs with the right argument thus becomes the part of a software specification. Thus, all ACM instructions were changed to mnemonic only instructions. A thread stall mechanism had to be developed so that the execution pipeline could be suspended until all GPRs containing the ACM instruction arguments could be accessed in order to pass them to the ACM hardware.
2. ACM request-response interface: To support multiple hierarchies ACM could require multiple number of cycles to respond to requests, hence an asynchronous interface was adopted between the processor pipeline and ACM hardware. Making the interface

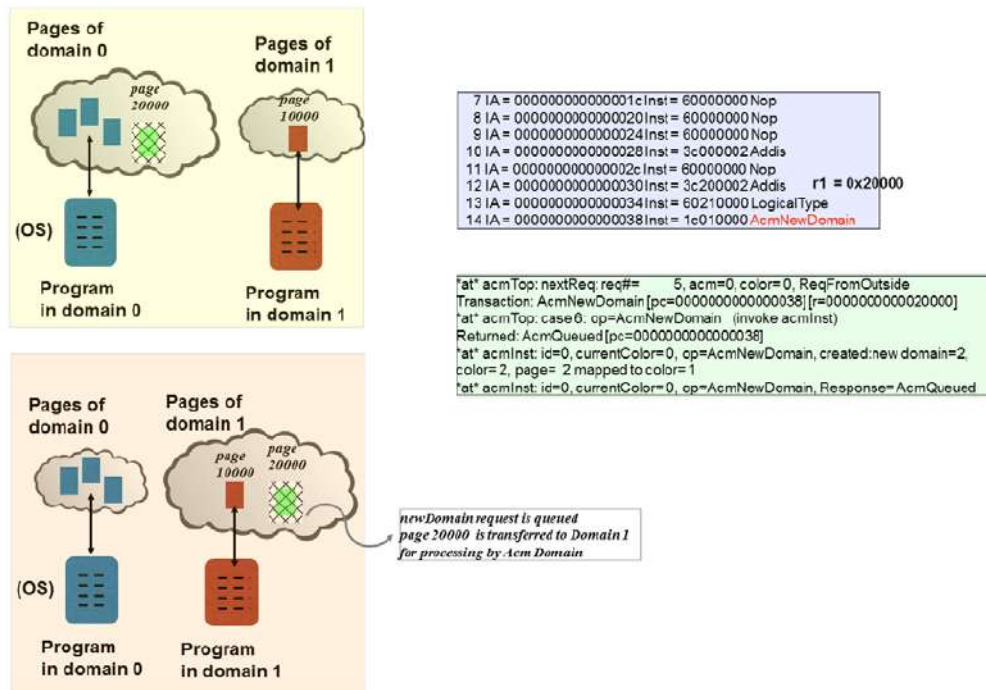
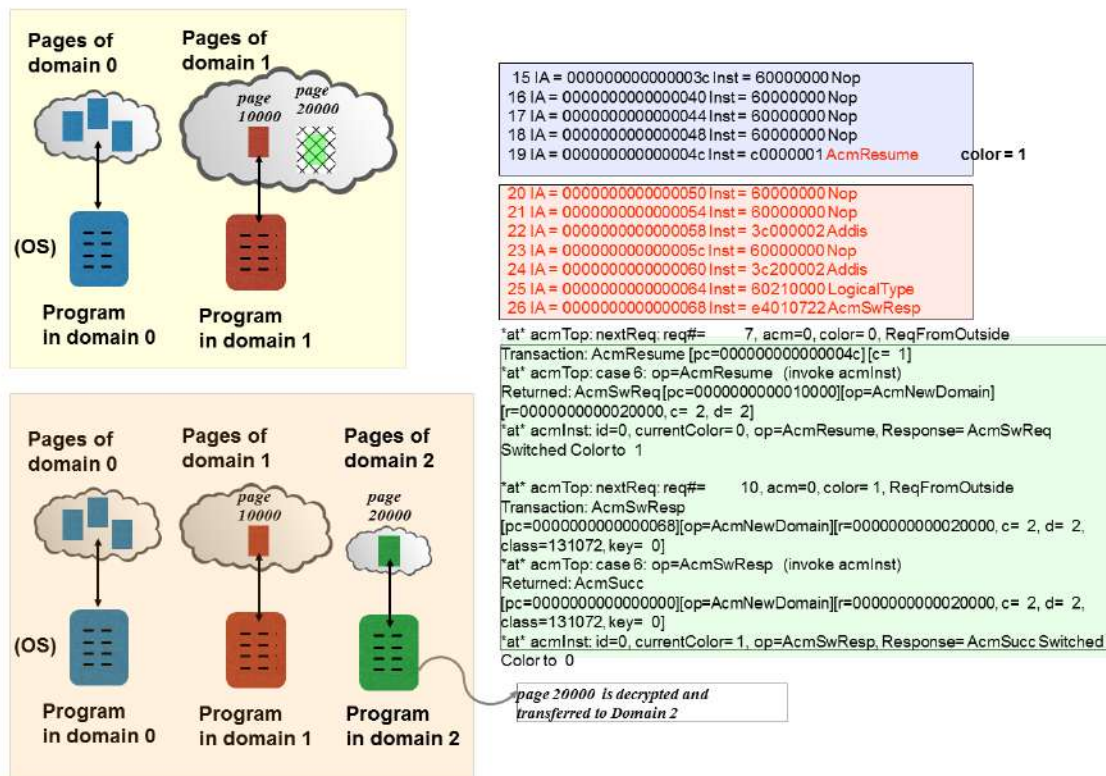


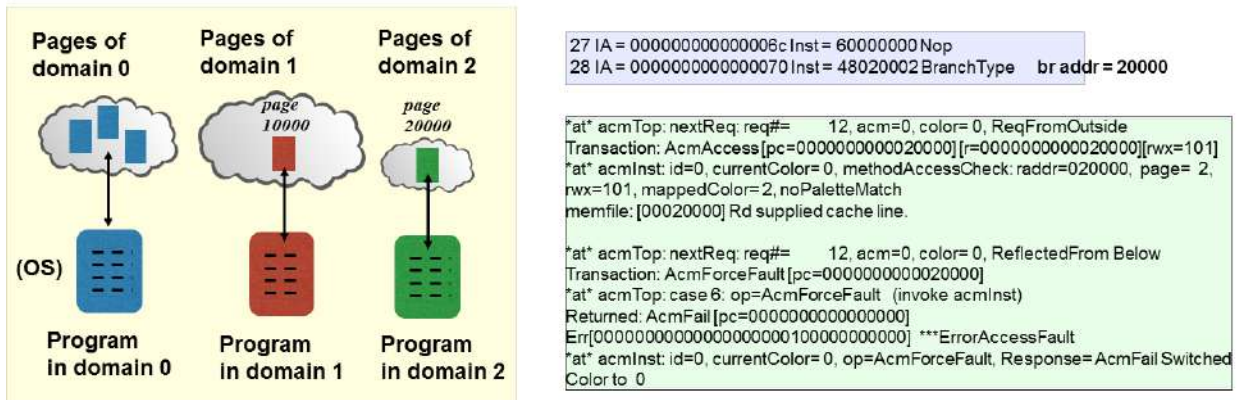
Figure 13 ACM demonstration: step 2 - domain D2 requested



**Figure 14 ACM demonstration: step 3 domain D3 completed**

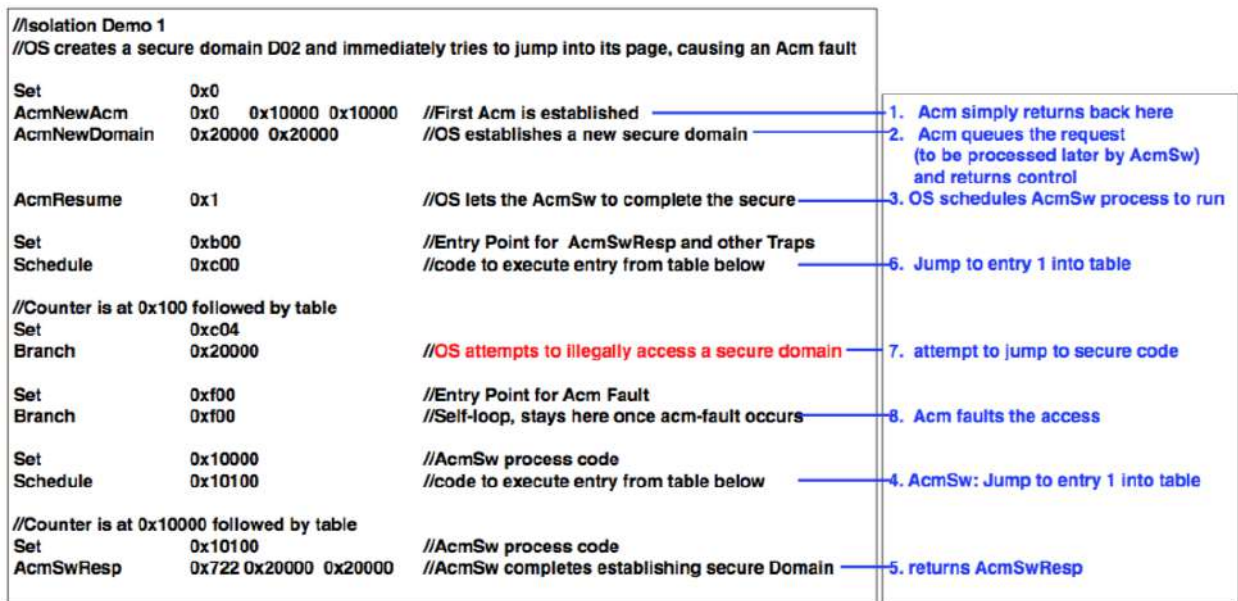
asynchronous increased the design flexibility but introduced the problem of inability of ACM to stop execution of instructions which caused an ACM fault. This could be prevented by suspending the thread for arbitrary amount of period till the ACM responded but this design choice was dropped as it has a huge penalty on performance. An alternative approach of ACM responses to be handled as asynchronous interrupts was adopted. ACM responses were divided into two parts ones causing access fault must be returned in single cycle and the second ones like creating domains or requesting service of ACM software processes could take an arbitrary number of cycles. As a result, the multi cycle responses could be handled by an interrupt mechanism without stalling the thread.

3. Instruction fetch caused ACM faults: The instruction fetch stage of the processor pipeline has a buffer to hide the latency of fetching from memory. The fetch stage fills this buffer based upon an algorithm which minimizes any present or future fetch latencies. All fetch requests are approved by the ACM before completion. Certain ACM instructions or an ACM response can cause context switch forcing the program counter to change to a branch address. The prefetch algorithm cannot predict this and thus fetches the instructions sequentially until its buffer is flushed. These fetches will be faulted by the ACM as the ACM expects the next instruction to be fetched from the branch address, thus creating an infinite loop of context switch and fetch faults. In order to solve this problem, we modified the pipeline to carry fault tags with instructions that had caused fetch faults and allowed the ACM to change its context only when the instruction carrying the fault flag arrived at the execute stage.



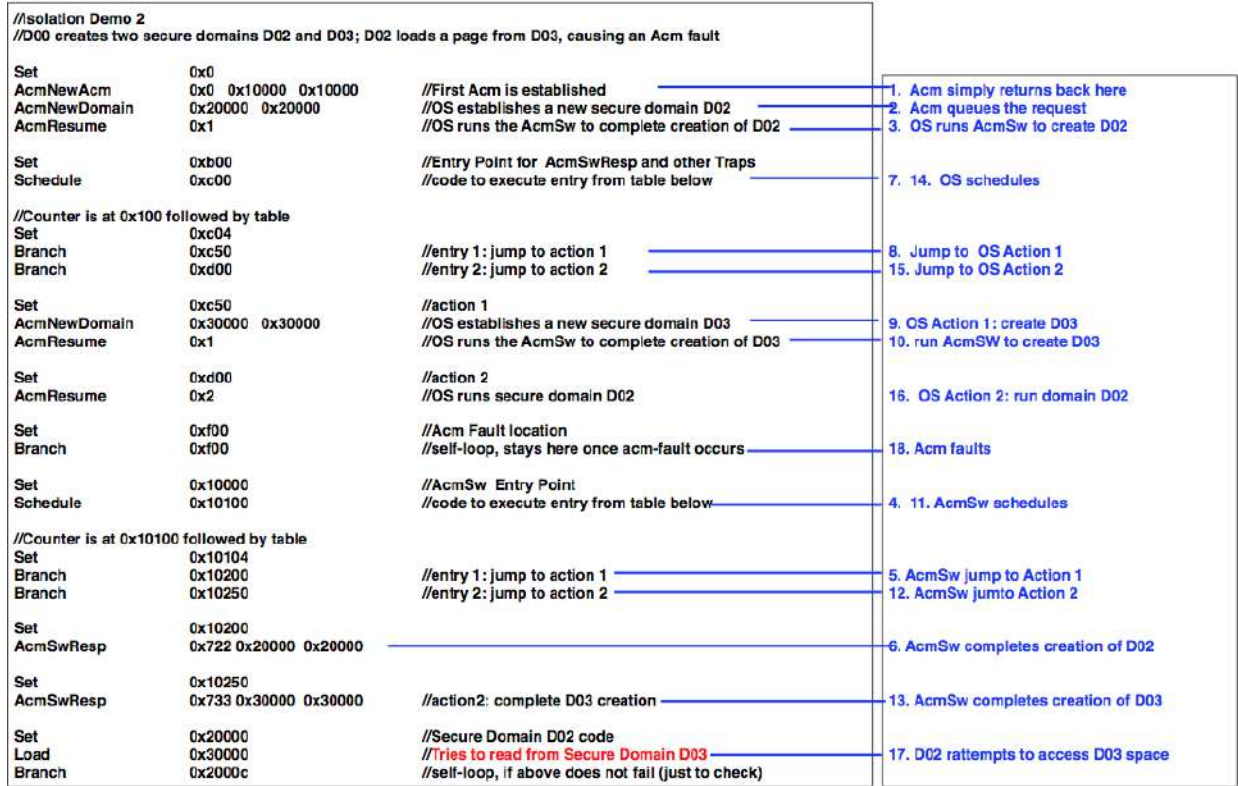
**Figure 15 ACM demonstration: Step 4 - OS attempts invalid access**

After making these three architectural changes, we enhanced the ACM model to maintain multiple hierarchical contexts. All the models were first tested using a Verilog simulator. We then synthesized these models to run on our FPGA platform.



**Figure 16 Process isolation OS jumps to secure code**

Figure 18 shows an illustration of the hierarchy demo. All the code is written in assembly language segments. First, an ACM data structure is created at level 0 by ACM hardware. This ACM data structure is then populated with two domains - OS (D00) and ACM software (D01). The OS at D00 requests the ACM hardware to create two new secure domains. ACM hardware allocates the two new secure domains (D02 and D03) different colors. To create another hierarchy the OS at level 0 requests the creation of another ACM at level 1 using the color allocated to one of the



**Figure 17 Process isolation secure domain invalid access**

secure domains (D03). The new ACM at level 1 data structure is then created by ACM hardware. The ACM hardware repeats the operation of populating the new ACM at level 1 data structure with colors for OS at level1 (D10) and ACM software at level 1 (D11). OS at level 1 then requests creation of a secure domain D12 at level1. We further demonstrate that when secure domain D12 tries to load the contents from a page that belongs to secure domain D02, the ACM hardware disallows this load operation by creating an ACM fault and forcing a branch to ACM fault handler at level0. This is a policy choice. We made this choice to enable processes at lower level in hierarchy to kill processes in higher level if a fault was committed against them. The following page explains the various code segments of the demo and corresponding program flow. The OS code at various levels as well as the ACM software code uses a simple scheduler to perform multiple tasks. This scheduler code was described in section 3.2.3 and is used as is.

### Hierarchy Demonstration 1

**//D00 creates two secure domains D02 and D03;**  
**//D00 creates new acm D10 using color of D03;**  
**//D10 creates secure domain D12; D12 tries to load from D02 creating an access fault**

<b>Set</b>	<b>0x0</b>	
<b>AcmNewAcm</b>	<b>0x10000 0x10000 0x0 0x0 0x0 --&gt;</b>	<b>Step1: OS requests creation of ACM0</b>
<b>AcmNewDomain</b>	<b>0x20000 0x20000 -----&gt;</b>	<b>Step2: OS requests creation of secure domain D02</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step3: OS notifies ACM of scheduling acmSw process</b>
<b>Set</b>	<b>0xb00 -----&gt;</b>	<b>Step8, 14: OS resumes control after acmSw releases</b>
<b>Schedule</b>	<b>0xc00 -----&gt;</b>	<b>Step9, 15: OS schedules the next process</b>
<b>Set</b>	<b>0xc10</b>	



<b>Branch</b>	<b>0xc50 -----&gt;</b>	<b>Branch address for OS scheduled process</b>
<b>Branch</b>	<b>0xd00 -----&gt;</b>	<b>Branch address for OS scheduled process</b>
<b>Set</b>	<b>0xc50</b>	
<b>AcmNewDomain</b>	<b>0x30000 0x30000 -----&gt;</b>	<b>Step10: OS requests creation of secure domain D03</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step11: OS notifies ACM of scheduling acmSw process</b>
<b>Set</b>	<b>0xd00</b>	
<b>AcmNewAcm of D03</b>	<b>0x40000 0x40000 0x60000 0x60000 0x3 ----&gt;</b>	<b>Step16: OS requests creation of ACM1 using color</b>
<b>AcmResume</b>	<b>0x3 -----&gt;</b>	<b>Step17: OS notifies ACM0 of scheduling D03</b>
<b>Set</b>	<b>0xf00 -----&gt;</b>	<b>Step28: Level0 fault handler when ACM denies D12 to D02 load</b>
<b>Branch</b>	<b>0xf00</b>	
<b>Set</b>	<b>0x10000 -----&gt;</b>	<b>Step4: PC changes to acmSw address</b>
<b>Schedule</b>	<b>0x10100 -----&gt;</b>	<b>Step5: acmSw scheduler code branches to addr 10200</b>
<b>Set</b>	<b>0x10110</b>	
<b>Branch</b>	<b>0x10200 -----&gt;</b>	<b>Step6: acmSw at level 0 runs the first time</b>
<b>Branch</b>	<b>0x10250 -----&gt;</b>	<b>Step12: acmSw at level 0 runs the second time</b>
<b>Set</b>	<b>0x10200</b>	
<b>AcmSwResp</b>	<b>0x7 0x20000 0x20000 0x2 0x2 --&gt;</b>	<b>Step7: acmSw finishes creating secure domain D02</b>
<b>Set</b>	<b>0x10250</b>	
<b>AcmSwResp</b>	<b>0x7 0x30000 0x30000 0x3 0x3 --&gt;</b>	<b>Step13: acmSw finishes creating secure domain D03</b>
<b>Set</b>	<b>0x40000 -----&gt;</b>	<b>Step21: PC changes to acmSw address at level 1</b>
<b>Schedule</b>	<b>0x40100 -----&gt;</b>	<b>Step22: acmSw scheduler code branches to addr 40100</b>
<b>Set</b>	<b>0x40110</b>	
<b>Branch</b>	<b>0x40200 -----&gt;</b>	<b>Step23: acmSw at level 1 runs the first time</b>
<b>Set</b>	<b>0x40200</b>	
<b>AcmSwResp</b>	<b>0x7 0x50000 0x50000 0x2 0x2 ---&gt;</b>	<b>Step24: acmSw finishes creating secure domain D12</b>
<b>Set</b>	<b>0x50000</b>	
<b>Load</b>	<b>0x20000 -----&gt;</b>	<b>Step27: D12 tries to load D02</b>
<b>Branch</b>	<b>0x50000 -----&gt;</b>	<b>If above load passes loop!</b>
<b>Set</b>	<b>0x60000 -----&gt;</b>	<b>Step18: OS at level1 gains control</b>
<b>AcmNewDomain</b>	<b>0x50000 0x50000 -----&gt;</b>	<b>Step19: OS level1 requests creation of secure domain</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step20: OS notifies ACM1 of scheduling acmSw</b>

## Hierarchy Demonstration 2

In the second demo we extend the example in demo1 by creating two secure domains at level 1 (D12 & D13). We then try to load contents of D13 from D12. The ACM denies this load but this time it transfers the control to the fault handler lying at level 1 thus illustrating the hierarchy of privileges brought in by a hierarchical ACM architecture.

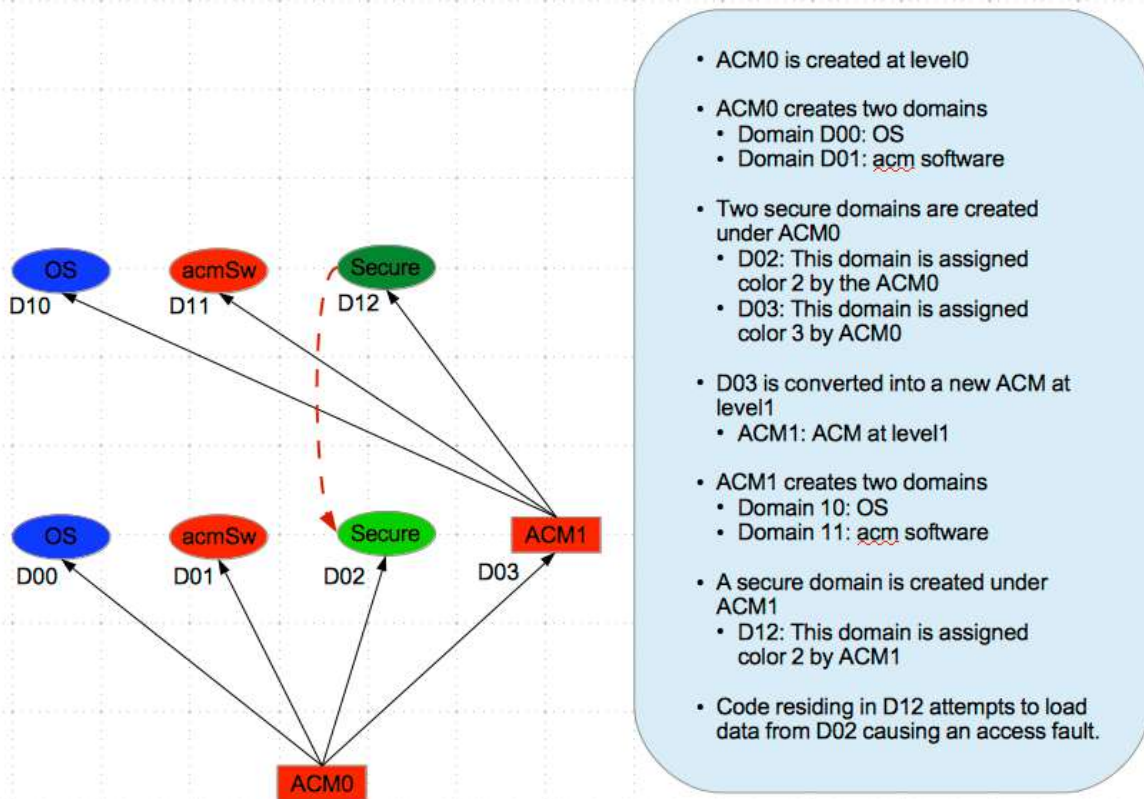
```
//D00 creates two secure domains D02 and D03;
//D00 creates new acm D10 using color of D03;
//D10 creates secure domain D12; D12 tries to load from D02 creating an access fault
```

<b>Set</b>	<b>0x0</b>	
<b>AcmNewAcm</b>	<b>0x10000 0x10000 0x0 0x0 0x0 ----&gt;</b>	<b>Step1: OS requests creation of ACM0</b>
<b>AcmNewDomain</b>	<b>0x20000 0x20000 -----&gt;</b>	<b>Step2: OS requests creation of secure domain D02</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step3: OS notifies ACM of scheduling acmSw process</b>
<b>Set</b>	<b>0xb00 -----&gt;</b>	<b>Step8, 14: OS resumes control after acmSw releases</b>

<b>Schedule</b>	<b>0xc00 -----&gt;</b>	<b>Step9, 15: OS schedules the next process</b>
<b>Set</b>	<b>0xc10</b>	
<b>Branch</b>	<b>0xc50 -----&gt;</b>	<b>Branch address for OS scheduled process</b>
<b>Branch</b>	<b>0xd00 -----&gt;</b>	<b>Branch address for OS scheduled process</b>
<b>Set</b>	<b>0xc50</b>	
<b>AcmNewDomain</b>	<b>0x30000 0x30000 -----&gt;</b>	<b>Step10: OS requests creation of secure domain D03</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step11: OS notifies ACM of scheduling acmSw process</b>
<b>Set</b>	<b>0xd00</b>	
<b>AcmNewAcm of D03</b>	<b>0x40000 0x40000 0x60000 0x60000 0x3 ----&gt;</b>	<b>Step16: OS requests creation of ACM1 using color</b>
<b>AcmResume</b>	<b>0x3 -----&gt;</b>	<b>Step17: OS notifies ACM0 of scheduling D03</b>
<b>Set</b>	<b>0xf00 -----&gt;</b>	
<b>Branch</b>	<b>0xf00</b>	<b>Level0 fault handler when ACM denies D12 to D02 load</b>
<b>Set</b>	<b>0x10000 -----&gt;</b>	<b>Step4: PC changes to acmSw address</b>
<b>Schedule</b>	<b>0x10100 -----&gt;</b>	<b>Step5: acmSw scheduler code branches to addr 10200</b>
<b>Set</b>	<b>0x10110</b>	
<b>Branch</b>	<b>0x10200 -----&gt;</b>	<b>Step6: acmSw at level 0 runs the first time</b>
<b>Branch</b>	<b>0x10250 -----&gt;</b>	<b>Step12: acmSw at level 0 runs the second time</b>
<b>Set</b>	<b>0x10200</b>	
<b>AcmSwResp</b>	<b>0x7 0x20000 0x20000 0x2 0x2 -----&gt;</b>	<b>Step7: acmSw finishes creating secure domain D02</b>
<b>Set</b>	<b>0x10250</b>	
<b>AcmSwResp</b>	<b>0x7 0x30000 0x30000 0x3 0x3 -----&gt;</b>	<b>Step13: acmSw finishes creating secure domain D03</b>
<b>Set</b>	<b>0x40000 -----&gt;</b>	<b>Step21, 30: PC changes to acmSw address at level 1</b>
<b>Schedule</b>	<b>0x40100 -----&gt;</b>	<b>Step22, 31: acmSw scheduler code runs</b>
<b>Set</b>	<b>0x40110</b>	
<b>Branch</b>	<b>0x40200 -----&gt;</b>	<b>Step23: acmSw at level 1 runs the first time</b>
<b>Branch</b>	<b>0x40250 -----&gt;</b>	<b>Step32: acmSw at level 1 runs the second time</b>
<b>Set</b>	<b>0x40200</b>	
<b>AcmSwResp</b>	<b>0x7 0x50000 0x50000 0x2 0x2 ----&gt;</b>	<b>Step24: acmSw finishes creating secure domain D12</b>
<b>Set</b>	<b>0x40250</b>	
<b>AcmSwResp</b>	<b>0x7 0x70000 0x70000 0x3 0x3 ----&gt;</b>	<b>Step33: acmSw finishes creating secure domain D13</b>
<b>Set</b>	<b>0x50000</b>	
<b>Load</b>	<b>0x70000 -----&gt;</b>	<b>Step37: D12 tries to load D13</b>
<b>Branch</b>	<b>0x50000 -----&gt;</b>	<b>If above load passes loop!</b>
<b>Set</b>	<b>0x60000 -----&gt;</b>	<b>Step18: OS at level1 gains control</b>
<b>AcmNewDomain</b>	<b>0x50000 0x50000 -----&gt;</b>	<b>Step19: OS level1 requests creation of secure domain</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step20: OS notifies ACM1 of scheduling acmSw</b>
<b>Set</b>	<b>0x60b00</b>	
<b>Schedule</b>	<b>0x60c00 -----&gt;</b>	<b>Step 25, 34: OS level1 regains control after acmSw</b>
<b>Set</b>	<b>0x60c10</b>	
<b>Branch</b>	<b>0x60c50 -----&gt;</b>	<b>Step 26: OS level1 schedules first process</b>
<b>Branch</b>	<b>0x60d00 -----&gt;</b>	<b>Step 35: OS level1 schedules second process</b>
<b>Set</b>	<b>0x60c50 -----&gt;</b>	<b>Step 27: OS level1 scheduled process runs</b>
<b>AcmNewDomain</b>	<b>0x70000 0x70000 -----&gt;</b>	<b>Step 28: OS level1 process requests creation of domain D13</b>
<b>AcmResume</b>	<b>0x1 -----&gt;</b>	<b>Step 29: OS level1 notifies ACM1 of scheduling acmSw</b>

**Set** **0x60d00**  
**AcMResume** **0x2** -----> **Step 36: OS level1 notifies ACM1 of scheduling domain D12**  
  
**Set** **0x60f00** -----> **Step 38: Level1 fault handler when ACM denies D12 to D13 load**  
**Branch** **0x60f00**

## Hierarchy Demo



**Figure 18 Process hierarchy illustration**

Both code pieces shown in Demo 1 and Demo 2 are written in plain text format. A parser coded in Haskell takes the text file and generates assembly code. This assembly code is then compiled to generate a Hex file. The Hex file is then used to seed the memory model for Verilog simulations. The same Hex file is also used to create a binary file. The binary file is loaded using a backdoor DDR interface into the memory of the processor implemented on the FPGA platform. While the Verilog simulator gives a detailed trace as each instruction is executed, the FPGA system has a separate debug interface which lets the user read the contents of any register or memory location after execution of one or more instructions. We thus verified the concept of hierarchical ACM using the two simulations as well as the hardware platform. In the next section we will describe and demonstrate another unique feature of our secure processor architecture | sharing. This feature allows sharing of memory pages between two or more secure processes. This property will be demonstrated in both simulation and in hardware on our FPGA platform.

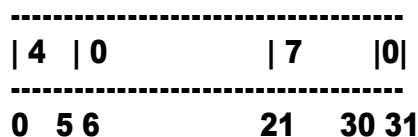
**Demonstration of Access Control Monitor enforced sharing of memory pages on FPGA platform** In previous paragraphs we described and demonstrated ACM enforced secure process hierarchy on the FPGA platform. We now describe how our ACM model and the Power core were

extended to support secure sharing of memory pages. Like secure process hierarchy, secure sharing of memory pages is a unique and innovative concept, which we have introduced to the secure processor design. Secure sharing allows a secure process to share its selectively chosen memory pages with another secure process. For example if a secure process needs to send data to another secure process for further computation on the data, it can now do so by sharing the memory page containing this data with another secure process. The ACM hardware will ensure that the secure processes with the right key will be the only processes that will have access to the shared pages. The ACM also offers further granularity with read only, write only, execute only or any combinations of access privileges for sharing of memory pages between processes. In order to enhance the ACM integrated Power core to allow hardware enforced secure sharing, many enhancements were made to the decode and execute pipeline.

Two new instructions were added with the following semantics:

**1. Instruction : AcmNewShare**

**Usage : AcmNewShare (GPR0), (GPR1)**



**GPR0 : Effective address of the shared page**

**GPR1 : Key associated with sharing**

**Function : Create a shared page**

**It can be issued by the secure domain owning the page**

**GPR0 contains the effective address of the page**

**GPR1 contains the identifying key**

**AcmRequest arguments :**

**Opcode : AcmNewShare**

**Program Counter : Program counter address**

**Argument List [0] : (GPR0) = Effective address of shared page**

**Argument List [1] : (GPR1) = sharing key**

**AcmResponse arguments :**

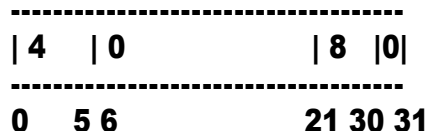
**Opcode : AcmSucc, AcmFail**

**Program Counter : Base effective address of next instruction**

**Argument List[0] : AcmNewShare (reflected request)**

**2. Instruction : AcmAddShare**

**Usage : AcmAddShare (GPR0),(GPR1)**



**GPR0 : Effective address of the shared page**

**GPR1 : Key associated with sharing**

**Function :** Request the permission to access a shared page.  
**It can only be issued by the secure domain requesting access to a page shared by another secure domain.**  
**GPR0 contains the effective address of the page**  
**GPR1 contains the identifying key**

**AcmRequest arguments :**

**Opcode :** AcmAddShare  
**Program Counter :** Current PC address  
**Argument List[0] :** (GPR0) = Effective address of shared page  
**Argument List[1] :** (GPR1) = sharing key

**AcmResponse arguments :**

**Opcode :** AcmSucc, AcmFail  
**Program Counter :** Base effective address of next instruction  
**Argument List[0] :** AcmAddShare (reflected request)

The general principle of operation is to first create secure domains. One of the secure domains then requests from the operating system a new page. On the FPGA platform model it then notifies the operating system that it would like to share this page with another process and associates a key with it using the instruction AcmNewShare. This instruction is trapped by the ACM hardware and an entry is made in its internal table for process versus page ownership. Any other process with the right key can request access to the page from the OS by using the command AcmAddShare. The ACM hardware traps this instruction and matches the key. If the match occurs, the ACM allows the program to proceed, if it fails the contents of the Program counter are modified to point to the fault location. There is no mechanism explicitly built in hardware for the sharing processes to exchange the keys. It is assumed that conventional methods of symmetric key encapsulation with public key cryptography will be used to perform the key exchange. The following two demos illustrate the basic working principle. The first demo shows how two secure domains can share a page. The second demo extends beyond the first demo where the OS tries to access the shared page between two secure domains and the access is denied.

### Sharing Demo 1

```

// Notation: Dij refers to j-th domain at the i-th level
// Thus, D00 = OS at level 0, D01 = AcmSw at level 0, D02, D03,... are other secure domains at level 0

// Thus, D10 = OS at level 1, D11 = AcmSw at level 1, D12, D13,... are other secure domains at level 1 and so on
// Sharing Demo 1

// D00 creates two secure domains D02 and D03; when it runs D02, D02 shares a page with D03

```

```

Set 0x0
AcmNewAcm 0x10000 0x10000 0x0 0x0 0x0 // Step1: First Acm is established
AcmNewDomain 0x20000 0x20000 // Step 2: OS establishes a new secure domain D02
AcmResume 0x1 // Step 3: OS runs the AcmSw to complete creation of D02
Set 0xb00 // OS trap location (in particular, AcmSwResp comes here)
Schedule 0xc00 // Step 8, 15, 19, 26, 31: Scheduler for system handler
Set 0xc10
Branch 0xc50 // Step 9: return after first acmSwResp
Branch 0xd00 // Step 16: return after second acmSwResp
Branch 0xd50 // Step 20: return after newShare
Branch 0xe00 // Step 26: return after acmMap
Branch 0xe50 // Step 32: return after addShare
Set 0xc50
AcmNewDomain 0x30000 0x30000 // Step 10: OS establishes a new secure domain D03
AcmResume 0x1 // Step 11: OS runs the AcmSw to complete creation of D03
Set 0xd00
AcmResume 0x2 // Step 17: OS runs secure domain D02 at 0x20000
Set 0xd50
AcmMap 0x40000 0x40000 0x4 // Step 21: OS notifies ACM of mapping new page
AcmResume 0x1 // Step 22: OS runs acmSw
Set 0xe00
AcmResume 0x3 // Step 27: OS runs secure domain D03 at 0x30000
Set 0xe50
AcmResume 0x3 // Step 33: OS maps allocates page to D03 also,
// runs secure domain D03 at 0x30000
// Acm Fault location
// Step 37: self-loop, stays here once acm-fault occurs
Set 0xf00
Branch 0xf00
Set 0x10000
Schedule 0x10100 // Steps 4, 12, 23: Scheduler for acmSw
Set 0x10110
Branch 0x10200 // Step 5: creation of D02
Branch 0x10250 // Step 13: creation of D03
Branch 0x10300 // Step 24: acm map for page share
Set 0x10200
AcmSwResp 0x10000002 0x20000 0x20000 0x2 0x2 // Step 6: new domain D02 created by acmSw,
// control goes back to OS 0xb00
Set 0x10250
AcmSwResp 0x10000002 0x30000 0x30000 0x3 0x3 // Step 14: new domain D03 created by acmSw,
// control goes back to OS 0xb00
Set 0x10300
AcmSwResp 0x10000006 0x40000 0x40000 0x4 // Step 25: acm map completed by acmSw,
// control goes back to OS 0xb00
Set 0x20000 // Secure Domain D02 code
AcmNewShare 0x40000 0x1729 // Step 18: Secure domain requests new page for sharing,
// control goes back to OS
// Secure domain D03 code
Set 0x30000 // Step 28, 34: Scheduler for secure domain
Schedule 0x30100
Set 0x30110
Branch 0x30200 // Step 29: Secure domain code for requesting shared page
Branch 0x30234 // Step 35: Secure domain code to attempt access to shared page
Set 0x30200
AcmAddShare 0x40000 0x1729 // Step 30: Domain D03 requests shared page access
// control goes back to OS 0xb00
Set 0x30234
Load 0x40000 // Step 36: Tries to read from Secure Domain D02
Branch 0x30240 // Step 37: self-loop if above load is successful
Set 0x30200
AcmAddShare 0x40000 0x1729 // Step 30: Domain D03 requests shared page access
// control goes back to OS 0xb00
Set 0x30234
Load 0x40000 // Step 36: Tries to read from Secure Domain D02
Branch 0x30240 // Step 37: self-loop if above load is successful

```

## Sharing Demo 2

```
// Notation: Dij refers to j-th domain at the i-th level
// Thus, D00 = OS at level 0, D01 = AcmSw at level 0, D02, D03,... are other secure domains at level 0
// Thus, D10 = OS at level 1, D11 = AcmSw at level 1, D12, D13,... are other secure domains at level 1 and so on
// Sharing Demo 2
// D00 creates two secure domains D02 and D03; D02, D03 share a page, OS tries to access the page.

Set          0x0
AcmNewAcm    0x10000 0x10000 0x0 0x0 0x0 // Step 1: First Acm is established
AcmNewDomain 0x20000 0x20000 // Step 2: OS establishes a new secure domain D02
AcmResume    0x1 // Step 3: OS runs the AcmSw to complete creation of D02

Set          0xb00 // Step 6, 13, 18, 25, 31: OS trap location
              (in particular, AcmSwResp comes here)
Schedule     0xc00 // Insert here OS scheduler code to jump to schedule at 0xc00

Set          0xc10
Branch       0xc50 // Step 7: return after D02 creation
Branch       0xd00 // Step 14: return after D03 creation
Branch       0xd50 // Step 19: return after new page for sharing request
Branch       0xe00 // Step 26: return to OS after map completion
Branch       0xe50 // Step 32: return to OS after add share by D03

Set          0xc50
AcmNewDomain 0x30000 0x30000 // Step 8: OS establishes a new secure domain D03
AcmResume    0x1 // Step 9: OS runs the AcmSw to complete creation of D03

Set          0xd00
AcmResume    0x2 // Step 15: OS runs secure domain D02 at 0x20000

Set          0xd50
AcmMap       0x40000 0x40000 0x4 // Step 20: OS maps new page
AcmResume    0x1 // Step 21: OS runs acmSw at 0x10000

Set          0xe00
AcmResume    0x3 // Step 27: OS runs secure domain D03 at 0x30000

Set          0xe50
Load         0x40000 // Step 33: OS tries to read the shared page between D02 and D03

Set          0xf00 // Acm Fault location
Branch       0xf00 // Step 34: self-loop, stays here once acm-fault occurs

Set          0x10000
Schedule     0x10100 // Step 4, 10, 22, acmSw location

Set          0x10110
Branch       0x10200 // Step 5: acmSw creates D02
Branch       0x10250 // Step 11: acmSw creates D03
Branch       0x10300 // Step 23: acmSw to complete new page share

Set          0x10200
AcmSwResp    0x10000002 0x20000 0x20000 0x2 0x2 // Step 6: D02 created, control returns to OS at 0xb00

Set          0x10250
AcmSwResp    0x10000002 0x30000 0x30000 0x3 0x3 // Step 12: D03 created, control returns to OS at 0xb00

Set          0x10300
AcmSwResp    0x10000006 0x40000 0x40000 0x4 // Step 24: page mapping completed by acmSw, return to OS at
0xb00
```

```

Set          0x20000          // Secure Domain D02 code
AcmNewShare 0x40000 0x1729   // Step 17: request for new page share by D02, return to OS at 0xb00

Set          0x30000          // Secure domain D03 code
Schedule    0x30100          // Step 28: Scheduler for secure domain

Set          0x30110
Branch     0x30200          // Step 29: Secure domain code
Branch     0x30234

Set          0x30200
AcmAddShare 0x40000 0x1729   // Step 30: Request shared page access, control returns to OS at 0xb00

Set          0x30234
Load       0x40000          // Tries to read from Secure Domain D02
Branch     0x30240          // self-loop

```

Both code pieces shown in Demo 1 and Demo 2 are written in plain text format. A parser coded in Haskell takes the text file and generates assembly code. This assembly code is then compiled to generate a Hex file. The Hex file is then used to seed the memory model for Verilog simulations. The same Hex file is also used to create a binary file. The binary file is loaded using a backdoor DDR interface into the memory of the processor implemented on the FPGA platform. While the Verilog simulator gives a detailed trace as each instruction is executed, the FPGA system has a separate debug interface, which lets the user read the contents of any register or memory location after execution of one or more instructions. We thus verified the concept of sharing using the simulation and on the hardware platform.

### **Booting Linux on Access Control Monitor Integrated Power core on the FPGA platform**

Eight new instructions have been added to the Power ISA in our prototype and many new service addresses to handle context switch and ACM faults. In order to enhance the adaptability of this new hardware we are moving from coding in assembly language to writing code in any higher level language with the operating system understanding and utilizing the new secure processor. Along this direction we have started booting the Linux kernel on the ACM integrated Power core. After running 200, 000 instructions in the Linux kernel we discovered that the core was hanging. Further debug led to the discovery of a Bluespec-Xilinx design bug in pre-fetch buffers. These buffers were implemented using Block-RAM based FIFOs, the threshold signals of FIFO - full and empty were incorrectly generated thus inserting invalid instructions in the pipeline occasionally. The library element of the Bluespec was changed to generate FIFOs using distributed RAMs with correct threshold detectors. This problem is now fixed. A TCL based interface was also created for the Verilog simulation environment. This interface is identical to the interface being used on the FPGA platform thus identical traces from both FPGA hardware and Verilog simulation can now be generated and compared for hardware debug.

## **4.2.2. MODIFYING THE MODEL OF AN EXISTING POWER PROCESSOR**

**Developing tools for building SVMs:** The initial SVM bootloader consisted of a compressed kernel image wrapped with some bootstrapping code. The compressed kernel image contains a Linux kernel (version 3.17) embedded with the Petitboot application; we called this component Linux-Petitboot. The Petitboot application is a user-space application that loads and executes the VMs kernel image using kexec. After the bootstrapping code loads and decompresses the kernel



image, Linux-Petitboot interacts with the Slimline Open Firmware (SLOF) using a callback pointer to retrieve information about devices available on the system in our case the VM. Due to security concerns, the SVM must enter secure mode (i.e., invoke the `esm ucall`) early in the bootstrapping code before transitioning to Linux-Petitboot. Consequently, this changes the interaction between Linux-Petitboot and SLOF to one that involves data transfer between memory regions with different security labels. Our ACM memory protection is designed to deny such data transfer, and thus this interaction requires special handling, such as wrapping calls to SLOF, to permit read/write from memory regions with different labels.

We addressed this interaction issue by using the flattened device calling convention when entering/calling Linux-Petitboot instead of the Open Firmware callback pointer. We designed and implemented code to build the flattened device tree data structure, set up the pre-boot runtime environment, and invoke Linux-Petitboot using this data structure. Building the flattened device required walking the device tree information maintained by SLOF. In the bootstrapping code, we added code to call SLOF functions to retrieve device information and build the flattened device tree data structure, which consists of nodes having property name/value pairs describing properties of the device. At a high level, the flattened device tree contains four sections: header, memory reserve map, string structure, and device structure. The header includes magic number, and offset and length information of the other sections. The memory reserve map section declares memory ranges that Linux should not allocate. An example is the memory region of the RTAS component. The string section has the list of strings corresponding to property names in the tree. The device section contains the tree nodes representing the devices on the system.

Setting up the pre-boot runtime environment required performing two tasks. The first task included instantiating the RTAS object and updating its RTAS property values specifying memory location and size in the device tree. The second task involved updating the `/chosen` and `/vga` nodes in the device tree with environment information required by Linux. For example, Linux requires the absolute path to the console to be specified in the `/chosen/linux, stdout-path` property.

We instrumented the bootloader with calls (ultracalls) to the ACM. After invoking the `esm ultracall`, which sets up the secure executing environment, the bootloader invokes the memory permission ultracall and then transfers control to the Petitboot kernel by executing its entry point. Changing the memory permission of unprotected memory segments prevents future running code in the SVM from reusing these memory segments (address space) for disclosing or leaking sensitivity information. We also automated the creation of SVMs from normal VMs. We designed and wrote two bash scripts to derive SVMs from VMs defined in the PowerLinux Qemu/KVM environment. The first script creates the SVM xml configuration file and image file based on an existing VM configuration in the system. The SVM image file is a clone of the original VM, except for the PreP partition in the original image file. The SVM image contains a PreP boot, boot and root partition with the appropriate sizes and attributes, such as PreP boot, boot, and lvm. The PreP partition is created with a size of 32,768 KB, which is large enough to hold the secure bootstrap loader which is 17 MB if embedded with a gzip compressed Petitboot Linux kernel image. The first script uses the `virt-clone` Linux command line tool to generate unique values in the SVM configuration file for MAC address and VM uuid. The SVM image filename is created by concatenating the original VM image filename with the string `svm`. The second script writes data into the SVM image disk. It copies the boot and root partitions from the original VM image into the SVM image file. It then writes the secure bootstrap loader into the SVM PreP boot partition.

The second script uses Linux utility tools `losetup` and `kpartx` to loop mount the original VM and SVM image partitions.

The tooling successfully creates an SVM which the prototype ACM firmware successfully boots.

**Modifying the P8 architecture** We modified the P8 architecture as described in our draft ASPLOS paper in Appendix A. This draft paper contains a detailed but high level description of the architecture so we will not reproduce it here.

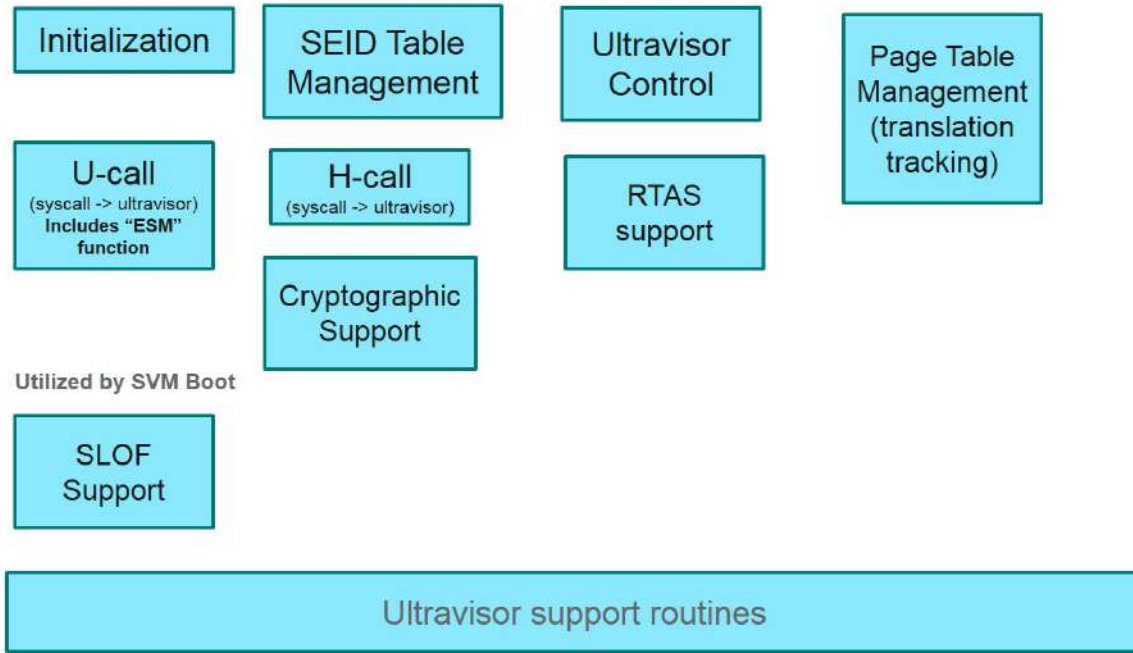
**Design Requirements** for ACM firmware The ACM firmware will operate in the following environment:

- The ACM firmware only receives control as a result of an interrupt or from HostBoot on power up.
  - In our simulated environment (Mambo) the bare bones ACM firmware receives control directly
- The interrupts that the ACM firmware receives can be broadly divided into three classes
  - Ultravisor mode (or ACM firmware) directed interrupts
  - Hypervisor directed interrupts
  - Supervisor directed interrupts
- ACM firmware functions are time limited. No function should take longer than the hypervisor interrupt time.
  - There will be no idle loop in the ACM firmware
  - Some functions cannot be completed in one time quantum. They will have to be implemented so that they can be suspended and resumed.
- The ACM firmware must be thread safe.
  - The ACM firmware operates on the thread of the process that was active when it received control. It must respect all time limitations. Consequently, there will be variable time periods available for its work.

Figure 19 identifies the major functions of the ACM firmware that we have developed for our prototype. At the bottom of the picture is a box titled *Ultravisor support routines* this refers to all of the support functions that are being written to support the main functions of ACM firmware. The main ACM firmware functions in the prototype are defined as follows:

- **Initialization:** This code initializes the ACM firmware
- **Ucall:** Handles ACM firmware calls
- **SLOF Support:** Support for Simline Open Firmware (SLOF)
- **SEID table management:** Management of the SEID table

## Functional overview of ultravisor



**Figure 19 Functional overview of ACM firmware**

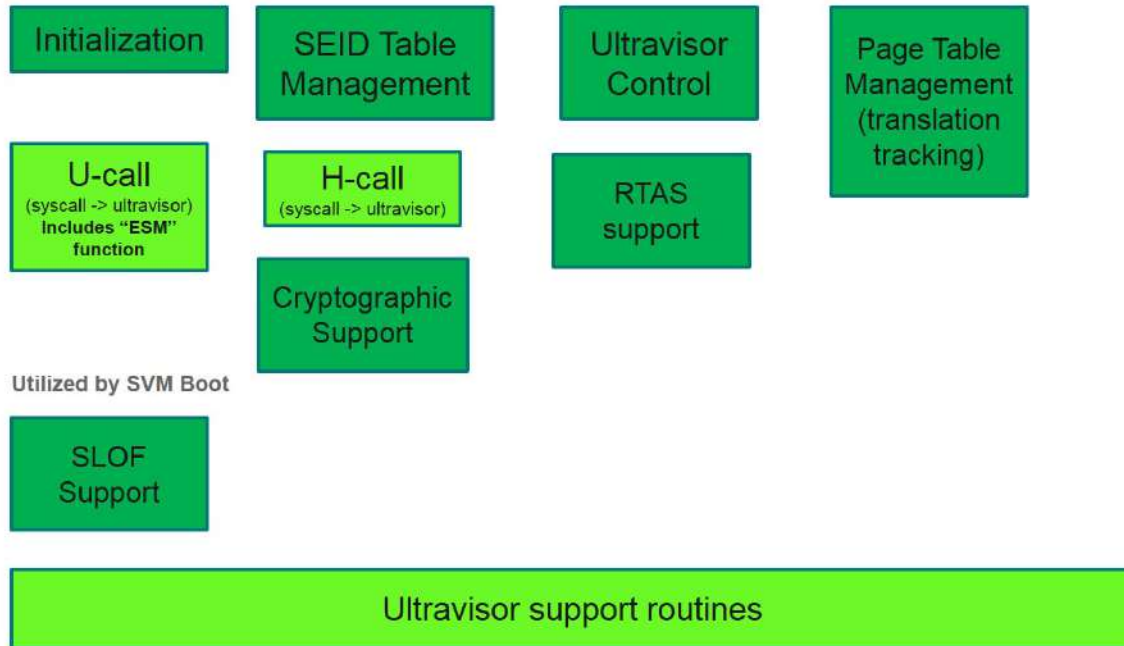
- **H-call Support:** for hypervisor calls (wrappers) made by SVMs.
- **Cryptographic Support:** Cryptographic routines in use by the ACM firmware.
- **Ultravisor Control Interrupt Processing:** Ultravisor control interrupt processing. This interrupt occurs when the systems touches a facility being monitored by the ACM firmware.
- **RTAS:** Real time abstractions services (RTAS) support.
- **Page Table Management:** Page table support. The ACM firmware monitors all page tables in the system
- **Ultravisor support routines:** Support routines required for ACM firmware.

For a full product version of ACM firmware on an IBM Power server additional support might be required such as:

- **Partition Migration and/or Hibernation:** Support to allow migration or hibernation of SVMs.
- **MM I/O Support:** Support for memory mapped I/O.
- **XIV Support:** XIV table support.
- **Window Context Management:** Window context management support.

**ACM firmware** The ACM firmware contains the nine functions identified in Figure 20. Many of the functions of the ACM firmware involve multiple interrupts. During the power on sequence, the ACM firmware takes control of the system and then passes control to the next executing

## Functional overview of **barebones** ultravisor



**Figure 20 Components required for bare bones ACM firmware**

component usually a hypervisor or an OS (for a guest VM). During the initial tests of ACM firmware, the hypervisor and the VMs it ran did not exploit any features of the ACM firmware (i.e., no SVMs were executing). The objective of this test was to demonstrate that the ACM firmware function are transparent to any system that is not exploiting the ACM/SB++ functionality. As previously stated, the ACM hardware does not include any timer that passes control to the ACM firmware. Consequently, the ACM firmware only responds to interrupts. Within these criteria, the full implementation of ACM firmware has to perform all of the functions listed in previously. Whenever Ultravisor mode is active, even when there is no ESM instruction or Ultravisor call, ACM firmware must do the following:

- Handle SEID faults because the SEID table is filled in on demand
- Handle Control calls
- Perform allowable operations on restricted registers.
- Watch SDR1 and RMOR, LPIDR and PIDR (Power architecture registers)
- Manage writes to context windows (SEID fault)
- Manage page tables (Ucalls)
- Monitor XIVE tables (SEID fault)

Our hardware architecture supported the development of type-1 ACM firmware. Consequently, the ACM firmware was developed incrementally. The green boxes in Figure 20 indicate the components we developed to complete our test. The seven dark green boxes have to be fully

implemented, the lighter green boxes are only implemented to the extent required by the four dark green boxes to successfully boot and run an SVM. The components that were fully or partially implemented are:

- **Initialization:** This code initializes the ACM firmware.
- **U-call:** Support for ultravisor calls. A full system will have more than we implemented.
- **SLOF support:** Support for SLOF.
- **SEID table management:** Management of the SEID table.
- **H-call Wrappers** for the H-calls used by the virtual machines we booted. A full system has to support all 118.
- **Cryptographic:** Cryptographic routines and TPM functions used by ACM firmware.
- **Ultravisor Control Interrupt Processing:** Ultravisor control interrupt processing. This interrupt occurs when the systems touches a facility being monitored by the ACM firmware.
- **RTAS:** Support for RTAS.
- **Page Table Management:** Page table support. The ACM firmware monitors all page tables in the system
- **Ultravisor Support Routines:** Common functions that can be shared by multiple ACM firmware components.

**Booting an SVM with Bare Bones ACM firmware** The prototype ACM firmware supporting an SVM contains the components shown inside the green boxes identified in Figure 20. In order to boot an SVM the ACM firmware must decrypt the secure header to obtain a symmetric key, an integrity root, the address range that is cryptographically protected (note that the entire address range that is integrity-protected). The ACM firmware then verifies the integrity of the initial code, assigns to the SVM an SEID (color) and a runtime key and returns to the SVM in secure mode at the start address for secure execution. To support this process we had to develop appropriate "wrappers" to facilitate data transfer to/from protected to unprotected memory space.

We tested our ACM firmware support for SVMs by booting Linux/KVM (as a hypervisor) on our hardware model, booting a virtual machine on top of Linux/KVM hypervisor. We booted and ran an SVM on top of the ACM firmware) (including virtual I/O). By booting Linux/KVM and multiple virtual machine we have demonstrated the feasibility of this approach to security. It is important to note that the simulator we are using is CPU focused and consequently does not contain an architecturally accurate simulation of the I/O subsystem. Linux/KVM boots off a disk, but the disk is provided by the simulator without simulating all of the actual hardware for I/O that exist in our current (or future) systems.

#### 4.2.3. PERFORMANCE OF BARE BONES ACM FIRMWARE

Based on our initial measurements the overhead for having the current ACM firmware manage page tables is 25% - 33%. This measurement was made by counting the number of instructions it takes to boot to the command prompt using the simulator on an unmodified processor and comparing that to the number of instructions it takes to boot to the command prompt on top of the bare-bones ACM firmware. This overhead is significant. The largest component of the overhead appears to be page table management. For every page table update Linux/KVM touches the page tables three time. First to get the lock, next to update the entry, and finally to release the lock. We expect that paravirtualization would reduce this overhead by about one-third.

Kernel same-page merging (KSM) is a facility in the Linux/KVM hypervisor that allows multiple virtual machines to share the same page of the kernel (write only). Our testing showed that this facility adds excessive cryptographic overhead when booting an SVM. Since KSM did not add value to our proof of concept, we disabled it in all SVMs we booted. It should not apply to an SVM because our cryptographic support encrypts each page uniquely so the KVM daemon will never find a match between two SVMs.

#### **4.3. CUSTOMER NEEDS ADDRESSED**

Our high-level objective is to build systems that can protect sensitive software and data from other software, including systems software and other applications, as well as rogue administrators. This objective is even more timely given that the frequency and impact of breaches and the consequent loss of sensitive data is increasing, despite increased investments in cybersecurity. It can be observed that it is very hard to verify the provenance, correctness and malware-free operation of all software components like hypervisors, operating systems, privileged software, etc. Security concerns are amplified by the increased popularity of multitenant and cloud computing models, which introduce multiple owners and system software components. Therefore, our key thesis is that building systems to protect sensitive software and data requires hardware enabled security. To facilitate achieving this objective, we need to minimize the hardware and software that needs to be trusted — also known as the Trusted Computing Base (TCB).

#### **4.4. COMPARISON WITH COMPETITION**

The most relevant related work is Intel Software Guard Extensions (SGX), which was announced after the start of this project. SGX is motivated by similar requirements for protecting the integrity and confidentiality of sensitive code and data. It employs a similar approach for creating “enclaves” protected by processor extensions. We note that Intel SGX publications refer to one of our IBM Research papers on SecureBlue++.

In terms of comparison, SGX appears to be more focused on protecting sensitive parts of an application - what we refer to as “fine grain” protection. This approach requires changes to application source code. Our initial focus is on “coarse grain”, end-to-end protection for entire Virtual Machines (or containers) — protection that is more applicable to cloud workloads and goes beyond what is currently possible with SGX. By end-to-end protection of VMs we mean that our approach safeguards SVMs across their life-cycle and, for example, allows for building, distributing and running SVMs with built-in secrets. Our approach also aims to be largely transparent to existing software — i.e. not requiring changes to application source code.

#### **4.5. TECHNOLOGY TRANSITION AND TRANSFER**

During this project we engaged with IBM product partners in the Systems Group, which is now the successor to the previously named Systems and Technology Group (STG), in order to pursue the commercialization of a subset or a derivative of the proposed architecture developed under this project. Our technical discussions with the IBM Systems Group have focused on secure processor architecture, design and prototyping and span both hardware, targeted for future generation processors, and modifications to the associated software stacks for firmware

(including the ACM firmware), hypervisor, guest OS and applications that take advantage of the hardware/processor enhancements.

As part of our technology transfer activities we developed the functionality of the simulators of our processor core extensions that are aligned with existing commercial POWER processor models, along with prototypes of the ACM firmware. We also generated a detailed analysis of the hardware and software overhead of our architecture on a future generation Power processor in order to provide a more accurate estimate of all the development requirements for our IBM commercialization partners.

A derivative of the ACM hardware extensions and associated ACM firmware developed under this project are planned for release in a next generation Power processor. The planned feature was referred to as “trusted execution enforced by hardware” in a recent presentation by IBM, titled “POWER9: Processor for the Cognitive Era”, at the Hot Chips Conference [7]. The relevant slides from this presentation are extracted and included in Appendix A.

## 5. CONCLUSION

End-to-end security and trust is a critical part for sensitive workloads that may be deployed across a combination of cloud, enterprise servers, mobile or IoT platforms. At the start of this project there was no proof that VMs could be isolated in the way that was proposed. The prototypes constructed during this project proved that hardware extensions, along with a type-1 ACM firmware can be developed in a way that is transparent to unmodified VMs while supporting SVMs. The “hardware enforced trusted execution” announced for POWER9 [7] is a derivative of the architecture and benefits from lessons learned from this project. If container technology evolves to exploit virtualization hardware, as is projected by some, containers will also be able to exploit this architecture and its derivatives. Further research can and will be done in this area; most likely driven by customer requirements and the evolution of container technologies and offerings.

For low-end and mobile clients we demonstrated that it is possible to build secure devices. This is an area that TCG is also focused on. We used a TPM chip to illustrate the concept for sensors and actuators, without employing the function of the TPM. TCG is developing a specification for RTM that will help IoT developers integrate trusted computing into their designs. We contributed to that standardization work as part of this project. For mobile platforms we developed and prototyped an architecture that supports separation between different personalities on the same platform, safeguarding enterprise from personal data and applications in a bi-directional manner. The technologies that were developed verified the integrity and trustworthiness of different virtual machines, which correspond to different personalities

Combining the different parts of our work produces a novel, realistic and feasible architecture for malware defense and end-to-end trust.



## 6. REFERENCES

- 1 swTPM, Ken Goldman and Stefan Berger, IBM <http://ibmswtpm.sourceforge.net>
- 2 IMA and EVM, Mimi Zohar, IBM <http://linux-ima.sourceforge.net>
- 3 OpenAttestation (OAT), Intel <https://github.com/OpenAttestation/OpenAttestation>
- 4 QEMU patches, Stefan Berger <http://lists.nongnu.org/archive/html/qemu-devel/2011-07/msg00525.html>
- 5 "KVM Virtualization on the Arm Chromebook", Alexander Spyridakis, and Nikolay Nikolaev, Virtual Open Systems, <http://www.virtualopensystems.com/media/chromebook/chromebook.pdf>
- 6 "A Virtualized Linux Integrity Subsystem for Trusted Cloud Computing", Stefan Berger, Kenneth Goldman, Dimitrios Pendarakis, David Sord and Mimi Zohar, The 2nd NSA Trusted Computing Conference and Exposition, September 20-22, 2011, Orlando, FL
- 7 "POWER9: Processor for the Cognitive Era", Brian Thompson, Hot Chips: A Symposium on High Performance Chips, August 21-23, 2016, Cupertino, CA. [http://www.hotchips.org/  
http://www.hotchips.org/wp-content/uploads/hc\\_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-national.pdf](http://www.hotchips.org/http://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-national.pdf)

## **APPENDIX A: PUBLICATIONS**

“Embedded Linux Integrity”, by David Sa\_ord, presented at Linux Security Summit (LSS)2013.

# Embedded Linux Integrity

David Safford

6/30/2013

## Abstract

*Linux is in widespread use in embedded devices, but these devices typically lack critical security features found in higher-end Linux systems. They typically do not have any way to validate their firmware, they do not have hardware roots of trust for trusted or secure boot, they do not have provisions for physical presence, to protect firmware from remote modification, and they do not have secure update. Vendors claim that these features are either too large, or too expensive to fit in their embedded devices.*

*This paper summarizes the recent widespread vulnerabilities and compromises of embedded devices, and shows how the given security features would defeat such attacks. It relates the concepts to the NIST SP800 guidelines for BIOS measurement and protection, and to the ongoing work on Linux secure boot for higher end devices. It looks at four typical embedded devices, shows how all of these features can be added at zero cost.*

## Introduction

Linux runs on an incredible range of devices from very small embedded devices, to the largest supercomputers. The devices cover a staggering 12 orders of magnitude in memory size, and 7 orders of magnitude in cost.

Embedded Linux devices typically consist of just three small chips – an SoC, flash, and RAM. The SoC (System on a Chip) normally includes a 32

bit ARM or MIPS CPU, along with flash, RAM, USB, ethernet and wireless interfaces. The flash is typically a 4 or 8MB SPI device, and the RAM is usually 32 – 64 MB. The firmware on these small devices includes a Linux kernel, stripped and compressed to under 1MB, and a squashed root filesystem under 3 MB. There is no initramfs.

For this class of embedded devices we are mainly interested in four foundational integrity features:

- initial firmware validation
- run-time firmware protection
- firmware update validation
- boot time integrity validation

While higher-end Linux devices in the mobile, PC, and server categories have one or more of these features, typical embedded devices have *none* of them. Table 1 shows some categories of Linux devices, and their typical integrity features.

Category	Cost	Size	Typical Integrity Features
Server	\$10K+	PB	4768 Crypto card Trusted and Secure Boot
PC	\$1K	TB	TPM Trusted and Secure Boot (Win8)
mobile	\$500	GB	Restricted Boot
embedded	\$50	MB	Nothing
Sensor	\$10	KB	Nothing

Table 1: Linux Spectrum

This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Homeland Security Advanced Research Projects Agency, Cyber Security Division (DHS S&T/HSARPA/CSD), BAA 11-02 and Air Force Research Laboratory, Information Directorate under agreement number FA8750-12-2-0243. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Department of Homeland Security, Air Force Research Laboratory or the U.S. Government.

For this paper, we selected four example embedded linux devices as representative: Linksys WRT54G, TP-Link MR3020, D-Link DIR-505, and the Pogoplug model 2.

The WRT54g was the first wireless access point to run embedded Linux. Linksys published all the GPL source code to this device, and today virtually all wireless devices use a derivative of this original embedded Linux. Openwrt is a community supported derivative that runs on most past and current wireless devices.

The TP-link and D-link are small, travel-sized versions with significant additional features, including support for USB attached network storage and media serving. Figure 1 shows images for these four selected devices.



Figure 1: Example Devices

Embedded Linux devices like these four have been extensively compromised recently. In 2012 4.5 Million home routers were compromised in Brazil [1]. In this one attack, many different Broadcom based devices, from multiple vendors and across four ISPs were compromised, redirecting all home client devices to malicious DNS servers. While the vulnerability was present only on the internal networks, the basic exploit was so simple that using CSRF (cross site request forgery) to “bounce” the attack off local

browsers was quite effective. The basic (trivial) exploit to get a plaintext copy of the admin password was:

```
get.pl http://192.168.1.1/password.cgi
```

In 2013, the D-Link DIR-645 home router was similarly found to give away the admin password in a trivial exploit [2]:

```
curl -d SERVICES=DEVICE.ACCOUNT
http://<device ip>/getcfg.php
```

In 2013, five new vulnerabilities in Linksys routers were discovered[3]. The WRT54GL (one of the devices selected), was found to have a CSRF vulnerability allowing unauthenticated upload of arbitrary firmware. The EA2700 was found to have a CSRF file path traversal vulnerability which can expose all files (including /etc/passwd):

```
POST /apply.cgi
submit_button=Wireless_Basic&change_acti
on=gozilla.cgi&next_page=/etc/passwd
```

It also had an interesting CSRF vulnerability returning the source code of any page. (This is one of the few known single character exploits, as adding trailing / is all that is needed.) For example:

```
http://192.168.1.1/Management.asp/
```

In 2013 researchers hacked over a dozen home routers [4] with two remote (CSRF) root exploits on the Belkin N300, and Belkin N900, and four local (WLAN) root exploits on the same Belkin devices, plus the Netgear WNDR4700.

The one-line root exploits were (N300):

```
<form name="belkinN300"
action="http://192.168.2.1/apply.cgi"
method="post"/>
```

N900 exploit:

```
<form name="belkinN900"
action="http://192.168.2.1/util_system.h
tml"
```

With so many of these devices vulnerable to such simple exploits, their integrity is at significant risk. While all of these vulnerabilities are in the web management interfaces of the devices, integrity measurement and protection mechanisms at the device level can detect and prevent successful exploits of all of these vulnerabilities.

### Threat Model and Integrity Goals

The threat model we consider includes supply chain attacks, and remote software attack. Can the attacker compromise the integrity of the device's firmware – either before purchase, or over the internet once it is installed? We are not concerned with local physical attack, as this does not scale, and is quite difficult to protect against. In fact, physical presence will explicitly be trusted as part of the proposed defenses.

So what integrity features are possible and appropriate for embedded devices? One starting point is to look at the NIST guidelines for BIOS Integrity Measurement and Protection. While these guidelines were intended for PC and server class machines, they are a good starting point. Currently there are two specific guidelines: NIST-SP800-155-December-2011 BIOS Integrity Measurement (Draft) [5], and NIST-SP800-147-April-2011 BIOS Integrity Protection [6].

The first presents guidelines for “trusted boot” - the incorporation of a hardware chip as a root of trust for measuring and attesting to the integrity of the BIOS itself, and of subsequent software as it is booted and executed. Note that this guideline does not discuss “secure boot”, the much more common hardware root of trust which validates signatures on software before booting.

The second guideline addresses integrity protection for the BIOS - the requirements that BIOS integrity should be protected from remote software attack, that any updates need to be either authenticated, or done in some physically protected local manner, and that the mechanisms for protection must not be by-passable.

From these guidelines we can derive four related integrity features desirable in the embedded device category. For this class of embedded devices we are mainly interested in four foundational integrity features (the NIST guideline terminology is in parenthesis):

- initial firmware validation (“BIOS measurement”)
- run-time firmware protection (“BIOS protection”)
- firmware update validation (“Secure/local updates”)
- boot time integrity verification (“Secure boot”);

Table 2 shows the four sample embedded devices, and that most of these requirements are not met.

Device	Measure BIOS?	Lock BIOS?	Secure-local updates?	Secure Boot?
Pogoplug	Yes - SATA	No	No	No
D-Link DIR-505	No	No	No	No
TP-Link MR3020	No	No	No	No
Linksys WRT54G	Yes - JTAG	No	No	No

Table 2: Initial Integrity Features

The first exception was that the WRT54GL does have a JTAG interface through which a user can read (and write) the firmware in the flash chip. Normally JTAG interfaces and corresponding software are quite expensive and complex, but the WRT community has articles showing how to do this with free software and roughly \$10 for a parallel port connecting cable.

The second exception is that the pogoplug is based on a SoC with the built-in ROM code to boot from a SATA disk drive, and articles show

how to boot firmware inspection tools securely from a trusted SATA disk image.

What can be done to cover all of the desired functions in all of the example devices? One problem is that vendors say that these features are simply too large (they won't fit in flash/RAM), or are too expensive (adding a \$0.75 TPM chip is simply not feasible). So in the remainder of this paper we show how all of the features can be added at zero cost and no additional storage space.

### Initial Firmware Validation

How do we verify that a BIOS is authentic? We can't just ask it while it is running, because it will lie if it is malicious. We already mentioned the two methods used by the Pogoplug and the WRT54: JTAG and trusted immutable boot ROM.

Another method similar to JTAG is to use the flash's SPI (Serial Peripheral Interface) to read the contents directly. Most embedded Linux devices use SPI flash for the firmware, and the D-Link and TP-Link devices both do. The SPI bus was designed to be sharable if it is properly buffered, and many PC motherboards feature buffered SPI interfaces for their BIOS for ease of modification (and the subsequent un-bricking).

Unfortunately the D-link and TP-Link do not properly buffer the SPI bus between the SoC and the flash, so any attempt to attach a hardware reader to the bus results in contention, and neither the SoC nor the reader function correctly. So at first this appeared to be an unworkable solution.

While there are many other theoretical ways to read the flash contents, such as real-time passive monitoring and reconstruction of the SPI data, or even power or RF monitoring, these methods are quite complex and expensive to implement. The only other known way to validate the flash's contents is to unsolder the flash chip from the mother board, so it can be read by the SPI reader without interference. Having actually done this,

we concluded this was not an acceptable method for routine validation.

Going back to the SPI bus reading method, we looked for the simplest, low cost way that an owner (or even better, the vendor) could properly buffer the SPI bus. Both the D-Link and TP-Link devices are based on the same Atheros SoC, so any solution would work on both devices. Our preferred method for reading/programming SPI flash devices is the Buspirate [9] shown in figure 2, combined with the open source flashrom software application [10].

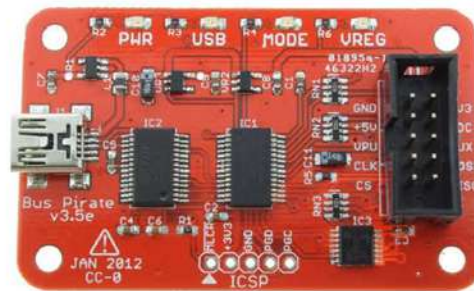


Figure 2: Buspirate SPI programmer

The buspirate is a \$30 device with a USB interface for power and control from a PC running flashrom. It in turn has a 10 pin header with power supply outputs for optionally powering the chip, and input/output pins for reading/writing. While there are cheaper parallel port cables for SPI programming, bit-banging the SPI lines through a parallel port is much slower than using the buspirate, which has circuits for generating the needed serial clock and data signals directly.

Using the buspirate and flashrom, we experimentally determined that the SPI bus could be sufficiently buffered for in circuit programming with just three additional resistors as shown schematically in Figure 3. Figure 4 shows the buspirate connected to the MR-3020 while reading the flash contents, and figure 5 shows a more detailed view of the (crudely) added buffering resistors. The D-Link and TP-Link circuit boards already have a large number of resistors; adding three more would cost less

than 1 cent in quantity, so we think this qualifies as a zero cost modification (and is much more convenient than chip unsoldering).

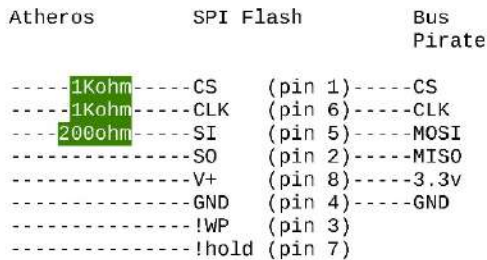


Figure 3: SPI buffering Schematic

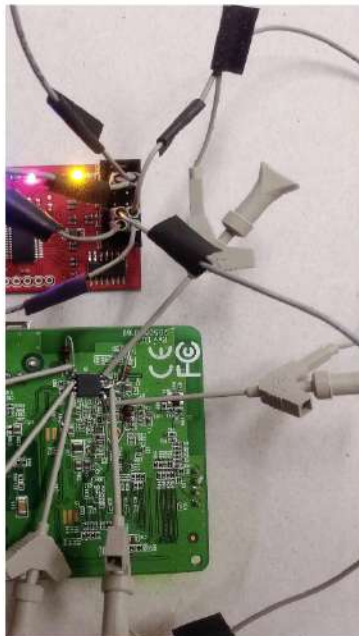


Figure 4: SPI in-circuit-programming

### Run-time Firmware Protection and Firmware Update Protection

The second main integrity goal is to protect the integrity of the firmware from remote software attack. While this could easily be done by permanently wiring the flash's !WP pin low, this would prevent valid firmware updates and re-

configurations. So the closely related third goal is to ensure that updates to the firmware are allowed, but validated or locally authorized in some way that is not by-passable by remote software attack.

The flash chips in all four of the sample embedded devices have a Hardware Protection Mode (HPM) which can block all writes to all or selected parts of the flash based on a combination of forcing the !WP pin low, and then setting a control register bit appropriately. The HPM control bit is non-volatile, and survives power cycles, so the only way to exit HPM mode is to force !WP high, and then reset the HPM control bit.

HPM leads to a very simple method for providing both flash locking, and secure local update. If the !WP pin on the flash is normally held low, with a physical momentary push button that can force the pin high, then the firmware bootstrap (u-boot [12]) can simply set the HPM mode bit, disabling all writes to the chip, locking the chip against any updates, including remote software attacks. Then, if an update or reconfiguration is desired, unlocking HPM mode can be done only if someone presses the button, establishing physical presence, for a secure local update.

The MR-3020 uses a Spansion S25FL032A [11] flash chip. This chip has HPM support, with the addition of 4 control register bits, which can select a subset of the chip's address space to protect. Table 3 shows the MR-3020's memory layout. For this prototype, the entire chip was locked, requiring physical presence for any update or configuration of the device, but by using the control bits, a subset could be protected to trade-off security and convenience.

The MR-3020 also has a convenient "WPS" momentary push button, normally used to begin Wireless Protected Setup for establishing a secure connection with a new client. This button has a default high output, which is forced low while the button is pushed, which is perfect for controlling the !WP pin. The button can be used for both functions, as the software context can understand

which function is being requested, although the normal WPS function should be disabled anyway, due to its security weaknesses.

Figures 5 and 6 show the MR-3020 with both the one wire !WP modification, along with the three resistors for SPI buffering. These are the only hardware modifications needed for this device.

Part.	Name	Size	Contents
mtd0	"boot"	64KB	u-boot
mtd1	"kernel"	1024KB	Linux Kernel
mtd2	"rootfs"	2816KB	Linux root filesystem
mtd3	"config"	64KB	config data
mtd4	"ART"	64KB	radio config data

Table 3: MR-3020 Flash Layout

U-boot was modified to demonstrate/test the HPM with physical presence control. At boot time, u-boot attempts to lock the chip, in case it was unlocked before, and then attempts to unlock it. If the WPS button is pressed, then the unlock will succeed, and updates can be applied. If the WPS button is not pressed, then the unlock will fail, and updates will not be possible. In either case, before booting the Linux kernel, u-boot will re-lock the flash.



Figure 5: modified MR-3020 Bottom View



Figure 6: modified MR-3020 Top View

The following u-boot console log shows debugging output with the button pressed and not pressed. A status register value of 2 is unlocked, and 9c or 9e is locked:

```
Write_protect: starting SR = 2
Write_protect: ending SR = 9c
Write_unprotect: starting SR = 9e
Write_unprotect: ending SR = 9e
Write_unprotect failed.
...
Write_protect: starting SR = 2
Write_protect: ending SR = 9c
Write_unprotect: starting SR = 9e
Write_unprotect: ending SR = 2
Write_unprotect succeeded.
```

#### Integrity Protection on DIR-505

The DIR-505 has a similar WPS momentary push button for !WP control. It also has an interesting sliding switch for controlling the operating mode of the device, to choose between Router, Repeater, or Hotspot modes. This sliding switch actually is a four position switch, with the fourth position unused. The fourth position can be accessed simply by trimming the plastic slide a bit, so that the fourth position can be reached, and used to force the !WP pin high to allow



updates/reconfigurations. Figure 7 shows the modified DIR-505 slide switch.



Figure 7: modified DIR-505

### Boot-time Integrity Verification

The fourth integrity goal is to validate the firmware at boot time (“secure boot”). Assuming that the u-boot partition is locked with HPM as discussed, then it can be a trusted root to validate the linux kernel before loading and booting it. If the kernel is signed with a private key, and the corresponding public key is stored in the protected u-boot partition, then u-boot can do the validation, and the key can be changed only with physical presence.

This secure boot, with physical presence controlled key management was implemented on the MR-3020. The MR-3020 was chosen as it provided the greatest challenge. With the smallest flash chip (4MB), its entire u-boot partition is only 64K bytes, and the existing u-boot code used 54K, leaving just 10K bytes to implement all of the needed functions (HPM flash locking with physical presence control, RSA signature validation, and public key storage and management.

The RSA signature verification code was derived from the PolarSSL library [13] by stripping out everything not needed. The kernel signature was created with standard openssl commands, and the resultant binary signature simply appended to the end of the kernel. The (single) validating public key was stored in binary form at the end of the u-boot partition. The combined flash locking and signature verification code added roughly 8K bytes to u-boot, increasing its total size to 62K, which with the public key still fits within the 64K partition.

The following u-boot console debugging output shows hex formatted output of the sha1 hash of the kernel, the public key modulus, the binary PKCS1.5 signature, and the results of the verification.

```
## Booting image at 9f020000 ...
kernel sha1
E9321D87C091F971C8D955C399EBA53807429A61
modulus:
9292758453063D803DD603D5E777D7888ED1D5BF
35786190FA2F23EBC0848AEA
DDA92CA6C3D80B32C4D109BE0F36D6AE7130B9CE
D7ACDF54CFC7555AC14EEBAB
93A89813FBF3C4F8066D2D800F7C38A81AE31942
917403FF4946B0A83D3D3E05
EE57C6F5F5606FB5D4BC6CD34EE0801A5E94BB77
B07507233A0BC7BAC8F90F79
signature:
2CB0F653FF3BCCFF2E31ACC0840F02A84975B716
7291BB36EEE3F74D02EB3B6A
ACADE02CBCF6E2326230C296E4D8A8D70F309479
B388A99591AD5C41938280E3
F51EA9865ED8A0360A0F5BD6A6C676C363B43E54
61D9CCF00D46E1B5449CB262
BDE36CAD4AFBEE51ED731BBF48340F290DF8DD84
4791D81259CEDF99CD1CA2E6
rsa verify kernel succeeded
Uncompressing Kernel Image ... OK
```

### Summary

Table 4 shows the final results of these modifications on the D-Link and TP-Link devices, and similar modifications on the Pogoplug and Linksys devices. With essentially zero cost hardware and software modifications we can meet all four integrity goals on all four example devices. With firmware measurement, we can detect supply chain or other firmware

modification. With HPM locking, we can protect the firmware from remote modification, even if the remote attacker gets the root password as in all of the earlier described web management vulnerabilities. As physical presence is needed to unlock the flash, we provide secure local update. If the kernel partition is not locked for convenience, secure boot can provide strong validation, with secure local update of the validating public key.

Device	Measure BIOS?	Lock BIOS?	Signed-local updates?	Secure Boot?
Pogoplug	Yes - SATA	Yes	Yes	Yes
D-Link DIR-505	Yes Buspirate	Yes	Yes	Yes
TP-Link MR3020	Yes Buspirate	Yes	Yes	Yes
Linksys WRT54G	Yes - JTAG	Yes	Yes	Yes

Table 4: Integrity features after modification.

## References

[1] Fabio Assolini, “The tale of one thousand and one DSL modems”, Securelist, 2012, [https://www.securelist.com/en/blog/208193852/The\\_tale\\_of\\_one\\_thousand\\_and\\_one\\_DSL\\_modems](https://www.securelist.com/en/blog/208193852/The_tale_of_one_thousand_and_one_DSL_modems)

[2] roberto@greyhats.it, Bugtraq Mailing List 27 February 2013 <http://archives.neohapsis.com/archives/bugtraq/2013-02/0151.html>

[3] Phil Purviance, “Don't Use Linksys Routers”, March 2013 <https://superevr.com/blog/2013/dont-use-linksys-routers/>

[4] ISE, “Exploiting SOHO Routers”, April 2013 [http://securityevaluators.com/content/case-studies/routers/soho\\_router\\_hacks.jsp](http://securityevaluators.com/content/case-studies/routers/soho_router_hacks.jsp)

[5] NIST, BIOS Integrity Measurement

Guidelines (Draft) SP 800-155, December 2011 [http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155\\_Dec2011.pdf](http://csrc.nist.gov/publications/drafts/800-155/draft-SP800-155_Dec2011.pdf)

[6] NIST, BIOS Protection Guidelines NIST-SP800-147, April 2011 <http://csrc.nist.gov/publications/nistpubs/800-147/NIST-SP800-147-April2011.pdf>

[7] Trusted Computing Group, “Trusted Boot” [http://www.trustedcomputinggroup.org/resources/trusted\\_boot/](http://www.trustedcomputinggroup.org/resources/trusted_boot/)

[8] UEFI, Secure boot UEFI specification 2.3.1, section 1.8.1 [http://www.uefi.org/specs/download/2\\_3\\_1\\_D.zip](http://www.uefi.org/specs/download/2_3_1_D.zip)

[9] Buspirate [http://dangerousprototypes.com/docs/Bus\\_Pirate](http://dangerousprototypes.com/docs/Bus_Pirate)

[10] Flashrom <http://flashrom.org>

[11] Spansion, S25FL032A reference manual [http://www.spansion.com/Support/Datasheets/S25FL032A\\_00.pdf](http://www.spansion.com/Support/Datasheets/S25FL032A_00.pdf)

[12] u-boot <http://www.denx.de/wiki/U-Boot/WebHome>

[13] Polar SSL library <https://polarssl.org/>

Invention, titled ``Architectural Prevention of Return-Oriented Programming'', published on August 4th, 2015 on [ip.com](http://ip.com), under the following link:  
<http://priorart.ip.com/IPCOM/000242687D>

Inventors are: [Guerny Hunt](#), Eric Hall, Rick [Boivie](#), Peter [Sandon](#), Dave [Safford](#), Jonathan D Bradbury, [Dimitrios Pendarakis](#), [Mohit Kapur](#), Ray Valdez.

Return-Oriented Programming (ROP) is an important attack technique that bypasses data execution prevention by utilizing fragments of code that are part of the underlying system. ROP exploits re-usable segments of code that end in a return-from-function in either the operating system (OS) or the application under attack. Return oriented programming works against Complex Instruction Set Computing (CISC) and Reduced Instruction Set Computing (RISC) systems with stack architecture, regardless of whether the stack is supported by hardware.

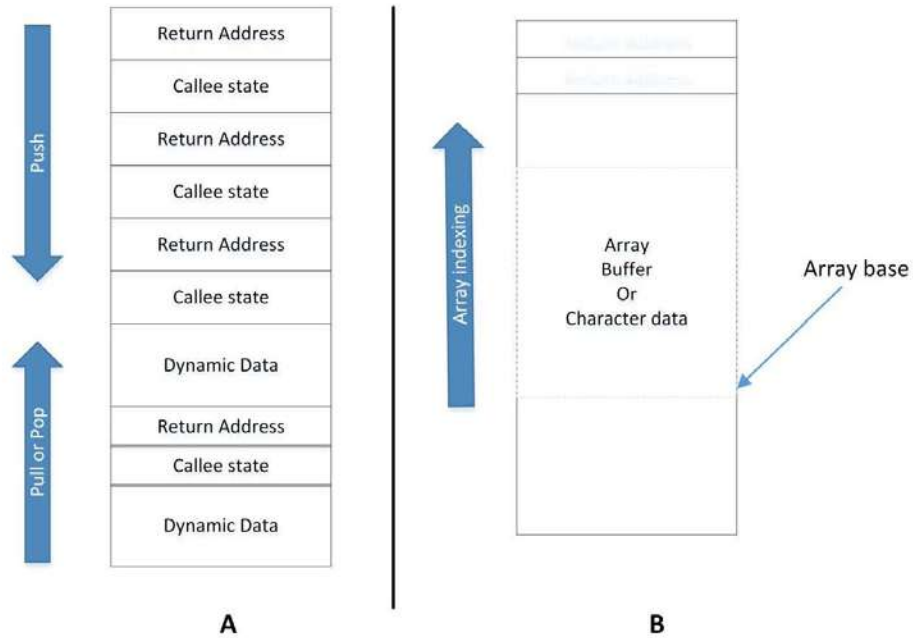
A separate linkage stack that is reserved for use by the operating system does not provide protection to application code. An attacker using ROP does not have direct access to the machine state. The attacker is interfacing to an application over the network through an Application Programming Interface (API), or is interfacing to the underlying OS or Hypervisor through the syscall or hcall interface. By supplying the chosen data to some legitimate interface, the attacker can cause the object under attack to write arbitrary data supplied by the attacker onto the stack. The data supplied by the attacker is the ROP.

In current hardware or software stack base architectures work, when one program calls another, the return address and some system states are pushed onto the stack. (Figure 1A). The order of pushing state onto the stack is not material. In some computer architectures, the return address is pushed onto the stack and the callee has to put the state of the caller onto the stack so that it can be restored. Upon return, the callee pops the state off the stack (or restores the callers state), and then executes a return instruction, which branches to the address on the top of the stack and pops that address off the stack.

This type of mechanism can be made arbitrarily complex. For security reasons, the responsibility for saving the caller's state can be moved from the callee to the caller and can be moved from software to hardware\*. Another ROP enabling feature is the allocation of dynamic variables on the stack. (Figure 1A) If there is a buffer overflow vulnerability or a similar program bug in an application or system, then an attacker can write chosen data onto the stack. The attacker can therefore overwrite the caller's return address with another address. (Figure 1B.)

Many modern processor architectures added Data Execution Protection (DEP), which prevented the execution of code from the stack (or other restricted data areas). The creators of ROP treated all of the code in the system as data and observed that many sequences of code ended in a return instruction. These sequences of code are referred to as *gadgets*. Code gadgets has been analyzed and found to be Turing complete. The attackers then shifted from writing executable code to utilizing tools that created useful attack functions from the gadgets available in a normal system. The attack occurs by exploiting a buffer overflow (or similar vulnerability) to write a sequence of return instruction onto the stack that corresponds to the desired function.

Figure 1: (A) Data typically found on a stack. This data can include arguments to target programs. (B) Illustrates why buffer overflows overwrite return addresses.

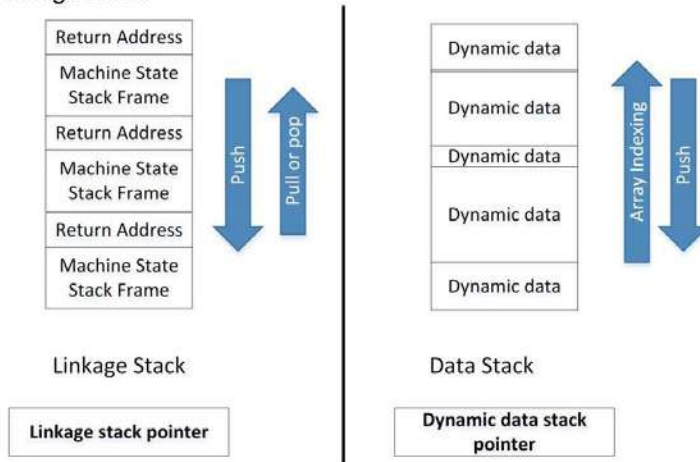


A method is needed to either prevent entering code at arbitrary points or prevent buffer overflows (and similar bugs) from enabling changes to the control flow of an application/system.

The first solution prevents control flow perversion by separating the stack containing return values from the stack containing dynamic data, making ROP no longer possible. This solution works because even though the attacker may find a buffer overflow vulnerability, the only consequence of the vulnerability is to overwrite data that belongs to some other routine on the call stack. This solution is described as a dual-stack architecture. (Figure 2)

On the first stack, called the *linkage stack*, the system places linkage information, and the second stack, called the *data stack*, is used for dynamic data (e.g., arguments that passed to functions and automatic variables instantiated on the stack). Buffer overflows can only occur on the second stack. This is true whether the stacks are implemented by hardware or software. Hardware DEP can be implemented for one or both stacks to increase protection. If a machine has hardware state save and linkage instructions (or function call and return instructions), then the hardware implementation of the linkage stack can prevent any instructions except linkage and state save instructions (or call and return instructions) from writing to the linkage stack.

Figure 2: The first approach separating the linkage/control stack from the data stack. In this approach, overwriting a buffer can never affect the linkage machine state on the linkage stack.

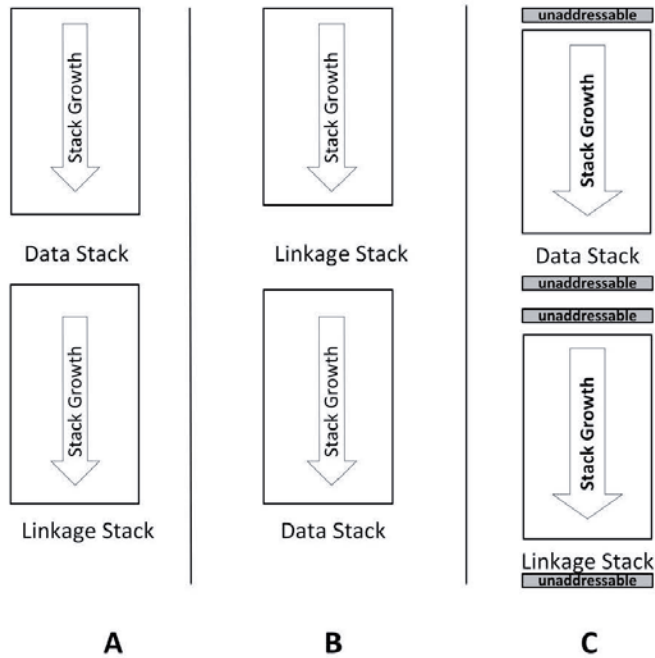


The minimal information required on the linkage stack is the return addresses. An obvious extension of this concept is to place all machine states on a separate stack from dynamic data to prevent an attacker from modifying any machine state of a prior computation.

Simply having two stacks is insufficient to prevent ROP if the stacks are incorrectly placed, as illustrated in figure 3A. To allow arbitrary placement, a mechanism is needed to stop an overflow from one stack progressing over the second stack. Options include:

- Incorporating segments (an architectural change that places the control stack in a separate memory from the dynamic stack. Only the linkage instructions and machine state save/restore instructions can be enabled to write to the control stack.).
- Placing the stacks properly with respect to associated growth (Figure 3B).
- Unmapping a small range of addresses to a bound stack in order to force an exception if a program writes beyond the limit. (Figure 3C) Any other hardware mechanism, such as base and bound registers for the stacks with automatic checking, is also sufficient.

Figure 3: Options for multiple stack pointers. (A) Incorrect placement of two stacks; ROP is still possible. (B) Correct placement of two stacks; ROP is not possible. In (C), the top and bottom of each stack is bracketed to eliminate the possibility of ROP.



ROP is enabled in part because stacks and arrays are indexed in the opposite order (figure 1A & 1B). The system pushes items onto a stack by decrementing and removes items from the stack by incrementing the stack pointer, respectively. When the elements of an array are referenced, the addresses of the elements are computed by adding (or incrementing) to the base address of the array. Consequently, when elements are written beyond the end of a dynamically allocated array, those elements overwrite other data that is on the stack. The data overwritten might contain return addresses. If, on the other hand, elements are added to the stack by incrementing and elements are removed from the stack by decrementing, then overwriting the end of an array writes into a stack area that has not yet been used. In this case, if the call stack goes deeper than the offending routine, then the elements that were written by the offending routine will/may be overwritten by stack frames or dynamic data from a routine deeper in the calling tree.

Simply changing the way the stack is indexed opens the possibility that someone could find an exposure in an API that causes a program to write in the reverse direction for dynamic variables. Therefore, changing array and stack indexing only affects the attack in its current form and not the fundamental issue, dynamic data intertwined with control data exposing the control data to manipulation via dynamic data. Protections similar to the two stack protections could/should also be employed.

A second aspect of the solution is a method for preventing execution through a linkage instruction such as a jump or return at arbitrary points in code. For any architecture with fixed length instructions, arbitrary assemblage of gadgets can be prevented by adding additional information into memory. Assume the architecture has instructions that are thirty-two bits long. A thirty-third bit can be added. The extra bit informs the instruction execution unit whether the associated instruction can be the target of a linkage instruction. The approach modifies the architecture so that any attempt to pass control to an instruction that is not flagged causes an exception. For arbitrary length instruction architectures, a similar approach can be taken. An extra bit can be added to the minimum width instruction. For example, if the minimum instruction is one byte wide, a ninth bit can be added to each byte. If the minimum instruction is two bytes, then the bit is added for every two bytes, etc. For both types of architectures, the compiler and assemblers have to be modified to add the information into the binary object so that memory is appropriately marked.

In an architecture in which call and return instructions automatically manipulate the stack, another approach to solve this problem is tag bits. Whenever a call instruction is executed, all of the data that it pushes onto the stack (register and return address) is marked with the tag bit set. The return instruction, when popping the data off the stack, clears the tag bit for the data it popped off the stack. Any store to a double word with the tag bit on, raises an integrity violation exception. As long as the only instructions that can write the tag bit are the call and return instructions, the linkage information and dynamic data may be combined and there is no risk from buffer overflows. In fact, any overflow from a data area into the linkage stack area will be detected. (An additional privileged instruction would/may be needed to clear out the tag bit in the case where a program abnormally terminates.)

These approaches can be combined, and are not mutually exclusive. ROP can be prevented with a change to the architecture of the underlying system. The difficulty of implementing the change depends on how the architecture is implemented.

\*US patent US8850557B2

[Note: This material is based on research sponsored by the Department of Homeland Security (OHS) Science and Technology Directorate, Cyber Security Division (OHS S&T/CSD) via BAA 11-02; the Department of National Defence of Canada, Defence Research and Development Canada (DRDC); and Air Force Research Laboratory Information Directorate via contract number FA8750-12-C-0243. The U.S. Government and the Department of National Defence of Canada, Defence Research and Development Canada (DRDC) are authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security; Air Force Research Laboratory; the U.S. Government; or the Department of National Defence of Canada, Defence Research and Development Canada (DRDC).]



An early draft of a paper, for which publication clearance was previously requested, which was submitted to ASPLOS 2017, the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems. This version was prepared for blind submission. At the request of the conference, the version actually submitted did not contain the footnote citing the sponsors. We were told that if the paper was accepted, the sponsor citation could be added before final publication.

# Hardware-based Isolation for Secure Execution of VMs

[In a Cloud Computing Environment] \*

## ABSTRACT

In computing systems, virtualization is the primary method of separating different users computation and data. The set of resources allocated for a user is exposed as a virtual machine (VM) and user workloads interface with a hypervisor for using physical computer resources. This abstraction can be further leveraged for isolating and protecting VMs from not only each other but from the underlying hypervisor.

In cloud systems, and in particular Infrastructure as a Service (IaaS) clouds, the cloud service provider (CSP) controls and manages the hypervisor. In current designs, this requires the customers to trust that the CSP is not malicious, and the CSP-managed hypervisor is secure against attacks that can compromise the applications of the customers. These assumptions lead to increased concerns over the security and privacy of customer data hindering the adoption of cloud services.

In order to isolate sensitive customer applications from other VMs as well as all privileged software, we present a modification to the Power Architecture called Access Control Manager (ACM). ACM is a hardware and firmware security enforcement mechanism prototyped in a server-class processor. ACM intercepts hypervisor accesses to secure VM data, and presents it in a form that protects integrity and confidentiality. Our architecture allows secure execution of a virtual machine (VM) transparently to the applications within the VM.

## 1. INTRODUCTION

\*Note: This material is based on research sponsored by the Department of Homeland Security (DHS) Science and Technology Directorate, Cyber Security Division (DHS S&T/CSD) via BAA 11-02; the Department of National Defense of Canada, Defense Research and Development Canada (DRDC); and Air Force Research Laboratory Information Directorate via contract number FA8750-12-C-0243. The U.S. Government and the Department of National Defense of Canada, Defense Research and Development Canada (DRDC) are authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Department of Homeland Security; Air Force Research Laboratory; the U.S. Government; or the Department of National Defense of Canada, Defense Research and Development Canada (DRDC).

ACM ISBN 978-1-4503-2138-9.  
DOI: 10.1145/1235

In cloud computing, customer applications rely on the system software managed by the CSP for providing services, including providing access to the system resources. However, in current designs, this also means the CSP-managed software has total control over the customer's data. The operating system or the hypervisor can access or modify the data of the application, or tamper with any security features implemented in userspace, without being detected. Therefore, all CSP-managed software has to be a part of the *trusted computing base* (TCB), and customers are forced to trust that: the entities that develop, configure, deploy, and control the CSP-managed software are not malicious, and the CSP-managed software itself is secure against attacks that can compromise the confidentiality and integrity of the customers' applications. This broad trust requirement is often difficult to justify and poses a significant risk, or prevents the adoption of public cloud services.

CSP-managed software, that includes the hypervisor and the operating system, comprises millions of lines of code, developed by large and often disparate teams of programmers. While these components are responsible for isolating executables as well as entire virtual machines from one another to protect sensitive information, they are themselves exposed to security vulnerabilities.

For example, a critical vulnerability in a virtual floppy disk controller in QEMU, called Venom, was discovered [21]. Venom allows the attackers to escape the VM into the host, and to access the data of other VMs. This vulnerability was introduced in 2004, and it was available in most major virtualization software for over a decade.

In order to provide protection that is independent of the security of the entire CSP infrastructure, we are introducing the Access Control Monitor (ACM) that provides secure isolation of VMs and applications from one another and from system software. ACM [5] is implemented in hardware and a software component called ACM firmware. At a high level, ACM protects physical pages that belong to a secure VM (SVM) from being accessed in clear text from other software. When system software accesses a secured page, ACM presents the page in a form that protects integrity and confidentiality.

Figure 1 shows the overall architecture of ACM. Each ACM-enabled processor has a set of asymmetric keys that are used for protecting the secrets of the SVM. An SVM is a regular VM that is packaged for ACM with its secrets encrypted with the public ACM key of the target processor. The private ACM key is protected by the Trusted Platform Module (TPM) and becomes available only when the correct ACM firmware is loaded during boot. The trusted entity that manufactures and distributes the ACM-enabled processors issues certificates for their public keys.

The ACM protection mechanism is based on an assignment of processes and their data to security domains. The hypervisor or the

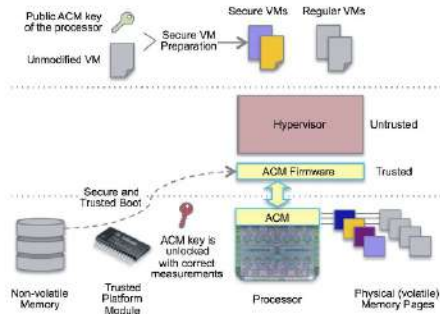


Figure 1: Overview of the ACM architecture

operating system (OS) is in the same security domain as all the normal VMs (NVMs), those not isolated (protected) by ACM security capabilities. Each of the SVMs is assigned to its own security domain so that its data (and state) can be protected from the others. Secure entity identifiers (SEIDs) represent the security domains in the hardware. Hardware enforces the isolation boundaries associated with security domains based on SEIDs.

ACM firmware runs in ultravisor mode, a new CPU mode that is at a privilege level above hypervisor mode. This firmware, along with the ACM hardware, is responsible for maintaining SEIDs associated with processes and memory and enforcing the associated access restrictions. We implemented our architecture using a full function simulator [8] by modifying the model for a Power 8 processor [26]. In addition, this firmware provides oversight of all hypervisor services, such as page table management, that must be coordinated with SEID management. Hardware mechanisms are used to invoke ACM firmware whenever a transition between security domains occurs, to enable the ACM firmware to assure that the state of a process in one domain is properly isolated from the state of a process in another domain. Finally, subsystems that can interact with the memory independently of the processor also have to be modified so that the ACM firmware can validate their accesses before they occur.

### 1.1 Contributions

This paper makes the following contributions:

- We introduce ACM, an Access Control Mechanism for cloud computing, that protects the confidentiality and integrity of a VM from software outside the VM including privileged software that is used to manage the cloud computing environment and malware outside the VM that may be able to obtain root privilege by exploiting a bug in the privileged software.
- We discuss the high-level architecture of ACM as well as an implementation of that architecture on an IBM POWER server.
- We demonstrate that Linux/KVM as well as an unmodified VM can boot on a type-1<sup>1</sup> implementation of ACM.
- We have reported on the initial performance measurements of this architecture.

<sup>1</sup>Type-1 ACM firmware required no paravirtualization Type-2 ACM firmware requires paravirtualization.

## 2. THREAT MODEL

ACM protects the confidentiality and integrity of an SVM's code and data from the other software on a system including the hypervisor and other VMs, whether the SVM is in execution or at rest. Even a privileged attacker, such as a malicious or compromised hypervisor, cannot access or tamper with (without detection) an SVM's memory pages or disk blocks. ACM makes no attempt to guarantee availability in the face of a hostile hypervisor. In other words, Denial of Service is out of the scope of the threat model.

ACM ensures that no unintended state of an SVM can leak to the hypervisor because of an asynchronous interrupt or when the VM makes a call to the hypervisor. An SVM's registers are protected from the hypervisor, and are saved and restored securely upon a context switch. ACM allows SVMs to share memory with one another. It also enables the SVMs to share unprotected pages with the hypervisor. ACM makes no attempt to protect network I/O, as this is addressed by existing technologies, such as TLS and IPSEC.

A malicious hypervisor can observe an SVM's memory access patterns, and measure the time that SVM takes before a context switch. In extreme cases, such side channel information may leak some private information. ACM does not protect against these side-channel attacks. Moreover, ACM does not encrypt data in memory unless it is accessed by an unauthorized entity. As a result, ACM may be susceptible to hardware attacks like memory probing. Also, persistent memory retains its contents even after loss of power. Consequently, if an SVM were running in persistent memory the unencrypted contents could be available after power down. Such hardware attacks are out of scope of the threat model. We note that hardware attacks such as memory probing are extremely difficult on POWER architecture because of its design of memory controllers.

Security is ultimately limited by the correctness of an SVM and the application(s) running inside the SVM. Logical or semantic weaknesses in the application or the guest OS, such as an exploitable buffer overflow, can allow an attacker to compromise an application or the SVM and gain access to confidential information. Properly written software can leverage ACM features to protect its secrets.

## 3. RELATED WORK

There is a long history of using hardware features to isolate software from a compromised system software, starting with XOM [20]. Table 1 compares Secure Executables to related work.

XOM [20] is a set of architecture changes that extends the notion of program isolation and separate memory spaces to provide separation from the OS as well. They introduce "secure load" and "secure store" instructions, which tell the CPU to perform integrity verification on the values loaded and stored. For this reason, the *application transparency* is limited; developers must tailor a particular application to this architecture.

Aegis [28] fixes an attack in XOM by providing protection against replay attacks. This requires using an integrity tree to protect memory. The authors also offer optimizations specific to hash trees in this environment that greatly reduce the resulting performance overhead. These optimizations include batching multiple operations together for verification, and delaying the verification until the time where the values affect external state. Again, the approach is not transparent to developers: the set of sensitive code areas must be explicitly specified by the developer. Additionally, the developer must identify the specific calculations requiring expensive integrity checkpoints, which are the only places tamper checking is performed. In contrast, ACM maintains integrity guarantees whenever the data is accessed by the system software.

	Flicker	SP	XOM	Aegls	Overshadow	SB++	SGX	Iso-X	ACM
<b>Requirements</b>									
Works without hardware changes	✓	X	X	X	✓	X	X	X	X
No OS in Trusted Code Base	Card OS in TCB	✓	✓	✓	Host OS in TCB	✓	✓	✓	No hypervisor in TCB
Works without OS support	✓	X	X	X	✓	X	X	X	✓
<b>Protection</b>									
Code privacy (transparently)	X	X	X	✓	X	✓	✓	✓	✓
Resilient to Memory Replay Attacks	✓	X	X	✓	✓	✓	✓	✓	✓
Protection from Physical Attacks	X	✓	✓	✓	X	✓	✓	X	✓
<b>Features</b>									
Multiple Simultaneous Instances	X	X	✓	✓	✓	✓	✓	✓	✓
Multi-threading/Multi-core Support	X	X	X	X	✓	✓	X	X	✓
Support Shared Memory Regions	X	X	X	X	X	✓	X	X	✓
Virtualization Support	X	X	X	X	✓	✓	X	X	✓

Table 1: Comparison with related works

**Secret-Protecting Architecture [19]** provides a mechanism to run code in a tamper-protected environment. The authors design in [13] a mechanism extending a root of trust to supporting devices. However, this technique does not have the transparency of other techniques. For example, there can be only one Secret-Protected application installed and running at a given point in time. The ability of protected applications to make use of system calls is likewise limited.

**TPM (Trusted Platform Module)**, widely deployed on existing consumer machines, can be used to provide guarantees that the operating system that was booted has not been tampered with. It does not exclude the operating system from the trusted code base, however. Instead, during a trusted boot all the software in the system is measured before it is executed and extended (or hashed) into a PCR (platform configuration register). Using remote attestation the state of the PCRs can be securely transmitted to a third party who can confirm that they are as expected. Linux IMA (Integrity Measurement Architecture) can use measurement and the TPM, if present, to protect against (detect and prevent execution) modification of modules and configuration files while the system is running.

A separate class of approaches, including Flicker, uses the TPM late-launch feature to protect software from a compromised operating system.

**Flicker [23]** provides a hardware-based protection mechanism that functions on existing, commonly deployed hardware (using mechanisms available in the TPM). As with other solutions, this incurs only minimal performance overhead. It provides a protected execution environment, but without requiring new hardware changes. The trade-off is that software has to be specifically developed to run within this environment. The protected environment is created by locking down the CPU using TPM late-load capabilities, so that the protected software is guaranteed to be the only software running at this point. System calls and multi-threading in particular do not work for this reason. Moreover, hardware interrupts are disabled, so the OS is suspended while the protected software is running. Thus, software targeted for Flicker must be written to spend only short durations inside the protected environment. The advantage of Flicker is the reduced hardware requirements: it is supported by existing TPM-capable processors.

**Overshadow [11]** provides guarantees of integrity and privacy protection similar to AEGIS, but implemented in a virtual machine monitor instead of in hardware. This approach has several other advantages as well: making the implementation transparent to software developers—for example, software shims are added to protected processes to handle system calls seamlessly. This means,

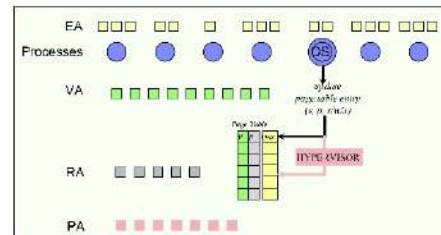


Figure 2: Current memory addressing scheme in server-class POWER processors.

however, that there is no protection provided against malicious system software. The *cloaking* approach, where the OS sees an encrypted version, is similar to the mechanism employed in ACM.

**SecureBlue++ [9] [10]** uses encryption and integrity trees for securing all data that leaves the processor. It provides isolation at the process granularity while supporting multi-threading and memory sharing. SecureBlue++ uses system call wrappers, which transparently manage system calls from a secured application, otherwise unmodified. The packaging of secure executables with public key encryption is similar to the packaging of VMs in ACM. SecureBlue++ protects the integrity of cache lines, which can result in an integrity tree with approximately 14% space overhead.

**SGX [24]** (Software Guard Extensions) is recently introduced in Intel processors. SGX protects portions of an application, called *enclaves* that are explicitly entered and exited. This reduces the transparency, since the developer has to slice the application and protect the interface. Haven [7] shows how SGX can protect an unmodified binary by creating a unikernel inside the enclave, but as pointed out in an SGX manual [17] enclaves should be made as small as possible because of the hard limit on protected memory size. Similar to SecureBlue++, SGX uses an integrity tree for enclave memory and its size is limited to 128 MB.

**Iso-X [14]** considers a threat model identical to ACM. However, the granularity of the protected portions is similar to SGX, and requires explicitly placing the secure code in a *compartment*.

## 4. MEMORY ACCESS CONTROL

### 4.1 POWER Memory Addressing

The current memory addressing scheme of POWER systems [22, 26] is represented in Figure 2. The yellow boxes at the top represent pages (size not defined) of memory. The blue circles below represent processes. One of those processes is the OS that manages resource allocation and scheduling for all the other processes in the system. The OS has a single virtual address space represented by the green boxes. The OS maps from effective addresses (EA), yellow boxes, to virtual addresses (VA), green boxes. The VA space can be significantly larger than the physical memory. The page table is used by the hardware to map from VA to physical addresses (PA). If there is no hypervisor, then the OS manages the page table and the associated mappings. If there is a hypervisor, then the server is divided into several partitions each of which appears to be a machine. The hypervisor owns the page table, the OS is relegated to mapping from VA to real addresses (RA). The hypervisor intercepts all actions of the OS on the page table and changes the RA specified by the OS to a PA.

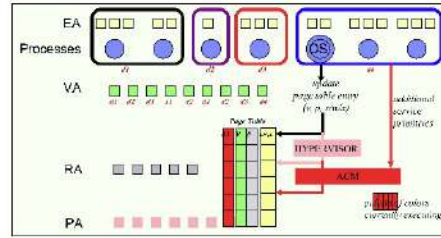


Figure 3: Power memory addressing and ACM domains.

This current model has the following properties:

- OS manages the translations EA to VA to RA and installs in the page table along with read/write/execute permissions.
- Hardware enforces the accesses permissions as specified by the page table.
- If a hypervisor is present, it intercepts the page table updates, translates RA to PA and then inserts the PA into the page table.
- For isolation, the hypervisor maps different EAs and VAs to distinct RAs.
- Sharing can be accomplished by either sharing the VA or the RA.
- In all cases the hypervisor retains access to OS pages

#### 4.2 ACM Memory Controls

ACM extends the addressing systems represented in Figure 2 with the concept of security domains as illustrated in Figure 3. The architectural concept referred to as a color in this section is represented by an SEID in the hardware implementation.

A security domain is simply a disjoint set of processes and their associated memory (EA). In Figure 3 the domains are illustrated by rectangular boxes surrounding processes and their associated memory (EA). Note that, as illustrated in Figure 3, the system software can be in a separate domain from the processes it is managing. Security domains have the following properties:

- Security domains are disjoint collections of processes; in particular all processes of a VM could be in one domain.
- They enforce data isolation and sharing across domains (the word "data" connotes both code and data pages).
  - Private data, that is data that is not supposed to be shared, in one domain cannot be accessed by another domain.
  - Designating and sharing of specific data across domains is permitted by explicit sharing-control primitives.
  - ACM assures that for cross domain operations only explicit parameters are passed between the domains, i.e., no parameter not explicitly needed for the sharing operation are passed.
  - ACM assures that no information leaks out of a domain due to an asynchronous interrupt.

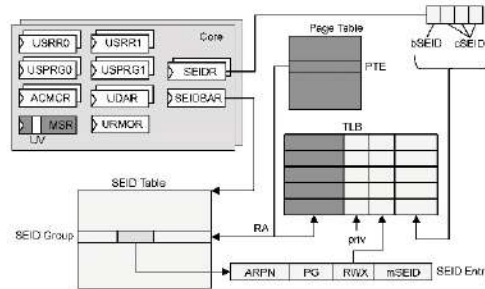


Figure 4: ACM registers and address translation

- The system software continues to do resource scheduling (CPU and memory) for all processes in all domains, and the system software can employ conventional techniques of address mapping to accomplish isolation and sharing of data/code among processes within a domain.
- The system software retains responsibility for paging. It must work with ACM to assure that pages have the correct color.

#### 4.3 Coloring of Private and Shared Pages

Coloring is introduced into ACM to facilitate controlled sharing of some pages.

Each domain is assigned a unique color as shown in Figure 3 where the colors assigned to domains are represented by  $d_1$ ,  $d_2$ ,  $d_3$ , and  $d_4$ . Each memory page that is assigned to a domain is given the same color as the domain. In Figure 3 domain  $d_3$  has two unique pages (VA) assigned to it, shown in green color. Any set of pages that is shared with two or more domains is also given a unique color. Shared pages are represented by  $s_1$  and  $s_2$  in Figure 3, where we can see that there are two pages with color  $s_2$  and one page with color  $s_1$ . Each domain also has associated with it a palette of colors. This palette contains the color of the domain and the color of every shared memory page that the domain is allowed to access. In order to enforce isolation and sharing, ACM manages the palettes and color mappings, as shown in Figure 3.

### 5. HARDWARE SUPPORT FOR MEMORY ACCESS CONTROL

This section describes hardware support for ultravisor mode. The hardware is described in terms of changes to an existing IBM POWER 8 processor [15, 26]. We added a new bit to the MSR, added seven

unchanged	HV	UV	unchanged	
0	3	4	5	63

Figure 5: One new bit, the UV, bit has been added to the MSR. When  $UV = 0$  the thread is not in ultravisor mode. When  $UV = 1$  If  $HV = 1$  and  $PR = 0$  (the problem state bit is not illustrated) the thread is in ultravisor mode. Other combinations are reserved.

reserved	NV	NX	NF	IS	SV	SE	DV	DE
32	56	57	58	59	60	61	62	63

Figure 6: The ACM Control Register (ACMCR) is a 32 bit register.

new control registers, modified the Page table layout, added a new table that is shared between the ACM firmware and the hardware, modified the operation of existing interrupts, we added some new interrupts, modified some existing instructions, and added some new instructions. All of our modifications to preexisting hardware are backward compatible.

### 5.1 Changes to existing control registers

A previously reserved bit in the MSR, as illustrated in figure 5, is used to indicate that ultravisor mode is active.

### 5.2 New Control registers

ACM adds six new per thread registers, ACMCR, USPRG0, USPRG1, USRR0, UDAR and USRR1, and one new per core register URMOR. The ACM control register is illustrated in figure 6. The remaining double word registers are modeled after existing hypervisor privileged registers. [ref power architecture document]. The formal names are as follows:

- **URMOR** Ultravisor real mode offset register.
- **USPRG0** and **USPRG1** Ultravisor mode special purpose registers 0 and 1 are only available to the ACM firmware.
- **USRR0** and **USRR1** ultravisor machine status save/restore registers 0 and 1. These registers hold the machine state at the time an interrupt, USRR0 is the instruction that was interrupted and USRR1 is the MSR.
- **UDAR** Ultravisor data address register. Set to the effective address associated with the storage access that caused an SEID interrupt.

The function of these registers is the similar to the hypervisor version, which all start with an H instead of a U as the first character.

The bits in ACMCR fall into two groups, those that inform ACM firmware about facilities used by the thread and those that control/influence the operation of the hardware. Three bits, vector unavailable, VSX unavailable, and FP unavailable, indicate whether the corresponding facility is available in the current SVM. ACM firmware only saves state associated with these features during a state transition when the state is being used by a thread. There are two bits, SV and SE, that tell the ACM firmware what kind of secure entity is active an SVM or a secure executable. When an SVM is active hypervisor directed interrupts are intercepted by the ACM firmware. There are two bits, DV and DE, that tell the hardware what kind of state transitions have to cause an ultravisor control interrupt. If DV is set all changes to LPIDR cause an ultravisor

E	reserved	Base Address	reserved	SZ
0	1	4	45	59 63

Figure 7: SEIDBAR: When bit 0 is 1 the ultravisor is enabled. The base address field contains the high order 41 bits of the 60 bit real address of the SEID table. The size field is used to generate an index into the table. The SEID table is referenced by both the ultravisor and hardware.

bSEID	c SEID 1	c SEID 2	c SEID 3
0	16	32	48

Figure 8: The SEID register contains a base and three compound SEIDs each 16 bits.

control interrupt. Finally there is one bit, IS, that is used to direct external interrupts to either the ACM firmware or the hypervisor. by controlling the operation of the hardware, the number of interrupts to the ultravisor can be reduced.

### 5.3 SEID support

The new ultravisor mode resources are two new registers and the SEID table. The SEID represents the color associated with a secure VM, the new registers are a per core SEIDBAR (SEID Base Address Register) and the per thread SEIDR (SEID Register). The SEIDBAR, illustrated in figure 7, has two functions enabling ACM and defining the SEID table.

The SEID register, illustrated in figure 8, contains the base SEID (16 bits) of the current secure entity and space for three additional colors. The bSEID field in SEIDR contains the base SEID for the current process. This SEID identifies the security domain in which the process executes. SEID are assigned by ACM firmware when a secure VM is first entered. ACM firmware assigns a single SEID to the hypervisor when it first transfers control to it. All normal virtual machines started by the hypervisor inherit this SEID.

Compound SEIDs are used to represent sharing relations among secure domains. When a page is to be shared between two or more secure domains, an otherwise unused SEID is assigned to that page, and added to the palette of SEIDs that each of those security domains are allowed to access. The cSEID fields in the SEIDR are used to cache up to three such compound SEIDs for the current process.

### 5.4 Page table modifications

Secure pages are those that belong to an SVM. Secure pages are indicated by a previously reserved bit, the SA bit, bit 6 of the second double word of a page table entry. When set it indicates that the page belongs to a secure virtual machine.

### 5.5 SEID Table

The SEID Table is a variable-sized data structure that specifies the SEID(s) associated with each 4KB block of real address space. The SEID table can be any size  $2^n$  bytes where  $19 \leq n \leq 43$ . The starting address must be a multiple of its size. The SEID Table contains SEID Table Entry Groups (STEG). A STEG contains 16 SEID Table Entries (STEs) of eight bytes each. ACM firmware is responsible for atomically updating the SEID table.

Each SEID Table Entry (STE) contains an mSEID for one 4KB block of real address space. Figure 9 shows the layout of an SEID table entry. The ARPN (Abbreviated Real Page Number) field of the STE provides a tag to disambiguate entries within a STEG.

//	ARNP	PG	//	RWX	SEID
0	4	40	43	45	48 63

**Figure 9: SEID table entry** The reserve fields in the SEID table entry are marked with //. ARPN is the Abreviated real page number, PG is an encoding of the page size (000 4K, 011 1GB, 100 16MB, 101 64KB, 110 16GB, 111 2MB, RWX are read, write execute permission for the page, and SEID is the mSEID of the page.

The SEID table is maintained by ACM firmware and used by the hardware.

During address translation when a page table search succeeds in finding a real address (RA) where the corresponding PTE[SA] bit is asserted, an SEID Table search is performed to validate access to that real address. If the PTE[SA] bit is negated, the SEID Table search is bypassed, and access is allowed.

Every real address is associated with a single SEID Table entry group (STEG). The starting address of that group is calculated by combining a hash of the real address with the base address specified in SEIDBAR.

Once the STEG is located, up to 16 entries in the group must be tested to determine if any match the current access. When the ultravisor is running SEID enforcement is bypassed. Otherwise for a match to exist, the following conditions must be satisfied

- The abreviated real page number must match
- The SEID must match one of the SEIDs of the secure entity.
- The access bit, RWX corresponding to the type of access must be set.
- The page size, STEPG, must be at least as large as is specified in the corresponding PTE

If the SEID table search succeeds, the access is allowed. In that case, the translation determined by the PTE, along with the PTE[SA] bit, the SEID, and RWX fields of the matching STE, are cached with the translation, as appropriate.

If the SEID table search fails, an instruction SEID mismatch exception or a data SEID mismatch exception occurs, depending on whether the real address is for an instruction fetch or a data access.

Conceptually, the SEID Table is an extension of the Page Table. When the Translation Lookaside Buffer (TLB) is used to cache page table entries, the corresponding TLB entries include enough information to allow the access controls defined by the SEID mechanism to be enforced. In particular For example, the PTE[SA] bit and the SEID and RWX bits from the SEID Table are included in a new TLB entry when it is created. Consequently, for a TLB hit this information must match when PTE(SA)=1, otherwise it is ignored.

When ACM firmware makes changes to the SEID table, it must perform the appropriate TLB invalidate operations to maintain consistency of the TLB with the SEID Table.

## 5.6 New Interrupts

There are three new interrupts the ultravisor control interrupt, The data SEID fault interrupt and the instruction SEID fault interrupt. All of these interrupts pass control to the ACM firmware.

### 5.6.1 UltravisorControl Interrupt

The ultravisorcontrol interrupt can only occur when ultravisor-mode is enabled (SEIDBAR<sub>0</sub> = 1). When enabled it will always be generated if the the PTCR (Page Table Control Register) register is

changed while not in ultravisor mode. It will also be generated when an SVM is running, (ACMCR<sub>SVM</sub> = 1), and there is a transition from hypervisor to problem state via an **brfid** or **rfid**. Finally, if the ACM firmware has requested notification of VM changes, (ACMCR<sub>DEV</sub> = 1), it will be generated if LPLDR is changed. Whenever the ultravisor control interrupt occurs USRR0 is Set to the effective address of the instruction that caused the interrupt. USRR1 is set to a "copy" (some bits are always set to zero) of the MSR at the time of the interrupt. Control is passed to the ultravisor through its interrupt vector.

### 5.6.2 Data SEID Mismatch Interrupt

A Data SEID Mismatch interrupt occurs when a data access cannot be performed because of an SEID fault for the real page being accessed.

An SEID fault occurs if an SEID in the SEIDR of the entity does not match the SEID of the page. For the SEID to match the access also has to match which means matching the page size, access type (RWX) and, for a cached translation matching the privilege level.

USRR0 and USRR1 set similar to the ultravisor control interrupt. Except that bits 33,34, and 35 in USRR1 are used to indicate where the fault occurred (they are all zero for the ultravisor control interrupt). Bit 36 and bits 42:47 are set to 0 (as in the ultravisor control interrupt). The remainder of the register is loaded from the MSR.

### 5.6.3 Instruction SEID Mismatch Interrupt

An Instruction SEID Mismatch interrupt occurs when instruction access cannot be performed because of an SEID fault (as previously defined) for the real page being accessed by the current process. USRR0 and USRR1 are set in the same way as for the Data SEID mismatch interrupt.

## 5.7 Modified Interrupts

A few of the existing interrupts have been modified to account for the changes introduced by ACM. In ultravisor mode all of these are reflected to the ACM firmware. For example, The trace interrupt does not trace the **urfid** instruction. Whenever a Hypervisor Data Storage interrupt occurs, it is reflected to the ACM firmware with HEIR register is set to a copy of the instruction that caused the interrupt.

For Floating-Point Unavailable, VSX Unavailable, and Vector Unavailable these interrupts occur when an instruction is executed from one of these facilities and the ACMCR indicates that the facility is not available (ACMCR<sub>SF</sub> = 1, ACMCR<sub>NV</sub> = 1, or ACMCR<sub>KX</sub> = 1).

A Hypervisor Virtualization interrupt occurs and is reflected to the ACM firmware when ACMCR indicates that this should occur. If ACMCR<sub>IS</sub> = 0 then these interrupts go to the hypervisor and when ACMCR<sub>IS</sub> = 1 They are reflected to the ACM firmware.

When a System Call instruction is executed bits 42 and 43 in SRR1 can be used to distinguish between **ucall**, **hcall** and **syscall**. The LEV field in the SC instruction LEV=0 is a **syscall**, LEV=1 (both bits off) is a **hcall** and LEV=2 (42 on, 43 off) (or LEV=3 42 off 43 on) is a **ucall**.

When one of these interrupts occurs the following registers are also set: USRR0 is set to the effective address of the instruction where the interrupt occurred. USRR1 is a copy of the MSR except bits 33-36 and bits 42-47 are cleared.

## 5.8 New Instructions

We added ultravisor return from interrupts (URFID) and a Real Mode invalidate Entry global (RMIEG) instructions.

The URFID instruction is ultravisor mode privileged instruction

19		//	//	//	306	/
0	6		11	16	21	31

Figure 10: Format of the ultravisorreturn from interrupt,URFID instruction

31		//	//	RB	???	/
0	6		11	16	21	31

Figure 11: Format of the real mode invlaide entry global, RMIEG, instruction

and illustrated in figure 10. The next instruction executed will be taken from USRR0 and the new MSR will be generated from USRR1. If no pending exceptions are enabled, then the next instruction is fetched, under control of the new MSR value. Otherwise, the highest priority interrupt will occur and the value placed into SRR0, HSRR0 or USRR0 by the interrupt processing mechanism is the address of the instruction that would have been executed next had the interrupt not triggered.

The RMIEG instructions is a new hypervisor privileged instruction for invalidation of real mode translations. It invalidates translation caches on all threads in the system. The format of the instruction is illustrated in figure 11. All entries in the translation cache that translate the indicated EA and have a page size that is equal to or smaller than the page size specified in RB are invalidated by setting the valid bit to 0, and remaining fields to undefined values.

## 5.9 Modified Instructions

Figure 12 shows the layout of the system call instruction. When the ultravisor mode is enabled, SEIDBAR<sub>L</sub> = 1, if LEV=1 then the hypervisor is invoked and if LEV=2 the ACM firmware is invoked. When the ultravisor mode is not enabled, SEIDBAR<sub>E</sub> = 0, bit 5 of the LEV field (instruction bit 25) is reserved with a value of 0. Consequently if LEV=2, the supervisor is invoked, and if LEV=3, the hypervisor is invoked.

The system linkage instructions rfid, rfalse, and hrfid are modified to not change the value of MSR<sub>UV</sub>. The Move To/From System Register Instructions are modified to accept the register numbers for the new Ultravisor special purpose registers.

## 6. BUILDING TYPE 1 ACM firmware

The ACM firmware uses the previously described hardware to assign SEIDs (or color) resources within the systems, memory, and threads of control. This separates those resources which are protected from one another and from those resources which are not protected. When ultravisor mode is enabled the ACM firmware is the software that receives control after secure and trusted boot. Once activated ACM firmware initialize all of its internal structures, measure and passes control to the next software usually a hypervisor or OS. During this process ACM firmware also extracts the private key of the machine from the TPM and extends the PCR so that no other software has access to the key.

The hardware described in section 5 enable the ACM firmware to keep memory separated by the use of the SEID table (see section 5.5). During normal execution ACM firmware only receives control as a result of an interrupt. There is also no timer that expires causing an interrupt to return control to ACM firmware. A full implementation of ACM firmware must be thread safe and re-entrant. The prototype version of ACM firmware reported on in this document is currently single threaded. Interrupts are not disabled while

17		//	//	//	LEV	1	/
0	6		11	16	20	30	31

Figure 12: Format of the System Call instruction

ACM firmware is running. On a server-class processor there are multiple cores and multiple threads per core. There will only be one copy of the ACM firmware which will run on every core and every thread.

When ultravisor mode is active, even if no SVMs are executing ACM firmware must handle all SEID (secure Entity ID) faults (the SEID table is filled on-demand); Handle ACM firmware control calls; and monitor page tables.

ACM firmware controls all resources in the machine that could allow a Hypervisor or OS to inspect a Secure Virtual machine. Control is transferred to ACM firmware when any component in the system touches one of these resources. ACM firmware will also get control when any software touches (read, write, or execute) memory that is not assigned to it. If ultravisormode is enabled, control will be transferred to ACM firmware when the target of the syscall indicates its target is ACM firmware or if a syscall targeting the hypervisor is made by an SVM. Finally if ACM firmware indicates the need to detect VM changes or it wants to receive hypervisor virtualization interrupts, it will receive control whenever the hypervisor changes LPIDR (see section 5.2)

ACM firmware will not support KSM [6] for secure virtual machines. Because ACM firmware uses the TAPM [18] mode of AES [27, 12], even if two secure virtual machine have identical pages, when encrypted they will appear differently in memory. Because KSM cannot be allowed to see the unencrypted contents, it would find nothing to share. Also allowing KSM to operate when SVMs are running causes a large number of unnecessary encryptions and decryptions.

The hypervisor is still responsible for paging virtual machines, consequently it must have access to the SVM memory. ACM firmware assures that whenever access is granted to an SVM's memory, it is encrypted with integrity. Integrity protection enables ACM firmware to determine whether any component that had access to SVM memory, made an unauthorized modification. The insecure page feature provided by ACM enables an SVM to securely communicate with another entity assuming it has appropriately protected the contents of the insecure memory. This feature also enables the SVM to utilize I/O.

## 6.1 VM Assumptions

Our work considers Linux VMs running on the PowerLinux QE-MU/KVM environment. In this environment a VM disk image consists of PReP boot, boot, and root partitions. The PReP Boot partition contains the bootloader, i.e., grub.core. The boot partition contains Linux kernels and their initial-RAM file system (initramfs) images. The root partition contains the root file system.

We make the following three assumptions about VM images that are converted into SVMs<sup>2</sup>. First, the root partition is encrypted using dm-crypt [25, 1], a kernel level block-layer encryption mechanism. During boot, the dm-crypt passphrase is provided to the kernel to unlock the encrypted partition where the root file system resides. Second, Linux kernel images are digitally signed, and Linux distro public keys are available for verifying kernel signatures. Finally, files in the initramfs are digitally signed and the

<sup>2</sup>Tooling to convert a VM into an SVM is not discussed in this paper.



kernel verifies their signatures during startup.

Although PowerLinux kernels are not currently digitally signed, the building blocks exist for implementing secure boot for PowerLinux VM kernels. The kexec utility, which performs a soft-boot by replacing the image of a running kernel with another, has been integrated with the Integrity Measurement Architecture (IMA) [16], which is part of the Linux integrity subsystem. This integration [2] enables images as well as files in the initramfs loaded by kexec to be measured and appraised by IMA.

## 6.2 SVM Image

Though VMs have some security features, e.g., dm-crypt, VMs still have security gaps at rest and during execution; the PRoP boot and boot partitions and the VM's memory can be accessed and tampered with. We resolved these gaps by creating a secure bootloader that enables ACM/memory protection early in the SVM boot sequence.

The secure bootloader provides integrity protection of the PRoP boot partition, invokes ACM memory protection, and performs a secure boot of the user selected VM kernel. Invoking ACM memory protection early in the boot process addresses runtime security and memory allocation issues. It prevents memory tampering even before the VM kernel is loaded and executed in memory. It also resolves memory allocation issues. When a VM enters secure mode, any previously allocated memory (i.e., malloc-ed before going secure) is automatically zeroed, by ACM firmware when referenced by the VM for the first time in secure mode. Thus, going secure early avoids any potential memory initialization issue during Linux kernel startup.

## 6.3 SVM Boot Sequence

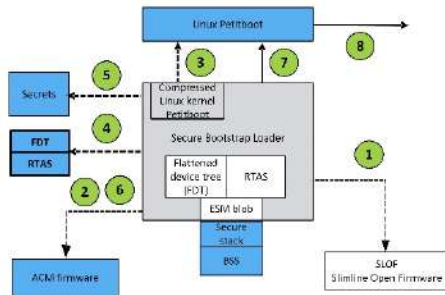


Figure 13: Secure bootloader startup flow.

The secure bootloader performs three basic operations: (1) retrieves from Slimline Open Firmware (SLOF) [4] the device tree and Run-time Abstraction Service (RTAS) object, which provides access to system dependent services during runtime; (2) requests ACM protection by invoking esm ucall passing a reference to the esm blob; and (3) decompresses and loads Linux-Petitboot<sup>3</sup>. The petitboot application then loads and boots the VM kernel in the boot file system. Figure 13 shows the secure bootloader interacting with SLOF, the ACM firmware and other components in memory when bootstrapping a VM's kernel.

## 7. EVALUATION

<sup>3</sup>This is a bootable Linux image configured to run petitboot [3], a user level bootloader application based on kexec.

## 7.1 Demonstration

The demonstration we developed for our funding agency has four phases 1) Booting Linux/KVM, 2) Booting a Normal virtual machine (NVM) on top of Linux/KVM, 3) Booting a secure virtual machine (SVM) on top of Linux/KVM, and 4) Running a test program that illustrates the security features of the architecture and the protections afforded an SVM. At the end of the demo, ACM firmware terminates an SVM when it detects an integrity fault, indicating that the SVM memory has been tampered with. Tables 3 and 4 are the performance measurements taken during the demo.

Phase one starts with booting a Linux/KVM hypervisor. During this boot, the dynamic construction of the SEID table by the ACM firmware can be observed (if the proper flags are set). This demonstrates that an unmodified hypervisor runs (boots) on top of ACM firmware. After the hypervisor boots phase two of the demo starts. We boot an unmodified<sup>4</sup> normal virtual machine (NVM) which confirms that an NVM runs on top of the hypervisor running on ACM firmware. In phase 3 we boot a secure VM (SVM). Booting of the SVM requires h-call support and vio support in ACM firmware. In order to boot secure virtual machines, we built the boot loader described in section 6.3.

After the SVM has booted the final phase, phase 4, runs a script that queries some information in the NVM from the hypervisor, Displays the information in the NVM to show that they are the same. Modifies the NVM information from the hypervisor and then queries the modified information from inside the NVM to show that the modifications worked. We modified the qemu-monitor-command so that it could modify the contents of a VM memory. This command is used by our demo script to display and modify VM memory. Next the demo script queries the "secrets" of the SVM from the hypervisor, which shows that the hypervisor cannot see the secrets. Then the demo queries the secrets from inside the SVM to illustrate that they are unchanged. Next we query the SVM secrets again from within the hypervisor which shows that each time the hypervisor looks into the SVM the encrypted information looks different even though nothing was changed. We query the secrets from within the SVM again to show that they are still unchanged. Next we modify the secrets in the SVM from within the hypervisor. Finally from within the SVM we query the secrets again. This causes a decryption of the page the hypervisor modified. Consequently, ACM firmware detects the integrity fault that was caused by the hypervisor modifying the page and terminates the SVM.

## 7.2 Performance

Boot	ACM	KVM	NVM	SVM	Demo
ACM Events	0	4,194,421	2,810,554	3,836,687	251,955
ACM Cycles	3,995,304	9,868 M	4,981 M	5,647 M	5,652 M
Run Cycles	3,996,335	20,596 M	24,335 M	21,019 M	9,316 M
ACM Failures					1

Table 2: Cycle counts for initial hardware design for initializing ACM, booting KVM, booting an NVM, booting an SVM, and the demo described in section 7.1. These cycle counts correspond to the measurements in table 3.

ACM firmware collects Time Base on entry to or exit from major components. Time Base is an architected register supported in our simulator [8] that counts the machine cycles from reset. Since we can boot Linux/KVM and boot multiple (our Demo boots two)

<sup>4</sup>A virtual machine that boots on an unmodified P8 running Linux/KVM as the hypervisor. The binary of the VM was moved to the simulator for this phase.

virtual machines on top, the prototype ACM firmware is sufficient to get initial measurements of the overheads imposed by this approach.

Our performance measurements were taken during the execution of the demonstration described in section 7.1. Each table has five columns Init ACM, boot KVM, boot NVM, boot SVM, and Demo. These columns identify the measurements taken during these phases of the demo. The init ACM phase covers time zero until ACM passes control to the hypervisor. On the left hand side of both tables are the ACM firmware operations we measured. Operations starting with uv are ultravisor initialization (init), control (ctl), zeroing of memory (zero), exception (excp), encryption and decryption (iapm), and hashing (hash). Operations that begin with uc are tracking ultravisor calls. The tables break out two calls starting an SVM (esm) and communicating with slof (slof) all others are relatively minor and captured under uc(call). Those that begin with pte are related to page table management: creating a new page table (new), creating a secure page table (sec) and handing a data storage interrupt to a page table (dsi). The line that begins with vio, is the cycle count for type-1 virtual I/O support. The line that begins with fp is the cost of handling floating point unavailable interrupts<sup>5</sup>. The lines that begin with isi and dsi are the cycle count of processing dsi and isi seid faults. The hv(mmio) is the cycle count for type-1 support for MMIO. The line tagged hv(wrap) is the cycle count for protecting h-calls from the SVM to the hypervisor. It is important to note that the measurements associated with Init ACM are one time cost of starting a machine. Also the measurements associated with pte(new) and pte(sec) are the cost of creating a page table when a VM starts.

Table 2 gives an overview of the performance of the architecture as describe in section 5. The first thing we notice is that there is a 92% overhead for booting LinuxKVM. This is excessive but since it is a one time cost it may be acceptable. We also see that there is a 26% overhead for booting a NVM. This is also excessive and is a result of having the ACM firmware track all page table writes. Finally the overhead on the SVM is 34%. These last two overheads are not one time cost. The fact that this overhead is mostly associated with page table management can be confirmed by examining table 3. While KVM is booting, it constructs a page table that covers all of the physical memory it knows of. Once the ACM firmware finds out about this table, it must go through the page table and confirm that there are no sensitive translations. Going through the newly constructed table is represented by the pte(new) entry in the table. Managing the modifications made to the page tables is represented primarily by the pte(dsi) line in the table.

Table 3 give the detailed performance breakdown for the previously discuss functions. Table 4 list the average number of cycles for each measured operation per phase. In these two tables a blank means that the operation did not occur in the phase. We can see from these two tables that page table magement is a significant cost for both the NVM and SVM.

## 8. DISCUSSION

This paper reports on our protections for VM running on a server, stand alone or in a cloud. The protection is rooted in hardware that starts with a secure and trusted boot. The private key of the server is sealed to a PCR that includes a measurement of ACM firmware. If that measurement is not correct the ACM firmware will not have access to the private key and will not be able to run any secure executables. While ACM firmware is initializing itself it gets a

<sup>5</sup>Measures when the ultravisor had to do a delayed state save for floating point and vector registers.

operation	Init ACM	boot KVM	boot NVM	boot SVM	Demo
uv(init)	3,995,304				
pte(new)		587 M	11 M	11 M	
pte(sec)				9,919,570	
pte(dsi)		2,211	2,092	2,101	
vio(dsi)				3,914	
isi seid		281,884		5,189	4,890
dsi seid		2,398	2,567	12,384	5,509
uv(iapm)				4,861	4,883
uv(hash)				154,707	
uv(zero)				182	3,582
fp*unavl		361		413	377
uv(ctl)		414	405	405	406
uv(excp)				447	463
hv(mmio)				360	
hc(wrap)				75	2,270
uc(slof)				12,677	
uc(call)				645,236	
uc(esm)				26 M	

Table 4: This table list the average cycle count (total cycles divided by number of occurrences) for each of the operations described in table 3.

random number from the TPM which it uses to seed its random number generation process. Once ACM firmware has the private key it can run those executables that have been configured for it

This proposal uses cryptography to protected the SVM while it is stored on disk. Each SVM includes a region, called the blob that is encrypted under the private key (or keys) of the machine (or machines) that are authorized to execute it. The blob contains the root of the integrity tree that protects the VM and the symmetric key used to encrypt the SVM. It also contains a clear text, with integrity protection, that code that is executed when the VM starts. This code contains a syscall to the ACM firmware requesting a transition to secure mode. As has been discussed the ACM firmware decrypts the blob. The blob contains all secrets, such as DM crypt pass phrases needed to start the SVM. Finally it checks that areas of the SVM that have only been integrity protected have not been modified. If these checks passes ACM firmware generates a run time key that will be used to protect the SVM when it is in computer memory.

When the SVM is in memory ACM firmware assures that the clear text is only available to the executing SVM. An integrity tree protects all of memory and the virtual address is part of the encryption key. These features prevent replay attacks and attacks based on moving a legitimate block of encrypted memory. The hypervisor and any other components that have a legitimate right to access the SVM memory only see an encrypted form. To facilitate I/O and proper communications with the hypervisor the SVM can mark pages as unprotected and/or shared with the hypervisor. Unprotected pages have no confidentiality or integrity information. unprotected page does not have to be shared with the hypervisor.

In a full implementation approximately 118 h-calls have to be protected. The ACM firmware provides wrappers which guarantee that only the state necessary to preform the requested function is transmitted to the hypervisor and only the results of the function are returned from the hypervisor. In the prototype all of the implemented hcall wrappers are in the ACM firmware. This approach is transparent to a VM that will be converted to a SVM, but it increases the size of ACM firmware.

The approaches described in this paper includes enabling I/O for SVMs. This is a desirable feature, and has to be considered as alternatives are explored. On the other hand, not enabling I/O make the security analysis of a SVM somewhat simpler.

During this project we discussed building type-1 and type-2 ACM firmware. We felt that a type-1 implementation would me more

operation	Init ACM		boot KVM		boot NVM		boot SVM		Demo	
	Count	Cycles	Count	Cycles	Count	Cycles	Count	Cycles	count	Cycles
uv(init)	1	3,995,304								
pte(new)			1	587 M	1	11 M	1	11 M		
pte(sec)							1	9,919,570		
pte(dsi)			418,467	9,249 M	2,264,173	4,737 M	1,572,156	3,303 M	234,257	521 M
vio(dsi)							1,672	6,544,812		
isi seid			2	563,790			12,819	66 M	505	2,469,929
dsi seid			12,947	31 M	5,109	13 M	93,927	1,163 M	2,790	15 M
uv(iapm)							8,521	44 M	389	1,189,851
uv(hash)							4,144	641 M		
uv(zero)							39,617	7,255,621	753	2,696,866
fp*unavl			2	722			19,823	8,189,570	397	149,825
uv(ctl)			2	828	541,272	219 M	470,473	190 M	1,594	648,105
uv(excp)							1,665,713	745 M	12,412	5,752,586
hv(mmio)							488	176,046		
hc(wrap)							1,481,803	111 M	8,480	19 M
uc(slof)							53	671,902		
uc(call)							6	3,871,421		
uc(esm)							1	261 M		

Table 3: Performance measurements from initial architecture. Each of the four phases of the demo are measured separately. For each operation in each phase we list the number of occurrences under count and the total cycles consumed by all occurrences under cycles.

easily accepted assuming the performance were acceptable. The type-1 ACM firmware represents an upper bound for the size of ACM firmware. One of our goals is for the ACM firmware to be small enough for some formal verification process. While we have done formal verification of the model presented in section 4, the implementation described in this paper differs from the one we verified.

ACM and ACM firmware cannot protect against poorly written software or security exposures that exist inside the SVM whether in its applications or operating system. For example this includes the usage of a web browser that goes to a site that exploits a vulnerability in the browser to compromise the SVM. A libOS or micro kernel approach promises to reduce the OS to its minimal footprint which would minimize the potential exposures within an SVM. Other approaches are required to deal with these issues.

## 9. FUTURE WORK AND CONCLUSIONS

We have demonstrated that it is possible to create a type-1 ACM firmware. The advantage of not having to include the infrastructure or cloud provider inside the security domain is significant. However, the associated overhead, runtime and boot, is higher than we were targeting. As was observed, ACM alone is not sufficient to protect against all threats to VMs. When security is a concern, the overheads associated with booting an SVM may be acceptable. However, we feel that the current overhead can be reduced.

As the work continues, we will explore techniques for reducing the impact of this architecture. We believe that the overheads quoted in this paper can be reduced by paravirtualization. Changing the hardware approach may also affect the overhead. The type-1 ACM firmware presented in this paper has to discern what the NVM and hypervisor are doing without hints. We expect that the reported overhead can be reduced with a type-2 implementation since paravirtualization would give hints to the ACM firmware, substantially reducing the cost of execution. It is also future work to determine the correct balance between new hardware and paravirtualization of the hypervisor and/or SVMs.

One of our goals is to keep the ACM firmware small enough to be formally verified through some mechanism. Consequently it is critically important to minimize the size. One alternative approach is to package as much of the hcall wrapper as possible with the VM. This makes the tooling somewhat more complex but minimizes the

size of the ACM firmware. A disadvantage of this is exposing the wrapper code to attacks from a compromised VM. While the current prototype exceeds the size of systems that have been formally verified in the past. Perhaps by changing the architecture concurrently with the increasing capability of verification systems we may be able to achieve this objective.

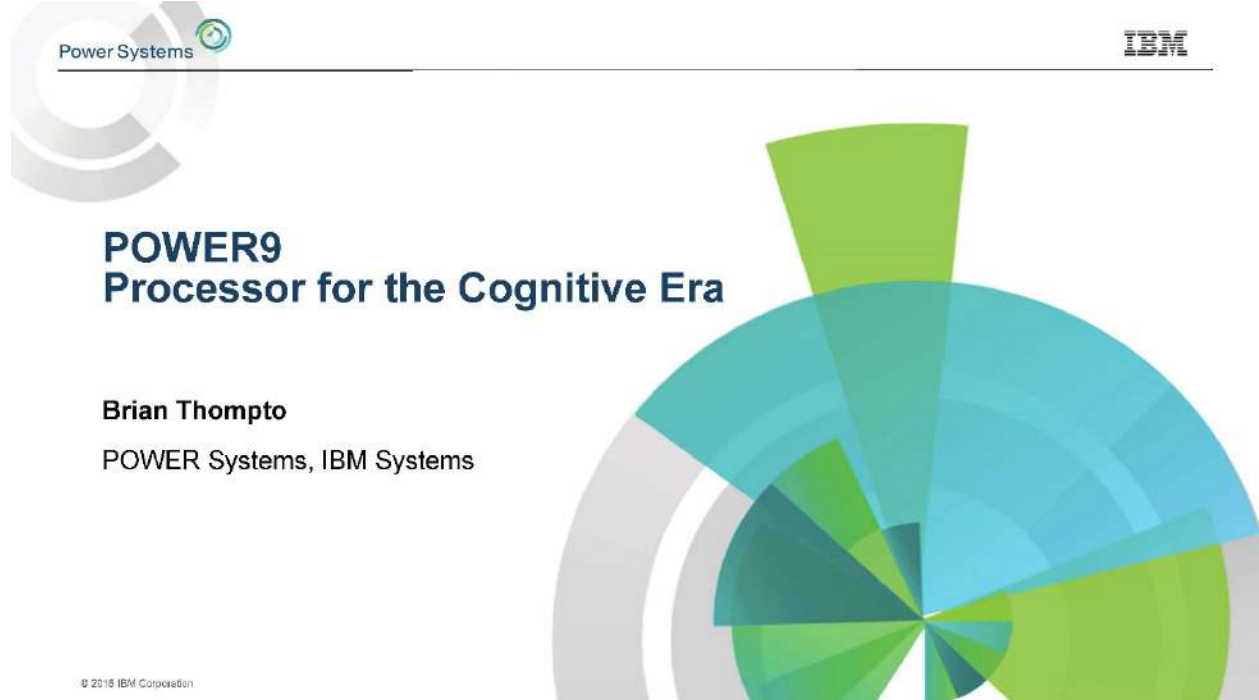
## References


- [1] dm-crypt: Linux kernel device-mapper crypto target. <https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt>.
- [2] [patch v4 00/19] vfs: support for a common kernel file loader. <http://comments.gmane.org/gmane.linux.file-systems/104832>.
- [3] Petitboot. <http://git.ozlabs.org/?p=petitboot;a=summary>.
- [4] Slimline Open Firmware. <http://git.qemu.org/SLOF.git>.
- [5] S. R. Ames Jr, M. Gasser, and R. R. Schell. Security kernel design and implementation: An introduction. *Computer*, 16(7):14–22, 1983.
- [6] A. Arcangeli, I. Eidus, and C. Wright. Increasing memory density by using ksm. In *Proceedings of the linux symposium*, pages 19–28. Citeseer, 2009.
- [7] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, Aug. 2015.
- [8] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. Van Hensbergen, and L. Zhang. Mambo — a full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, Mar. 2004.
- [9] R. Boivie. Secureblue++: Cpu support for secure execution. Technical Report RC25287, IBM Research, May 2012.
- [10] R. Boivie and P. Williams. Secureblue++: Cpu support for secure executables. Technical Report RC25369, IBM Research, April 2013.

- [11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dvoskin, and D. R. Ports. Over-shadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, 2008.
- [12] J. Daemen and V. Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.
- [13] J. S. Dvoskin and R. B. Lee. Hardware-rooted trust for secure key management and transient trust. In *Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07*, pages 389–400, New York, NY, USA, 2007. ACM.
- [14] D. Evtuyshkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47*, pages 190–202, Washington, DC, USA, 2014. IEEE Computer Society.
- [15] IBM. *Power ISA™ Version 2.07*. Power.org, May 2013. soft copy distribution: <http://www.power.org/documentation>.
- [16] Integrity Measurement Architecture. <http://sourceforge.net/p/linux-ima/wiki/Home>.
- [17] Intel. *Intel Software Guard Extensions Evaluation SDK for Windows OS*, January 2016.
- [18] C. S. Jutla. Encryption modes with almost free message integrity. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 529–544. Springer, 2001.
- [19] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dvoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture, ISCA '05*, pages 2–13, Washington, DC, USA, 2005. IEEE Computer Society.
- [20] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *SIGPLAN Not.*, 35(11):168–177, Nov. 2000.
- [21] C. List. CVE-2015-3456. Available from MITRE, CVE-ID CVE-2015-3456., April 2015.
- [22] C. May, E. Silha, R. Simpson, H. Warren, et al. *The PowerPC Architecture: A specification for a new family of RISC processors*. Morgan Kaufmann Publishers Inc., 1994.
- [23] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 315–328, 2008.
- [24] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, pages 10:1–10:1. ACM, 2013.
- [25] M. Peters. Encrypting partitions using dm-crypt and the 2.6 series kernel. <https://www.linux.com/news/encrypting-partitions-using-dm-crypt-and-26-series-kernel>, June 2008.
- [26] B. Sinharoy, J. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. Brown, J. E. Moreira, et al. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2–1, 2015.
- [27] N.-F. Standard. Announcing the advanced encryption standard (aes). *Federal Information Processing Standards Publication*, 197:1–51, 2001.
- [28] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th Annual International Conference on Supercomputing, ICS '03*, pages 160–171, New York, NY, USA, 2003. ACM.

A set of slides extracted from a larger presentation given by Brian Thompto, an IBM employee, at the Hot Chips 2016 Conference, August 21-23, in Cupertino CA. This presentation was not generated by this project but is included as a reference for the technology transfer and commercialization status of the technologies developed under this project. The relevant slides are pages 4 and 10, and specifically the reference to “Hardware Enforced Trusted Execution”. The presentation is reference [7] in our list and can be found at

[http://www.hotchips.org/wp-content/uploads/hc\\\_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-final.pdf](http://www.hotchips.org/wp-content/uploads/hc\_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.921-.POWER9-Thompto-IBM-final.pdf)





Power Systems 

# POWER9 Processor for the Cognitive Era

**Brian Thompto**  
POWER Systems, IBM Systems

© 2016 IBM Corporation

Power Systems  **POWER9 Processor – Common Features** 

**New Core Microarchitecture**

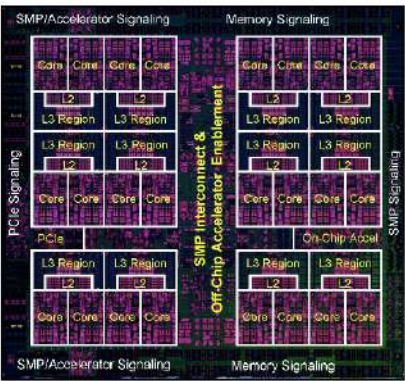
- Stronger thread performance
- Efficient agile pipeline
- POWER ISA v3.0

**Enhanced Cache Hierarchy**

- 120MB NUCA L3 architecture
- 12 x 20-way associative regions
- Advanced replacement policies
- Fed by 7 TB/s on-chip bandwidth

**Cloud + Virtualization Innovation**

- Quality of service assists
- New interrupt architecture
- Workload optimized frequency
- Hardware enforced trusted execution



**14nm finFET Semiconductor Process**

- Improved device performance and reduced energy
- 17 layer metal stack and eDRAM
- 8.0 billion transistors

**Leadership Hardware Acceleration Platform**

- Enhanced on-chip acceleration
- Nvidia NVLink 2.0: High bandwidth, advanced new features (25G Link)
- CAPI 2.0: Coherent accelerator and storage attach (PCIe G4)
- New CAPI: Improved latency and bandwidth, open interface (25G Link)

**State of the Art I/O Subsystem**

- PCIe Gen4 – 48 lanes

**High Bandwidth Signaling Technology**

- 16 Gb/s interface
  - Local SMP
- 25 Gb/s interface – 25G Link
  - Accelerator, remote SMP

© 2016 IBM Corporation

4

## **New Instruction Set Architecture Implemented on POWER9**

### **Broader data type support**

- 128-bit IEEE 754 Quad-Precision Float – Full width quad-precision for financial and security applications
- Expanded BCD and 128b Decimal Integer – For database and native analytics
- Half-Precision Float Conversion – Optimized for accelerator bandwidth and data exchange

### **Support Emerging Algorithms**

- Enhanced Arithmetic and SIMD
- Random Number Generation Instruction

### **Accelerate Emerging Workloads**

- Memory Atomics – For high scale data-centric applications
- Hardware Assisted Garbage Collection – Optimize response time of interpretive languages

### **Cloud Optimization**

- Enhanced Translation Architecture – Optimized for Linux
- New Interrupt Architecture – Automated partition routing for extreme virtualization
- Enhanced Accelerator Virtualization
- Hardware Enforced Trusted Execution

### **Energy & Frequency Management**

- POWER9 Workload Optimized Frequency – Manage energy between threads and cores with reduced wakeup latency



## **Special notices**

This document was developed for IBM offerings in the United States as of the date of publication. IBM may not make these offerings available in other countries, and the information is subject to change without notice. Consult your local IBM business contact for information on the IBM offerings available in your area.

Information in this document concerning non-IBM products was obtained from the suppliers of these products or other public sources. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. Send license inquiries, in writing, to IBM Director of Licensing, IBM Corporation, New Castle Drive, Armonk, NY 10504-1765 USA.

All statements regarding IBM future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

The information contained in this document has not been submitted to any formal IBM test and is provided "AS IS" with no warranties or guarantees either expressed or implied.

All examples cited or described in this document are presented as illustrations of the manner in which some IBM products can be used and the results that may be achieved. Actual environmental costs and performance characteristics will vary depending on individual client configurations and conditions.

IBM Global Financing offerings are provided through IBM Credit Corporation in the United States and other IBM subsidiaries and divisions worldwide to qualified commercial and government clients. Rates are based on a client's credit rating, financing terms, offering type, equipment type and options, and may vary by country. Other restrictions may apply. Rates and offerings are subject to change, extension or withdrawal without notice.

IBM is not responsible for printing errors in this document that result in pricing or information inaccuracies.

All prices shown are IBM's United States suggested list prices and are subject to change without notice; reseller prices may vary.

IBM hardware products are manufactured from new parts, or new and serviceable used parts. Regardless, our warranty terms apply.

Any performance data contained in this document was determined in a controlled environment. Actual results may vary significantly and are dependent on many factors including system hardware configuration and software design and configuration. Some measurements quoted in this document may have been made on development-level systems. There is no guarantee these measurements will be the same on generally-available systems. Some measurements quoted in this document may have been estimated through extrapolation. Users of this document should verify the applicable data for their specific environment.

Revised September 26, 2006

## Special notices (continued)

IBM, the IBM logo, IBM.com AIX, AIX (logo), IBM Watson, DB2 Universal Database, POWER, PowerLinux, PowerVM, PowerVM (logo), PowerHA, Power Architecture, Power Family, POWER Hypervisor, Power Systems, Power Systems (logo), POWER2, POWER3, POWER4, POWER4+, POWER5, POWER5+, POWER6, POWER6+, POWER7, POWER7+, and POWER8 are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries.

A full list of U.S. trademarks owned by IBM may be found at: <http://www.ibm.com/legal/copytrade.shtml>.

NVIDIA, the NVIDIA logo, and NVLink are trademarks or registered trademarks of NVIDIA Corporation in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries or both.

PowerLinux™ uses the registered trademark Linux® pursuant to a sublicense from IBM, the exclusive licensee of Linus Torvalds, owner of the Linux® mark on a world-wide basis.

The Power Architecture and Power.org wordmarks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

The OpenPOWER word mark and the OpenPOWER Logo mark, and related marks, are trademarks and service marks licensed by OpenPOWER.

Other company, product and service names may be trademarks or service marks of others.

## LIST OF SYMBOLS ABBREVIATIONS AND ACRONYMS

ACM	Access Control Monitor
AFRL	Air Force Research Laboratory
ARM	acorn RISC machine
BIOS	basic input output services
DDR	double data rate
DHS	Department of Homeland Security
DRAM	dynamic RAM
ELF	executable and linkable format
ESM	enter secure mode
FIFO	first in first out
eBIOS	extension to SEA BIOS
eTIS	emulated hardware TPM interface for QEMU
eTPM	swTPM emulation for QEMU
EVM	extended verification module
CPU	central processing unit
GPR	general purpose register
FPGA	field programmable gate array
IBM	International Business Machines
IMA	integrity measurement architecture
IoT	Internet of things
KVM	kernel virtual machine
KSM	kernel same page merging
LPC	low pin count
LXC	Linux containers
NVM	normal virtual machine
OAT	open attestation
PCR	platform configuration register
PPC	Power PC
QEMU	quick emulator
RAM	random access memory
RHEL	Redhat Enterprise Linux
RISC	reduced instructions set computing
RTAS	real time abstraction services
RTM	Root of trust for measurement
SB++	secure blue ++
SEID	secure entity identifiers
SGX	Software Guard Extensions
SLOF	simline open firmware
SPI	serial programming interface
SRAM	static RAM
STG	Systems and Technology Group
SVM	secure virtual machine
S&T	Science and Technology Directorate
swTPM	software based TPM emulator
TCB	trusted computing base
TCG	Trusted Computing Group



TCL	tool control language
TPM	Trusted Platform Module
TPMDD	TPM device driver
VM	virtual machine
vTPM	virtual TPM