SECURE AND ENERGY EFFICIENT EXECUTION FRAMEWORKS USING VIRTUALIZATION AND LIGHT-WEIGHT CRYPTOGRAPHIC COMPONENTS

Satyajeet Nimgaonkar

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2014

Nimgaonkar, Satyajeet. Secure and Energy Efficient Execution Frameworks Using Virtualization and Light-Weight Cryptographic Components. Doctor of Philosophy (Computer Sciences and Engineering), August 2014, 108 pp., 16 tables, 28 figures, 78 numbered references.

Security is a primary concern in this era of pervasive computing. Hardware based security mechanisms facilitate the construction of trustworthy secure systems; however, existing hardware security approaches require modifications to the micro-architecture of the processor and such changes are extremely time consuming and expensive to test and implement. Additionally, they incorporate cryptographic security mechanisms that are computationally intensive and account for excessive energy consumption, which significantly degrades the performance of the system.

In this dissertation, I explore the domain of hardware based security approaches with an objective to overcome the issues that impede their usability. I have proposed viable solutions to successfully test and implement hardware security mechanisms in real world computing systems. Moreover, with an emphasis on cryptographic memory integrity verification technique and embedded systems as the target application, I have presented energy efficient architectures that considerably reduce the energy consumption of the security mechanisms, thereby improving the performance of the system. The detailed simulation results show that the average energy savings are in the range of 36% to 99% during the memory integrity verification phase, whereas the total power savings of the entire embedded processor are approximately 57%.

ACKNOWLEDGMENTS

This doctoral research would not have been possible without the contributions of a number of people, to whom I am greatly indebted.

To my major professor, Dr. Mahadevan Gomathisankaran for his guidance, support, and patience throughout this research. He provided an exceptional environment for research and the endless brainstorming sessions helped me sharpen my analytic skills. His motivation, enthusiasm and immense knowledge are the driving force behind the successful completion of this research.

I would also like to thank my co-major professor, Dr. Saraju P. Mohanty and rest of the committee members, Dr. Krishna Kavi and Dr. Song Fu for their encouragements, insightful comments and critiques.

To my wife and best friend Pooja, for her sustained support, encouragement, patience, unyielding love, and sacrifice. She has guided me through the highs and lows in life and her words have often motivated me to do better and achieve bigger things. She has been the rock upon which the foundation of my life is built. I thank her for gifting me our beautiful son Ishan, who has brought a sense of calm, stability, and good luck in our lives.

My parents and grand parents have been instrumental in my success in higher education. They inculcated in me the importance of education. Their prayers and blessings have been a great source of inspiration and motivation to me. To my sister for her love and support. To my mother-in-law for all her belief and support in me and my decisions.

Most of all, I would like to thank the almighty God for His grace, blessings, and guidance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

OVERVIEW

The current trends in technology have allowed computing systems to evolve into complex forms. Personal computers, smart-phones, PDAs, network sensors, and network routers etc., have become ubiquitous in this era of computing. As the dependence on these systems increase, so also does the sensitivity of information stored in them. This information may include confidential personal data like secret passwords, credit card numbers and bank account numbers etc. Thus information security has now become imperative so as to prevent these systems from leaking out this critical information to unauthorized entities.

In general, computer security aims at providing confidentiality, integrity, and availability to computing systems. Confidentiality is breached when information is accessible to an unwanted and unauthorized entity. This entity could be a human, a software program or another computing system. Similarly, integrity is infringed when information is modified by such an entity and availability is broken when this entity succeeds in making the host computing system not serve legitimate user requests.

There are two approaches to provide security in a computing system. The first is to include software protection mechanisms like software obfuscation, watermarking, encryption, isolation, and so forth. The root of trust is entrusted in the security of the operating system (OS). However commodity OS are significantly large with huge code base and are often prone to security vulnerabilities. The most common software attack that exploits the software vulnerabilities in application code and OS is the Buffer Overflow Attack [11]. Software solutions share their memory with other softwares and the OS that could potentially contain a vulnerability and hence provide an entry point to an attack. Moreover, this kind of security cannot protect a system from a physical attack. Thus the security they provide is inadequate. The second solution is more effective. The idea is to add some hardware security mechanisms that can provide tamper evident and tamper resistant environment [44, 35]. These include secure processor architectures like ABYSS [71], AEGIS [67], Arc3D [22], XOM [38], and

HIDE [78] that propose modifications to the processor architecture. Typically they employ hardware Encryption/Decryption and Memory Integrity Verification (MIV) mechanisms to protect the confidentiality and preserve the integrity of the applications. Thus the root of trust is delegated in the security provided by the secure hardware architectures. Hardware support for security facilitates the construction of trustworthy secure systems. However, there are two significant issues that hinder the adoption of secure hardware architectures. These issues are described in Section 1.1 and form the crux of this research in terms of problem definition and motivation.

1.1. Problem Statement and Motivation

- The secure hardware architectures require changes to the micro-architecture of the processor and such changes are extremely time consuming and expensive to test and implement. This is where, we derive our first motivation. A virtualization layer can be used to test the security of an architecture. This provides a time and performance efficient alternative to actual hardware testing. Also, once an architecture is successfully tested for its security, the entire virtualized setup can be used to secure applications running on a cloud service platform.
- The second issue with these architectures is that they incorporate cryptographic security mechanisms that are computationally intensive and account for excessive energy consumption. While this may not be a big issue in desktop computers and large servers, it certainly becomes a huge problem in embedded systems. Embedded systems are highly resource constrained. Most of these devices are battery powered and it is essential to have minimal energy consumption in-order to achieve high speed and performance. Therefore, the motivation is to propose secure but light weight cryptographic alternatives so that the energy overhead imposed is minimal. In this research, the emphasis is specifically on reducing the energy overhead imposed by the MIV mechanism in embedded processors.

1.2. Novel Contributions of this Dissertation

The major contributions of this dissertation are as follows.

- The first contribution is to provide a platform to test the hardware security mechanisms in a time and cost effective manner. For this, we have proposed the Virtualization Based Secure Framework (vBASE) that leverages the power of virtualization technology to realize hardware security mechanisms inside the virtualization layer. This makes the testing process flexible, time & cost efficient and comparatively easy as against actual hardware testing. This is demonstrated in the proposed vBASE Testing framework.
- We then present the vBASE Execution framework — SecHYPE and CTrust with a primary goal to prove the practicality of secure hardware architectures, once they are thoroughly tested. The inclusion of hardware security inside the hypervisor leads to the design of a secure hypervisor framework - SecHYPE. With cloud computing as the target application, we propose the Cloud Trust architecture - CTrust that deploys the SecHYPE framework to provide root of trust and security to the applications running in the cloud.
- The focus of the research then shift towards the performance overhead imposed by the hardware security mechanisms. We have concentrated specifically on reducing the energy consumption of the cryptographic MIV mechanism. The first technique that we have proposed is the Memory Detect and Protect mechanism (MEM-DnP), which relies on a sensor module to detect any memory based attack on the system. The MIV operation is only carried out in an event of an anomaly i.e., when the fluctuations exceed the threshold value V_T. The simulation results show that the average energy savings are in the range of 85.5% to 99.998% during the integrity verification phase.
- The second technique is the Timestamps Verification (TSV) mechanism. This scheme is based on the principle of locality and uses timestamps to uniquely identify the memory block written to the memory. The simulation results show that the

energy savings are in the range from 36% to 81% during the integrity verification phase.

- Finally, we present the Hash Function-Less Verification mechanism to completely eliminate the performance cost of invoking a Hash Function during the MIV. The simulation results prove that this mechanism reduces the power consumption of an embedded processor by 57.24%. This is a significant improvement considering most embedded devices are battery powered.

1.3. Organization of this Dissertation

The rest of this dissertation is organized as follows. Chapter 2 presents our proposed Virtualization Based Secure Framework (vBASE). There are two flavors of vBASE framework — Testing framework that is used to test the security of a given secure hardware architecture and an Execution framework that constitutes the SecHYPE and CTrust, which are used for secure and trustworthy application execution in cloud computing platforms. Chapter 3 briefly discusses the cryptographic memory integrity verification (MIV) mechanism and describes the proposed MIV architecture and an algorithm to calculate total number of hash invocation required during the execution of a program. Chapter 4 describes our first energy efficient MIV solution known as the Memory Detect and Protect (MEM-DnP). Here, we have demonstrated how on-chip sensors can be used to detect a memory based attack on an embedded system. The MIV is executed only when there is an attack on the system. This leads to significant energy savings as exhibited in the simulation results. Chapter 5 presents our second energy efficient MIV solution, called the Timestamps Verification (TSV) mechanism. This scheme exploits the principle of locality to reduce the number MIV invocations, thereby reducing the energy consumption of the processor. Finally, Chapter 6 describes the Hash Function-Less Verification mechanism, where the aim is to completely eliminate the cost of invoking a hash function during the MIV process, thus significantly reducing the power consumption of the processor. Chapter 7 presents the conclusion of this dissertation.

CHAPTER 2

VIRTUALIZATION BASED SECURE FRAMEWORK (VBASE) [1]

2.1. Introduction

The recent years have seen a wide scale growth, adoption, and popularity of the virtualization technology (VT) [68], to provide efficient and cost-effective usage of expensive hardware. VT not only increases hardware utilization through server consolidation but also provides benefits for application development and testing [70]. VT introduces a software abstraction layer or virtualization layer (virtualization software) between the hardware and the operating system, thus decoupling them from each other. This software abstraction layer is known as the virtual machine monitor (VMM) [61] or the hypervisor. A VMM/hypervisor allows the user to create multiple virtual machines (VMs) on a single physical hardware platform, each capable of running an OS and its applications. A virtualization software emulates the underlying hardware to provide a known interface for the OS and applications to run on. The virtual CPU, memory, storage, and the introspection APIs in a VMM/hypervisor can be modified to mimic any given secure hardware architecture. This makes it easier, time, and cost efficient to test the security of a hardware architecture as compared to on-chip testing. Besides, the combination of hardware secure architectures with the virtualization software can strengthen the security of the hypervisors. They can then be used to provide trust and security to the applications running in cloud computing systems. With this motivation, we have presented two different designs of the vBASE framework: vBASE Testing and vBASE Execution framework.

The vBASE framework is implemented on top of the XEN VMM/hypervisor [5], an open-source virtualization software. Therefore our framework uses some of the components already provided by XEN. However, it is important to stress that the security mechanisms implemented in the vBASE framework are generic to any virtualization software and so the

[1]Parts of this chapter have been previously published, either in part or in full, from [34] with permission from IASTED and Actpress, [50] with permission from IEEE and [51] with permission from Academy of Science.

vBASE framework can be easily ported to another VMM/hypervisor. Since vBASE adheres to the structure of XEN VMM, it is worthwhile to discuss briefly, the architecture of XEN before presenting the vBASE framework. The XEN VMM/hypervisor is the basic abstraction layer (virtualization software) that resides directly on the hardware below the OS. It emulates the underlying hardware and is responsible for CPU scheduling and memory partitioning in order to allow multiple VMs to run on a single hardware platform. XEN controls the execution of all the VMs running on it, however it has no knowledge of networking, external storage devices, video, or any other common I/O functions found in a computing system. The XEN structure defines two types of VMs: Domain 0 (DOM 0) and Domain U (DOM U). Domain 0 is a privileged VM that has the capability to access physical I/O resources and communicate with other VMs running on the system. In its simplest form, a Domain 0 is a modified Linux kernel, that must be running on all XEN platforms before any other VM could be started. Domain U is an unprivileged VM that is provided to the users to host their applications.

The rest of this chapter is organized as follows. Section 2.2 describes the vBASE Testing framework and its core software components. Section 2.3 presents the vBASE Execution framework that constitutes the secure hypervisor framework - SecHYPE and the cloud trust architecture - CTrust. The prototype implementation of vBASE framework is presented in Section 2.4 followed by detailed analysis in Section 2.5. Section 2.6 details the previous related research. Finally Section 2.7 presents the conclusion of the vBASE research project.

2.2. vBASE Testing Framework

The primary goal of the proposed vBASE Testing framework is to provide a test bench to evaluate the security capabilities of given hardware architecture. This provides a time and cost efficient alternative to on-chip hardware testing. The framework is shown in Figure 2.1. The main components of this framework are - secure architecture plugin interface (SAPI), secure application, event trigger mechanism, monitor and the controller.

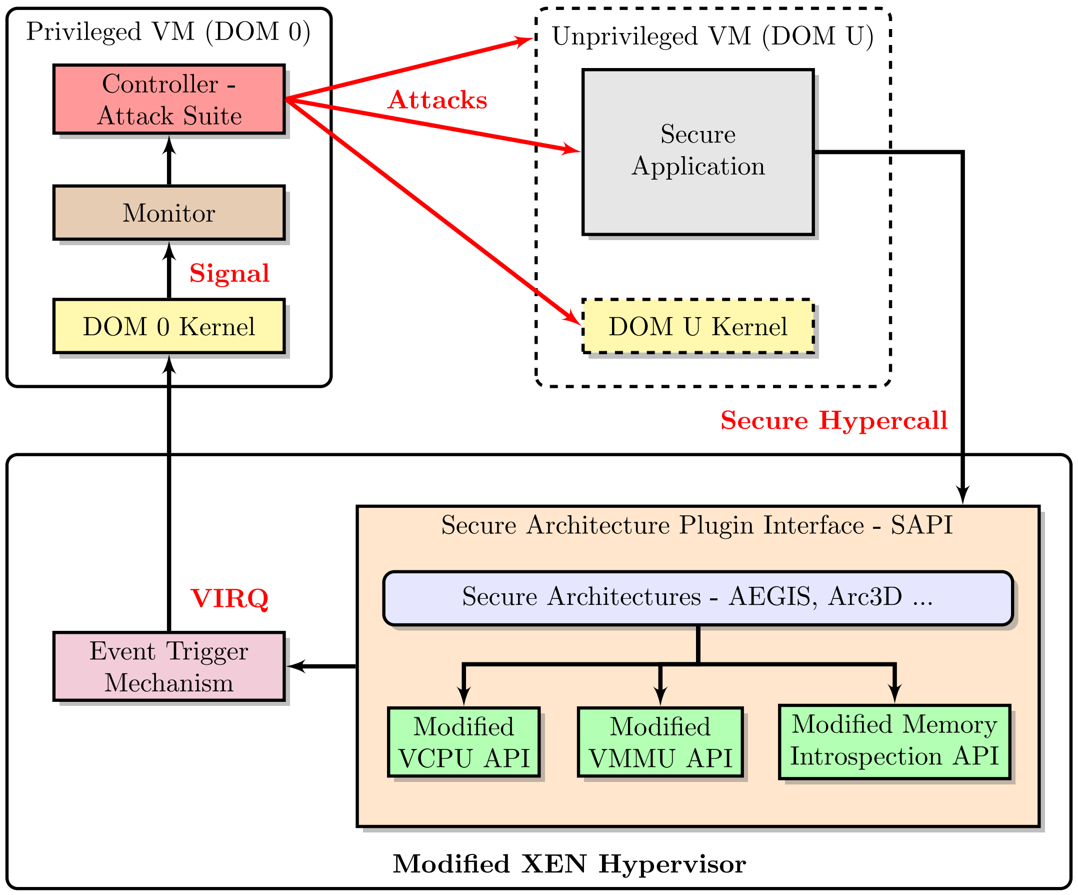


FIGURE 2.1. vBASE Testing Framework

2.2.1. Secure Architecture Plugin Interface (SAPI)

The SAPI is a generic interface exposed by the vBASE framework to embed any secure architecture within the XEN VMM. SAPI is a collection of APIs which are used to modify the virtual CPU (VCPU), virtual memory management unit (VMMU) and provide memory introspection functions into the modified components of XEN. Secure architectures are typically changes to the CPU and memory of the hardware which facilitates secure execution and isolation of a process. These components are available as software modules in XEN. These modules are modified to resemble a particular secure architecture. As shown in Figure 2.1, the SAPI primarily has three components: modified VCPU API, modified VMMU API and modified memory introspection API. The modified VCPU API has all the set of functions which modifies the Virtual CPU provided by XEN. These modifications include providing encryption and decryption capabilities to the processor, add additional secure registers, modify or implement new cache memories etc. The modified VMMU has set of functions which enhance the existing virtual memory management unit provided by XEN. These changes include switching on/off the conventional virtual memory layout of the OS, encryption and decryption functions for main memory and adding additional access restrictions to the secure process memory pages. The modified memory introspection API enhances the memory monitoring features provided by XEN.

2.2.2. Secure Application

This is a special process whose data needs to be protected from attacks on the system. This process is aware of the security mechanisms provided by the secure architecture plugged in to the vBASE framework. Secure applications or processes protect their confidential data by storing them in specialized protected memory regions.

2.2.3. Event Trigger Mechanism

Event trigger provides a mechanism to initiate inter-VM communication. Traditionally, XEN provides some mechanisms and tools like split device driver, xenstore, grant tables, and ring buffers to carry out inter-VM communication. However the primary drawback of these mechanisms and tools is that they need the support from DOM U kernel to initiate communication. The vBASE framework assumes all DOM U kernels to be untrustworthy and vulnerable to attacks. Hence instead of using the traditional mechanisms, a new hypercall mechanism independent of the DOM U kernel, has been developed. A hypercall works similar to a system call in an OS. It is an interrupt, INT 82h in XEN that is used to switch the control between the kernel and the XEN hypervisor. The hypercall interrupts the processor and is trapped in XEN using the hypercall handler functions. A new mechanism was developed to interrupt the processor from ring-3 of a DOM U VM. This is developed by using the CPUID instruction. The CPUID is meant to report processor features to user-level applications. A custom code — 0x92h was developed as an argument for the CPUID instruction that can trap into the XEN VMM from DOM U user-space. At that point, the control is transferred to the hypercall handler which will issue a virtual interrupt (VIRQ) to the DOM 0. VIRQ is a software interrupt, notified by setting up a bit in the Virtual CPU data structure present in XEN. It is used for communication from XEN to the VM kernel.

2.2.4. Monitor

It is a program running in the DOM 0 that receives notifications about the secure application, through the Event Trigger mechanism. It is responsible for monitoring critical events pertaining to the secure application and detect any possible attacks on it. It contains a monitoring script, which is a collection of watch events and actions to be performed when a particular event occurs. During secure execution, monitoring is carried out at multiple layers such as, application layer: secure and normal applications, OS layer, hypervisor layer and the secure architecture layer. In the application layer, common events that are monitored are API function calls, library calls, network API calls and messages encapsulating secret keys etc. At the OS layer, monitoring is done for software interrupt handlers, virtual memory mappings, memory accesses, I/O accesses, process scheduling and so on. At the hypervisor layer, common events are instruction execution, interrupt handlers, memory translations, page fault mechanism and so forth. Finally, the events monitored at the secure architecture layer are secure instruction entry and exit points, secure page fault mechanism, accesses to secure memory locations and registers etc.

2.2.5. Controller

The secure hardware architectures must be rigorously tested against various attacks. The Controller is a comprehensive attack suite made up of host and network based attacks. During the execution of the secure application, the controller mounts attacks on different layers of the system stack. A vulnerability at any layer in the stack is exploited to generate an attack scenario. At the application layer, an attack is attempted to breach the confidentiality and integrity of process's code and data, inject malicious code during function calls, exploit library calls, intercepting network traffic etc. At the OS layer, attacks attempt to modify the virtual memory mappings, values passed to software interrupts and system calls, kernel level data structures, I/O messages and so forth. At the hypervisor layer, attacks are aimed to tweak the inter-VM memory mappings, parameters passed to hypercalls and interrupts and I/O data packets etc. Finally, the plugged-in secure architecture is also attacked by trying to read/modify its secure registers, modifying its essential data structures, attempt to leak its cryptographic keys and so on.

2.2.6. Framework Functioning

The first step of the functioning is to plug-in a secure architecture by using the generic interface provided by the SAPI. Once this is done, the VCPU, VMMU and virtual memory introspection functions are appropriately modified to align with the plugged-in secure architecture. All the information pertaining to the security mechanisms provided by the secure architecture is then reported to the Monitor through the Event Trigger mechanism. The Monitor, thus now has complete visibility and understanding of the secure regions within the secure architecture. The secure application can then start executing in an unprivileged VM. A secure application identifies its critical confidential data and then transitions to secure execution state. In the proposed framework, this is achieved by invoking a secure hypercall enter_vbase. Once the secure application invokes this hypercall, the secure architecture and the Monitor are informed about the secure execution of the application. Its now the responsibility of the secure architecture to protect the confidential data of the application, while the Monitor is responsible for tracking all the necessary events related to the application. While the application is executing securely, the Monitor informs the Controller to conduct a series of attacks on the secure application. The secure application can exit secure execution by invoking the exit_vbase hypercall. At this instant, the Monitor stops tracking the events while the Controller suspends its attacks. Once the secure application terminates, all the events recorded by the Monitor are logged. This log serves as a benchmark to judge the effectiveness of the security mechanisms provided by the secure architectures.

2.3. vBASE Execution Frameworks - SecHYPE and CTrust

This Section presents the second type of vBASE framework called the Execution framework. The target application of this framework is cloud computing. It has been designed and developed with an intention to provide trust and security to the applications executing in cloud computing platforms.

2.3.1. Introduction

Cloud computing [46], envisions universal access to a shared pool of configurable resources such as compute, storage, network, and software. In the recent years, cloud computing platforms have procured great success due to its cost effectiveness, flexibility, increased storage, and elasticity. It provides on-demand computing, where the customer has complete control on its services to the finest granularity and can tailor them based on their needs. Cloud computing allows consolidation of resources thus enabling new applications. Currently e-commerce, on-line auctioning companies, travel agencies and other such services use clouds to provide services to their users.

In-spite of several advantages, the adoption of cloud computing is impeded by critical security issues. The National Institute of Standards and Technology (NIST), has identified security as a primary concern in cloud computing. Security issues in cloud arise at different levels: infrastructure level, software service level and user level. Prior research [9, 64, 10, 56] exemplify how security vulnerabilities in cloud are exploited by motivated attackers. A recent global information security survey, Into the Cloud, Out of Fog [28] conducted by Ernst & Young in November 2011 states that 72% of respondents see an increasing level of risk due to increased external threats in cloud computing.

Cloud computing currently relies heavily on VT to virtualize the CPU and storage resources so as to meet its elastic demands. VT provides security by isolating the execution of each OS and its applications in a VM. Thus a security breach in one VM does not affect the working of other VMs. However, isolation security is inadequate to protect an application from a malicious OS or another application running in the same VM. Thus additional security measures have to be incorporated in the virtualization softwares to provide security in cloud computing platforms. The security challenges in cloud computing can be reduced to, how to establish root the trust and to protect the confidentiality and integrity of the applications in the cloud. A VMM can be used as the root of trust. Given a small footprint of the VMM, it would be reasonable to verify its correctness and thus the security of a VMM. As shown in Section 2.2, it is possible to plugin a secure hardware architecture inside a VMM and this modified secure VMM can be used to provide security to the applications executing in the cloud. Here, the VMs are run on top of the modified secure VMM. Thus the root of trust for the applications is relegated in the security of the VMM.

2.3.2. SecHYPE Framework

The idea behind the SecHYPE framework is to bolster the security features of a virtualization software, so that it can be used as a root of trust for user applications running in the cloud. The main advantage of this approach is that the underlying hardware need not be modified in order to accommodate additional security features. We have already demonstrated in Section 2.2 that it is feasible to realize and implement secure hardware architectures inside the XEN VMM. Once, a particular hardware architecture is thoroughly tested and proven to be secure, can be ported in cloud computing platforms to provide security to the applications. We call this secure hypervisor as the SecHYPE framework, shown in Figure 2.2.

The SecHYPE framework is very similar to the vBASE Testing framework in terms of its software components and overall operation. The only exception is the Controller component. The Controller is a software component of vBASE framework that is used to launch attacks on secure Applications during the testing phase. It is a consolidated attack suite that inter-operates with the Monitor program to thoroughly test the security features of the secure hardware architecture under test. However, there is no need to launch attacks on applications during the executon phase and therefore, the Controller is not present in the SecHYPE framework. The rest of the components and framework functioning remains the same as in the vBASE testing framework.

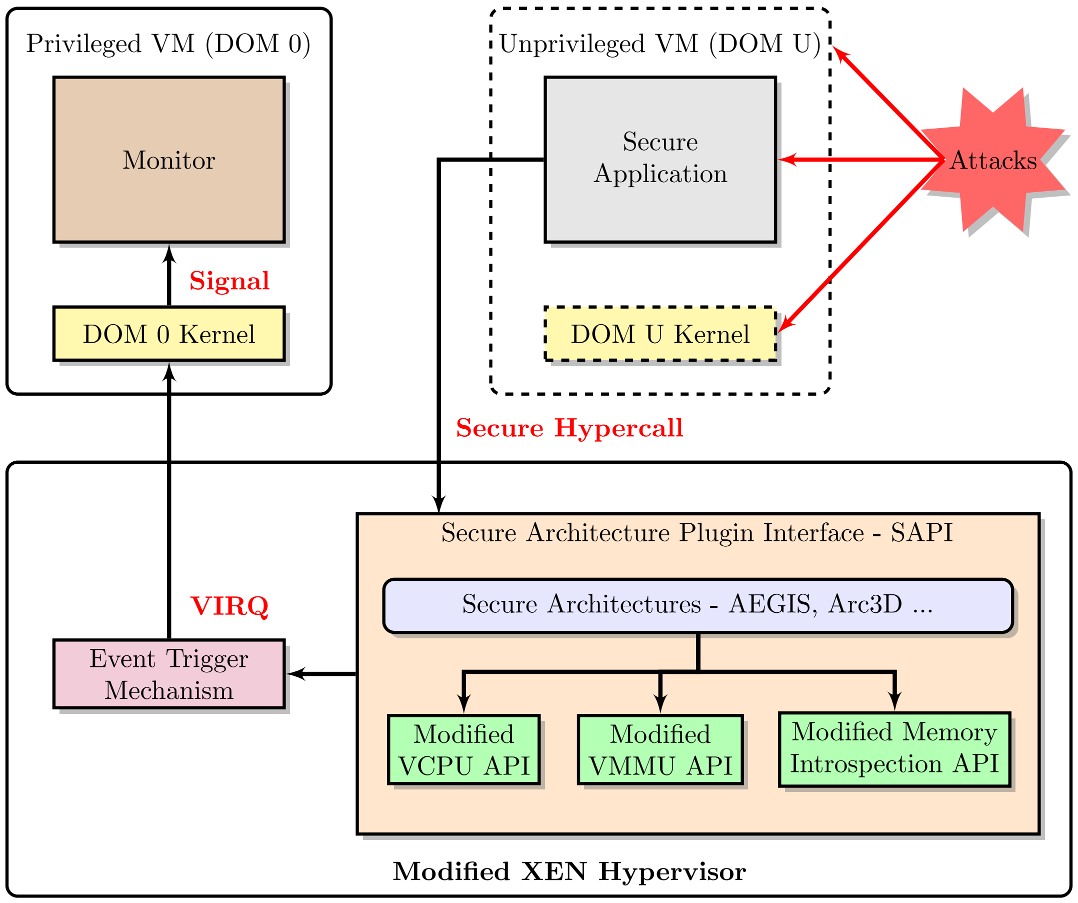


FIGURE 2.2. SecHYPE Framework

2.3.3. CTrust Framework

The intuition behind the CTrust architecture is to deploy the SecHYPE framework in a cloud environment to provide security and root of trust to the applications running in the cloud. Figure 2.3 shows the proposed CTrust architecture. This represents a prototype implementation of a private cloud built on top of the SecHYPE framework. In this architecture, the cloud is composed of a cluster of real computing machines known as the physical nodes. The nodes could be easily added to the cloud based on its load. Each node is connected to each other via a physical network. The CTrust architecture is implemented on XCP 1.0 [72] that contains XEN hypervisor version 3.4.x and CentOS with 2.6.32.x Linux kernel. Since the SecHYPE framework is developed by modifying the XEN hypervisor, it is possible to replace the original XEN hypervisor in XCP 1.0 with the SecHYPE framework.

The CTrust architecture is implemented with one HVM unprivileged VM. A HVM domain is a type of XEN VM which is capable of running unmodified OS (e.g. Windows). The HVM domain has been chosen to implement CTrust for two reasons. The first reason was to present a solution which doesn't require changes to the OS. The second reason is that the HVM guests have great paging support available in XEN API. However, it is important to stress that the same solution would be easier to port to PV domains also. The primary reason for this is that a PV domain runs a modified OS which is aware of the changes done by the underlying VMM.

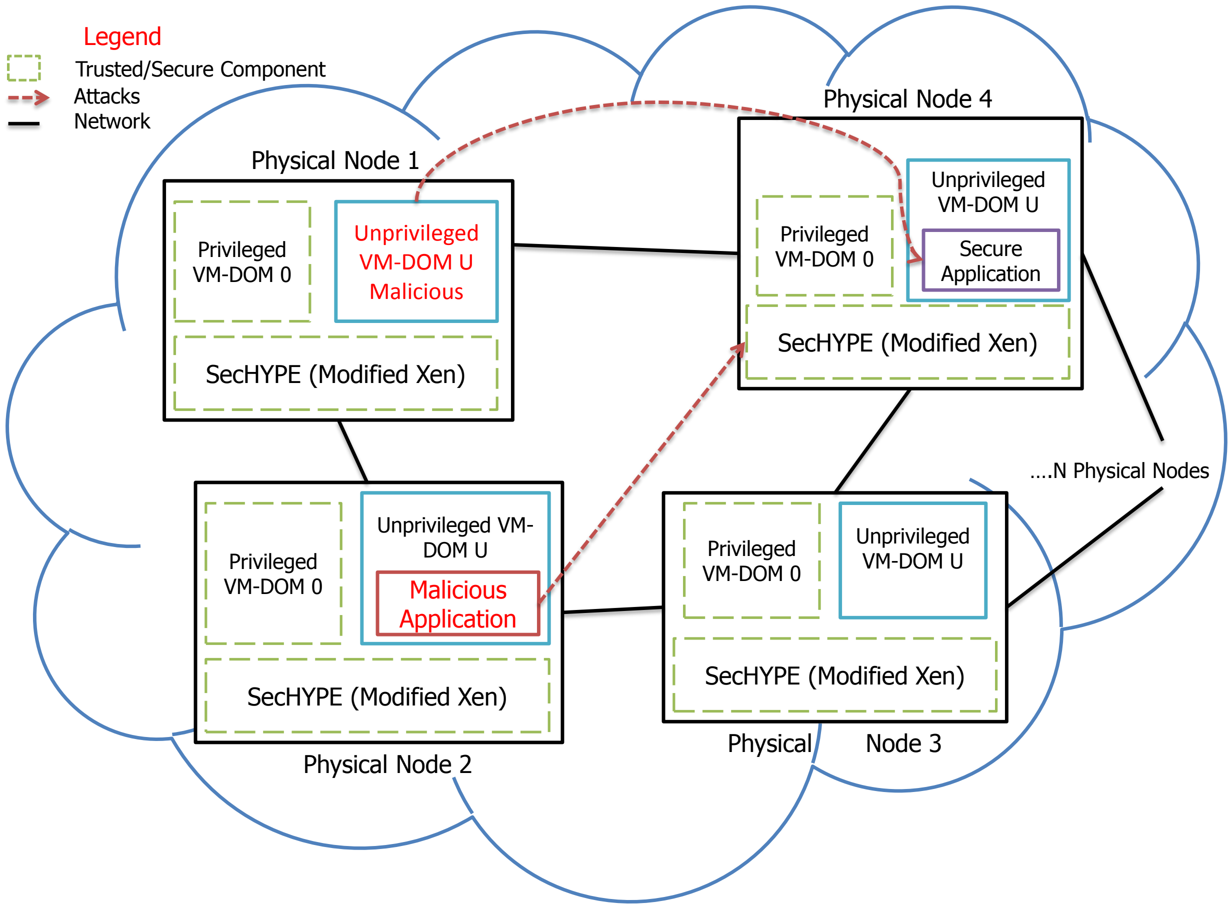


FIGURE 2.3. CTrust Architecture

A secure application can be under an attack from multiple avenues in a cloud environment. The attacks can be (but are not limited to) network based attacks, attacks from an external malicious VM or even parent VM and even attacks from applications running in the same VM. The secure hardware architecture inside the SecHYPE framework is responsible for protecting against these attacks. The security strength of the hardware architecture determines the overall security of the cloud platform. If an attack is beyond the security realms of the hardware architecture, then that particular attack would succeed. Hence it is the decision of the cloud provider to determine the level of security desired and to choose the most suitable hardware architecture. There are several advantages of the CTrust architecture: it does propose and enforce a new security mechanism, but instead empowers the cloud provider with the flexibility of choosing the most favorable security mechanism. It enables the cloud vendor to provide security assurance to its clients, thus honoring the service level agreements (SLAs). And finally, a client application can utilize the security mechanisms exposed by this architecture and achieve security during the epoch of application execution.

2.4. vBASE Prototype Implementation

This section presents a prototype implementation of the vBASE framework. The overall process is described in Algorithm 1. It is important to emphasize that the prototype implemented is just a small instance of framework. This prototype serves as a proof of concept to prove the effectiveness of the framework. This implementation includes a very simple secure hardware architecture functionality and not a fully functional architecture. The implementation is not as powerful as the framework itself as its primary goal is to show the readers how the various components of vBASE work with each other during secure execution. The secure application runs in a HVM DOM U and once it begins execution, it allocates a chunk of memory for secure data/variables. Later the secure process allocates memory from this secure chunk, to the secure data/variables. The implemented secure architecture protects this chunk of memory from any memory based attacks. To achieve this, the secure process copies the virtual address of the secure chunk of memory and passes it to the hypervisor. The size of the secure chunk of memory is aligned with that of the memory pages. It is relatively easier to mark the pages read-only than making individual memory locations read-only.

Algorithm 1 Secure Process Execution

```
1: Secure application/process begins execution.
2: It allocates secure memory buffer.
3: Transmit virtual address (VA) of secure memory buffer through a ring3_hypercall.
4: Receive VA in hypercall handler in the hypervisor.
5: Compute guest frame number (GFN) for the VA.
6: Compute machine frame number (MFN) for the GFN.
7: MFN is marked read-only.
8: Dispatch VIRQ to the DOM 0 kernel.
9: Dispatch a SIGNAL from DOM 0 kernel to the Monitor program.
10: Monitor prints the events to user.
```

The primary advantage of vBASE implementation is that, it eliminates the OS in making the secure process's memory read-only. To achieve this, it uses the CPUID instruction. All other instructions except CPUID are short circuited to the OS's interrupt table and do not directly trap in to the VMM. The CPUID, on the other hand, traps directly into the VMM. This instruction was originally intended to report processors features to ring-3 software applications. It takes various parameters and reports different features of the processor depending on those parameters. A custom parameter: 0x92, has been added to the CPUID instruction. This parameter is passed in the eax register and takes the virtual address of the secure memory in the ebx register. The secure application code containing the CPUID instruction is trapped in the VMM. The VMM recognizes the operand 0x92 and passes the virtual address to the SAPI component. In the SAPI component, the virtual address is translated into Guest Frame Number (GFN) and eventually into Machine Frame Number (MFN). This MFN is then made read-only and a Virtual Interrupt (VIRQ) is sent to the DOM 0. The VIRQ is handled by the kernel module in the DOM 0 and it notifies the Monitor through a Signal. The Monitor prints the confirmation message to the user and the secure process resumes its execution. Since the secure memory pages are protected by the VMM, even overwriting requests from OS will be ignored. The xm daemon will report

```
char secure_mem[PAGE_SIZE] secure_memory __attribute__((
   __aligned__(PAGE_SIZE)));
```

Listing 2.1. Secure Memory Allocation

```
long * secret_key=(long *) secret_mem;
*secret_key = 654321;
```

Listing 2.2. Secret Key Linked to Secure Memory

if any such attempts to overwrite the read-only (secure) pages is done.

We now present the source code implementation of vBASE. In the listing 2.1, a chunk of memory: secure_mem of PAGE_SIZE is allocated and the beginning of the chunk of the memory is aligned to beginning of a page.

In listing 2.2 we have declared a long pointer - secret_key and type casted the char buffer pointer to an long pointer (secret_mem). The secret_key is assigned the address of the secure_memory. Now the contents of the address contained in secret_key will be stored in the chunk of memory allocated in listing 2.1.

The address of the secret key is to be protected. To protect it we will make it read only. In listing 2.3, we pass the secret key to the hypervisor using ring3_hypercall. The ring3_hypercall is an assembly macro which takes the virtual address of the secure memory and a command integer as parameters. In the macro these parameters are passed as operands for the CPUID instruction. The CPUID instruction will then trap into the VMM. Along with the virtual address we send a command to the SAPI module in the hypervisor which decides the action to be performed on the virtual address. Initially we will send a PROTECT command. This will protect the virtual address by making the corresponding Machine Frame Number read-only. Originally XEN allows hypercalls to be done only through privileged interface in the OS. But we have solved this problem by using CPUID instruction. We call this mechanism as ring3_hypercall, which stands for hypercalls made from ring3 of DOM U.

Hypercalls are trapped into the VMM, thus giving VMM control. The blocks of VMM

```
int ring3_hypercall(unsigned long gva, int cmd)
{
        int ret;
   __asm__ __volatile__(
        "cpuid"
        : "=a"(ret)
        : "a"(0x92), "b"(gva), "c"(cmd)
        :"cc", "edx"
        );
        return ret;
}
unsigned long gva=(unsigned long) secret_key;
ret = ring3_hypercall(gva, PROTECT);
```

Listing 2.3. Pass the virtual address to the VMM

where hypercalls are handled is called as hypercall handler. The Guest Virtual Address (GVA) is received in the hypercall handler and Machine Frame Number (MFN) is computed which is the hardware machine's RAM frame number. The function that computes MFN needs Guest Frame Number (GFN) as the argument. So first we computed GFN using paging_gva_to_gfn() function. This GFN is then passed on to the gmfn_to_mfn where MFN is calculated. This MFN is marked read-only by the XEN's in built function. This function requires previous type (read-only, write-able) of the MFN. Hence we first invoke a function which returns the old type of the MFN. The MFN is passed along with the old type and new type (p2m_ram_ro). This will update the MFN type in XEN's tables. Any further requests to overwrite the pages identified by the MFN are dropped by XEN. Attempts for trying to write to the read-only MFN are reported to DOM 0 through Xend. This is shown in listing 2.4 below.

In listing 2.5 the code sends a global VIRQ to the DOM 0. In XEN VIRQ's are

```
struct vcpu *v=current;
struct domain *d=v->domain;

p2m_type_t old_type;
mfn_t mfn;

gfn=paging_gva_to_gfn(current, gva, &pfec)

mfn=gfn_to_mfn(d, gfn, &old_type);
p2m_change_type(d, gfn, ot,p2m_ram_ro);
```

Listing 2.4. Translate GVA to MFN and mark MFN read-only

```
send_guest_global_virq(dom0, VIRQ_VBASE);

bind_virq_to_irqhandler(VIRQ_VBASE,0,vbase_handler,NULL,NULL,
   0);
```

Listing 2.5. Dispatch VIRQ from XEN

handled by the guests as interrupts. So we need to bind the VIRQ to an IRQ Handler. After the VIRQ is populated from the hypercall handler, the DOM 0 is resumed and an interrupt (dynamically bound by the kernel) is issued. This will trap the control into the IRQ handler bound earlier.

In listing 2.6 the task_structure of the Monitor process is found using find_task_by_pid function provided by the kernel (note that pid of the monitor can be sent through a system call or by writing to kernel filesystems). After finding the task_struct a signal is issued to monitor process. This will notify that the virtual address of the secure process has been made read-only.

Therefore it is evident from the prototype implementation, that the vBASE frame-

```
find_task_by_pid_type(PIDTYPE_PID, monitor_pid);
send_sig_info(SIG_VBASE, &info, task_structure);
```

Listing 2.6. Dispatch a SIGNAL to the Monitor

work detects any unwanted modification to the secure application memory. These modifications can result from malicious applications, VMs, vulnerable OS and network channels. The plugged-in secure hardware architecture, for implementation purposes, employs an encryption and integrity verification mechanism. This allows vBASE to detect malicious modifications and eventually discard them.

2.5. vBASE Analysis

In this Section, we present a formal analysis of the vBASE framework in terms of its security and performance.

2.5.1. Security Analysis

The vBASE framework is designed and implemented in XEN hypervisor. Therefore its architecture conforms to the overall structure of XEN. Nonetheless, the security mechanisms available in vBASE are developed using software and hardware utilities that are universal to any virtualization software. In XEN, the hypervisor layer and the Domain 0 (DOM 0) are at higher privilege level as compared to Domain U (DOM U), where user applications are executed. The vBASE framework exploits this design by placing the Monitor and Controller components inside the privileged DOM 0. The application that needs to be protected is run inside the DOM U. The first assumption in vBASE is that there are no preexisting vulnerabilities in the XEN hypervisor itself. The XEN hypervisor is shipped with a modified Linux kernel that is by default loaded at system. It is assumed that this Linux kernel does contain vulnerabilities that can be exploited by potential attackers. Though, in such scenarios it is reasonable to assume that a TPM can be used to verify the integrity of the XEN hypervisor. With regards to the execution frameworks – SecHYPE and CTrust, it is reasonably considered that the cloud provider itself is not malicious. Also, it is expected that the secure application that needs to be protected, is itself not malicious. It is important to emphasize that vBASE by its own does not propose any secure architecture. It is a reconfigurable framework to test the security of a hardware architecture and then export the most suitable hardware architecture into the cloud to provide root of trust to the applications running in the cloud.

Finally it is also essential to discuss about cryptographic key management in the vBASE framework. This may not be important in the testing framework, but certainly becomes crucial in the execution framework. The secure hardware architectures integrated within the SecHYPE framework typically include an Encryption and a Memory Integrity Verification security mechanisms to provide confidentiality and preserver integrity of the data. These mechanisms require cryptographic keys for computing cyphertexts. These keys must be kept secret and distributed to only trusted entities as leaking of keys would compromise security of the entire system. Thus the management of such keys becomes very important. The vBASE framework does not handle the process of cryptographic key management and instead relies on the hardware architecture vendor to define this process. There are multiple recommendations and best practices to securely and efficiently manage keys in cryptography. These can be found in NIST recommendations on key management [6].

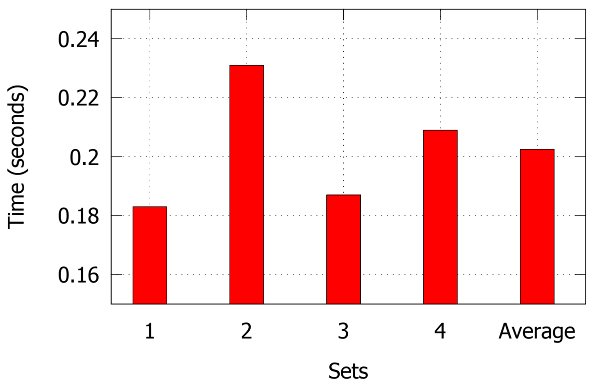
2.5.2. Performance Analysis

The current computing frameworks have been predominantly designed around the principle of resource sharing and performance. Be it the virtual memory in traditional computers, or a virtualized desktops or a cloud computing system, the primary goal is to improve resource utilization and enhance the performance of the system. However, security and performance are two sides of a coin and security implementations may often lead to severe performance bottlenecks. Since, the sole purpose of the vBASE testing framework is to test hardware security architectures, analyzing its performance is of no use. However, performance analysis becomes imperative in vBASE execution frameworks (SecHYPE and CTrust) as their main focus is cloud computing platforms. In cloud systems, performance is very important as multiple VMs and application are competing for shared hardware resources. The security implementations in cloud computing can introduce severe performance trade-offs in the system. Furthermore, the backbone of vBASE security implementation is the embedded hardware secure architecture. A vital facet of hardware security mechanisms is that they require modifications to the internal design of the processor, so as to add new hardware components. This additional hardware consumes logic, memory and clock resources, thereby impacting the performance of the system. Thus analyzing the performance of vBASE becomes utmost important.

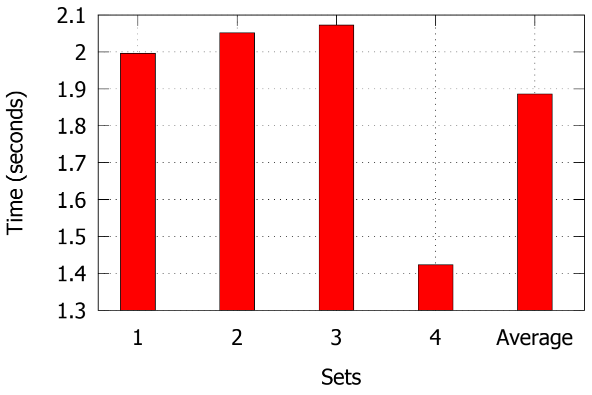
The performance of the vBASE framework is analyzed by measuring the total time it takes to complete the secure communication channel. This channel is composed of a secure hypercall, Event Trigger Mechanism and a SIGNAL. The communication channel contains the following intermediary steps.

- Secure process (in DOM U) invokes the hypercall.
- Hypercall gets trapped in the VMM.
- Hypercall handler in VMM issues a VIRQ.
- The VIRQ gets trapped in the DOM 0 kernel.
- VIRQ handler in DOM 0 issues a SIGNAL to Monitor in DOM 0 userspace.
- Monitor receives and prints appropriate notification to user.
- Control goes back to DOM 0 kernel.
- Control goes back to VMM.
- Control gets back to DOM U secure process.

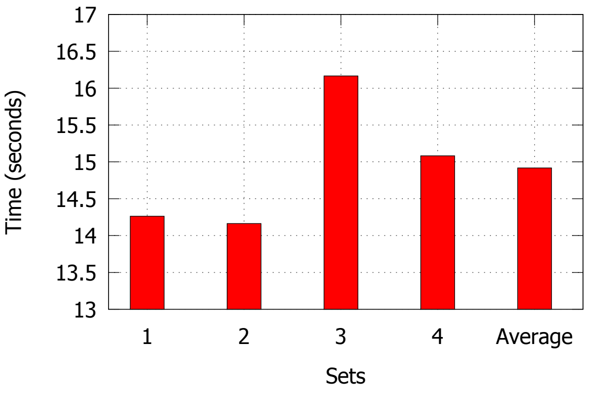
For each of the steps mentioned above, time has been measured from user-space, by using the time utility provided by the Linux kernel. In reality, multiple VMs run simultaneously in a cloud system. Each VM is in-turn capable of running multiple applications that may require secure execution. Hence the secure execution channel may be invoked several times at any given instant of time. This may introduce unwanted and unavoidable latency in the system. To simulate the effect of simultaneous secure channel invocations, the performance of the vBASE is measured for three values: 10,000, 100,000 and 1,000,000 invocations. These invocations are introduced at the same time and the corresponding time required to serve these requests is measured.



(A) for Invocations - 10,000



(B) for Invocations - 100,000



(C) for Invocations - 1,000,000

FIGURE 2.4. Time required for Secure Channel Invocations

The Figures 2.4a 2.4b 2.4c show the total time required for 10,000, 100,000 and

1,000,000 secure channel invocations respectively. For each type, four sets of reading are presented along with an average reading. The average reading for 10,000 invocations is approximately 0.2 seconds, while for 100,000 invocations is approximately 1.9 seconds and finally for 1,000,000 invocations is approximately 15 seconds. The measured time increases at lower rate from 10,000 invocations to 100,000 invocations but rises steeply from 100,000 invocations to 1,000,000 invocations. These readings are further used to estimate the average amount of time it takes to serve a single secure channel. The average time is 20 μs (Average = 0.2/10,000) during 10,000 invocations, 19 μs during 100,000 invocations and 15 μs during 1,000,000 invocations. This indicates that the performance overhead imposed by the security mechanisms implemented in vBASE is significantly low and hence it does not degrade the overall performance of the cloud system.

2.6. Previous Related Research

This Section provides a detailed description of the previous related research to the vBASE project. This literature survey is segregated in three categories - virtualization-based security approaches, cloud-based security security approaches and software security approaches.

2.6.1. Virtualization-Based Security Approaches

A lot of research has been focused towards the design and implementation of new and secure hypervisors. Some of these solutions are described in this section.

Overshadow [73] is a virtual machine based system that protects the privacy and integrity of application data even in an event of total OS compromise. This approach is designed and built on the concept of multi-shadowing — a mechanism that presents different views of physical memory, depending on the context performing the access. In a virtualized system, there are 2 levels of memory translations. A VMM/hypervisor maps the machine page number (MPN) to a guest physical page number (GPPN). This gives a VM an illusion that the entire memory is available for it to use. The OS in each VM translates each GPPN to a guest virtual page number (GVPN) to support multiple application execution. In this case, multi-shadowing is a mechanism that supports context dependent, one-to-many GPPN-to-MPN mappings. Memory cloaking combines multi-shadowing with encryption, thus presenting different views of memory plaintext and encrypted to different guest contexts. Overshadow provides and efficient and reliable solution for securing applications against a malicious guest OS. Its primary disadvantage is that all its security mechanisms are built around assuring application security against malicious guest OS. However, there is no consideration made for attacks originating from malicious application running in the same VM.

TrustVisor [40] is a specialty hypervisor that furnishes code/data integrity and secrecy for applications, permitting certain parts of an application to run fortified with security, by being in isolation, from OS and DMA capable devices. These security sensitive codeblocks are known and names as Pieces of application Logic (PALs). What serves to safeguard the memory, are 3 basic operating modes: Host, legacy, and secure. Host mode allows for TrustVisor code execution at the highest level of privileges. Host mode subsequently supports the two other modes of legacy guest and secure guest. In legacy guest mode, an OS and its applications can execute free from the adjacency and collaboration of TrustVisor's presence. This is not so far for secure guest mode, for which a PAL executes separately from legacy OS and its applications, eliminating any party's presence. TrustVisor relies upon available hardware virtualization support to offer memory isolation and DMA protection for each PAL. The authors have devised a mechanism called the TrustVisor Root of Trust for Measurement (TRTM) for applicable conductings of PALs. The TRTM is actualized by including a TrustVisor-managed, software micro TPM (μTPM) instance to be associated with each PAL. The TRTM and its microchip safekeeps sensitive code at a very fine granularity and also has a very small code base (approximately 6K lines of code) that makes its verification realistically possible and duplicable. TrustVisor can also establish the existence of execution running in isolation to an external entity. This specialized hypervisor imposes less than 7% overhead on the legacy OS and its applications, in most typical conditions.

sHype [54] is a secure hypervisor architecture, independent of an OS, to control information flow between the OS and the shared hardware platform. This approach is an extension of full-isolation hypervisor with security mechanisms that enable controlled resource sharing among VMs. it deploys a reference monitor interface inside the hypervisor to enforce information flow constraints between partitions. This reference monitor is implemented by using Enforcement hooks. These hooks retrieve access control decisions from the access control module (ACM). The ACM applies access rules based on security information stored in security labels attached to logical partitions. The ACM stores all the security policy information in the hypervisor and supports efficient policy management through a privileged H_Security hypervisor call. sHype is build on top of vHype, which utilizes CPU protection rings to ensure that the partitions cannot execute privileged instructions. The protection mechanisms in sHype only talk about access control and privilege separation. It has been shown in the past that only employing these mechanisms are not adequate to protect against a more powerful and resourceful attack. The information about the logical partitions is reported to the ACM by the means of a secure H_Security hypervisor call. Traditional hypercall mechanisms depend on guest OS functions and so the authors have assumed the guest OS to trustworthy. There have been a lot of research depicting the vulnerabilities in commodity OS due to their bulky codebase and hence these OS cannot be considered untrustworthy.

VMInsight [37] is a hardware-virtualization-based security monitor system, which can supply load-time and run-time monitoring for processes and can intercept system calls and process behaviors by observing changes in the VM CPU register. VMInsight is transparent to the softwares and OS running in the VM, because it is implemented in the hypervisor. VMInsight has 3 modules: information obtaining module, monitoring analysis module and information display module. The information obtaining, monitoring analysis, and information demonstration. The information obtaining module, being within the KVM module, possesses 3 functions: register reading, memory reading and PageFault. The monitoring analysis module is managed by the Linux system driver module method and able to load and unload per requirements. It obtains two callback interfaces of module: writeCr3Handle() and pageFaultHandle(). Finally the information demonstration module provides ioctl interface through the monitoring analysis module, obtains the process security monitoring information of virtual machine, saves to the security event database and finally demonstrates the information using the WEB method. It also collects user control information and sends its feedback to the monitoring analysis module, to handle the operation of processes. What the experimental results reveal is that the performance overhead of VMInsight comes at less than 10%, which makes it feasible to join into application for third-party security monitoring. The drawback to this approach is that the monitoring process functions at the granularity of a VM but yet not so for specific processes contained inside the VM. Consequently, the monitoring that would be needed could be an extortionately unwieldy amount. This bulkiness can markedly degrade the performance when more than one VM is in operation on just one sole hardware platform. The monitoring module is entirely constructed inside the kernel and relies on the kernel drivers to garner monitoring information. Inasmuch, the kernel becomes integral to the TCB, which can open back doors for determined attackers to exploit current security vulnerabilities in kernel code.

VASP [42] is a hypervisor based monitor that allows a trusted execution environment to monitor various malicious behaviors in the OS. This approach leverages the features provided by x86 hardware virtualization and self-transparency technology to provide an unified security protection to unmodified OS such as Linux and Windows. The VASP hypervisor Layer exports 3 sets of interfaces - memory management interface, trap register interface and debugging interface. Memory management interface is used to realize the memory management of VASP itself. Trap register interface supports the extension usage of VASP and provides configurable interceptions. And finally the debugging interface is used for dynamic analysis when developing the hypervisor. The monitoring mechanism in VASP is implemented in the form of kernel module that can execute any special instruction with highest privileges. Thus it relies on the kernel which may introduce some security vulnerabilities. To close this vulnerability, the authors have implemented memory self-transparency in VASP. In this, a clone of OS page table is made and provided to the hypervisor while executing.

Also, some spare memory space is used as pseudo memory space of the hypervisor and the physical address of hypervisor is modified. As a result, the application or kernel in the guest operating system can only access the physical address of pseudo memory when using the virtual address of hypervisor, but hypervisor can access its real physical address using the same virtual address when it executes in the root mode. A significant disadvantage of VASP is that it provides a monitoring mechanism to protect against an entire OS. This in itself is a rigorous task as every single access that goes through the OS has to be monitored. And even though the performance analysis in the paper, does not reflect this, the performance will degrade exponentially when multiple VMs run on the same hypervisor. Moreover since the monitoring mechanism is implemented as kernel module, it has to be pre-included in the VM image.

Finally authors Dwoskin et. al [14] have proposed a testing framework, which provides APIs for monitoring hardware and software events in the system under test. The testing framework is divided into two systems - the System Under Test (SUT) and the testing System (TS). The SUT is meant to behave as closely as possible to a real system which has the new security architecture. It runs a full commodity guest OS, which is vulnerable to attack and is untrusted. The TS machine simulates the attacker, who is trying to violate the security properties of the SUT. It is kept as a separate virtual machine so that the TS Controller can be outside of the SUT to launch hardware attacks. Attack Scripts reside on the TS and specify how particular attacks are executed on the SUT. Based on the requirements, the attack model and attack scripts can be elegantly modified.

2.6.2. Cloud-Based Security Approaches

This section emphasizes on research ideas that present consolidate frameworks/architectures to provide security in cloud computing environments.

NoHype [31] proposes the approach of removing the virtualization layer while retaining only its key features. The multi-tenant design of virtualization technology, though has several advantages, is a serious security concern in Cloud Computing. A motivated attacker can attack the virtual machines or the virtualization software and if successful it will compromise the confidentiality and integrity of the user applications and data in the cloud. In NoHype framework, only one VM is run on each processor core. This removes the need for active VM scheduling done by the hypervisor and protects against software cache based side channel attacks. Each guest OS has a view of memory where it has a dedicated and guaranteed fraction of physical memory. The guest operating system is assigned it own physical device and given direct access to it. The Ethernet switches in the data center network should perform the switching and security functions, not a software switch in the virtualization layer. Some of the drawbacks of NoHype are that it allocates each VM to a core. Such hard allocation of server resources can cause performance overheads and can be ineffective. The hardware memory is partitioned and each guest OS has the view of the entire memory. Here the authors are assuming that the OS itself is not malicious. Whereas it has been proved over and over again that OS can have a lot of vulnerabilities. In such cases , attackers could carry out memory based attacks thus compromising the confidentiality and integrity of data of all other VMs. The biggest disadvantage is that the OS is root of trust in this architecture.

CloudVisor [76] - In this paper, the authors propose a transparent, backward-compatible approach that protects the privacy and integrity of customer's VMs on commodity virtualized infrastructures, even facing a total compromise of the VMM and the management VM. The key feature of this approach is the separation of the resource management from security protection in the virtualization layer. A tiny security monitor is introduced underneath the commodity VMM using nested virtualization that provides protection to the hosted VMs. CloudVisor is an extremely lightweight security monitor that runs at the host mode i.e. highest privilege level in the system. The VMM is deprivileged to a lower privilege level. CloudVisor enforces the isolation and protection of resources used by each guest VM and ensures the isolation among the VMM and its guest VMs. The traditional virtualization functions such as resource management, VM management and scheduling etc. are still done by the VMM. CloudVisor transparently monitors how the VMM and the VMs use hardware resources to enforce protection and isolation of resources used by each guest VM. CloudVisor relies on TPM and trusted execution design for ensuring the security of a cloud system. This is a significant drawback as the TPM cannot protect against many of attacks that threaten privacy of users. Also the TPM does not reduce the threat from spywares that could monitor and profile user's activities, such as browsing habits, and send them to a remote party. It is also vulnerable to power analysis which can break tamper-evident property of the TPM by being able to extract information from protected storage without being detected. In CloudVisor, the security of the VMs and itself is tested at start up by comparing its current image to a know secure image stored in the TPM. This still does not protect from vulnerabilities and attacks introduced during the running of VM and CloudVisor. Since the VMM is now deprivileged but is still placed at a higher privilege level than the guest VMs, this introduces another level of paging within CloudVisor. This adds to complexity of the system.

CloudSec [27] is a virtualization-aware monitoring appliance that supplies active, transparent and real-time security monitoring for hosted VMs in the IaaS model. It relies on virtual machine introspection (VMI) techniques to supervise the physical memory of guest VMs at the hypervisor level. CloudSec externally reconstructs a high-level semantic picture of the running OS kernel data structures instances for the monitored VM's OS. The predominant concept here is to map with accuracy the underlying hardware memory layout as against the OS kernel structure. The VMI layer in CloudSec holds two components: the back-end component that enables the hypervisor to acquire control over the hosted VMs in order to, if necessary, delay any access to the physical memory and CPU, and the front-end component, which is comprised of APIs that enable attainment of data about the monitored VM's running OS from the hypervisor and the regulation of accesses to the physical memory and CPU registers. A proof-of-concept prototype has been developed using VMsafe libraries on a VMware ESX platform. The primary preoccupation centers on the performance of CloudSec. Reconstructing the OS high level semantic view externally of all the VMs running in the cloud daunts as an especially costly process, time-intensive and performance-burdened, notwithstanding that a large pubic cloud could be occupied by thousands of VMs and thus the scalability of CloudSec at that dominates as an issue. No clarification exists on how this framework will be able to assure that its own execution is a secure one. If ever should occur an attack that succeeds on the CloudSec itself, the whole VMI will be rendered untrustworthy.

In [39], Lui et. al present a framework where a measurement module (MM) is added in each guest VM. It measures every running executable in that VM. The MM transfers new measurement values to the trusted VM via standard inter-VM communication mechanisms. The trusted VM stores those values in sequence in a measurement table (MT). At the same time, the system extends these measurement values into a specified platform configuration register (PCR). To ensure the measurement process's trustworthiness, a memory watcher (MW) module has been added to the VMM. The assumption that the guest OS is trustworthy is the biggest drawback on this framework. A lot of research has shown that the commodity OS can extremely vulnerable to attacks due to their code size and thus highly untrustworthy. The security measurement is only performed statically during boot up time. This does not account for attacks introduced while the executable is running. The inter-VM communication channel used for storing the executable measurement relies on kernel functions, which may be compromised themselves. The verification process only involves hash verification. This only guarantees the integrity of the executable. There is no verification like encryption to ensure the confidentiality of the executable is also intact.

2.6.3. Software Security Approaches

Some researchers are motivated to propose new software security solutions that deal with application level and OS level changes to provide trustworthy and secure execution in cloud computing. For instance, Ashish Thakwani has proposed dfork [1] to provide process-level isolation using virtualization. dfork is a clone of posix fork where a separate kernel and file-system is allocated to every new process to isolate its operations and interactions from other processes. This method gives an ability to review the changes done by the application before committing to the underlying hardware, thus giving control to accept, isolate or discard some of these changes. Rutkowska et al. propose a new (modified Linux) operating system called Qubes OS [29]. It is very similar to dfork in a way that every process that is being secured gets its own kernel and environment in a virtualized container.

Qubes OS provides security by isolating application execution in its own virtual machine. Finally, Microsoft OS research has proposed an approach called Singularity [26] to solve the process isolation problem by introducing a new concept of software assisted process isolation as opposed to widely accepted, de-facto, standard of hardware assisted process isolation.

2.7. Conclusion

Advances in computing power and constant inter-connectivity have made modern day attackers extremely powerful. Softwares are increasingly vulnerable to attacks originating from multiple sources and growing into various forms. Therefore, Software Protection is critical n this era of computing. The need for hardware assisted software security stems from the ever expanding worth of softwares. In addition to this, the security mechanisms implemented in softwares are inadequate and untrustworthy. Trust can be removed from the softwares by properly leveraging the hardware security mechanisms. However, the wide scale adoption of hardware security mechanisms has been impeded due to the emphasis given to performance over security during the design phase.

In an effort to answer this question, we have proposed the virtualization based secure (vBASE) framework. In the vBASE testing framework, we have successfully demonstrated how hardware secure architectures can be realized and tested in an efficient manner using the data structures provided by the virtualization technology. The mechanisms implemented in vBASE removes OS from the Trusted Computing Base (TCB), which has proved to be a big challenge for years.

We have extended vBASE to an execution framework. Here we present the Cloud Trust architecture - CTrust, which is developed around a fundamental security paradigm of root of trust. The intuition behind this approach is to strengthen the security of the virtualization software that forms the backbone of any cloud platform. Thus we have proposed SecHYPE, a secure hypervisor framework that contains enhanced security implementations. The SecHYPE is deployed in CTrust architecture to root the trust of user applications in its security implementations. The SecHYPE framework employs secure architectures that are proven to achieve high level of security. This framework is flexible to incorporate any secure architecture as desired by the cloud provider. We have also presented performance evaluation that proves that the time latency introduced by a single secure channel is significantly low in the range of 15 μs to 20 μs.

CHAPTER 3

MEMORY INTEGRITY VERIFICATION IN EMBEDDED SYSTEMS [1]

3.1. Introduction

An embedded system, in contrast to a general purpose computing system, is a dedicated system designed to serve a specific task within a larger system. They are high performance systems, flexible enough to perform a variety of computing tasks in a cost effective manner. Embedded systems have now evolved into complex systems. The modern day embedded devices are often small, portable and highly interconnected. They are capable of tracking, storing information and even transmitting essential data over the internet. These characteristics have enabled embedded systems to pervade all facets of human life. They are being used everywhere from home media systems, portable players, smart phones, automobiles, embedded medical devices to mission critical defense systems. These systems are empowered to access, store and communicate sensitive information. This information may also include confidential personal data including secret passwords, credit card information, bank account details and so forth. Thus security has become very important in embedded systems.

To make matters worse, the operating environment of embedded systems allow the adversary to have complete control of the computing node for example acquiring supervisory privileges along with complete physical and architectural object observational capabilities. The design phase of embedded systems often does not provide for the security axis thus increasing the security risks. Moreover, embedded systems are physically dispersed to public locations that are available to potential attackers. This makes them vulnerable to physical attacks. Though these systems are small and flexible, their design is complex which adds to their security issues. Despite these problems, embedded systems are deployed widely. Hence motivated attackers can exploit these vulnerabilities to extract confidential information from

[1]Parts of this chapter have been previously published, either in part or in full, from [48] with permission from Elsevier and [49] with permission from Springer.

these devices. For instance, Mobile and Smart Device Security Survey [43] conducted by Mocana Corporation in Spring 2011, revealed that 65% corporate personnel require a regular attention from their information technology staff for mobile and smart phone based device attacks. The Cyber Security Watch Survey [12] conducted by CSO, the U.S. Secret Service, the Software Engineering Institute CERT Program at Carnegie Mellon University and Deloitte in January 2011, disclosed that more than 58% of attacks are caused by outsiders i.e by unauthorized access to network systems and data, thus contributing to a staggering annual monetary losses of $123,000 per organization.

A security solution for embedded systems is the use of hardware secure architectures. To achieve this, the micro-architecture of the embedded processors must be modified to incorporate security mechanisms. These typically include (but are not limited to) the addition of an encryption mechanism to protect the confidentiality of the data, memory integrity verification mechanism (MIV) to preserve the integrity of data, new registers, cache memories and so forth. However, these security mechanisms are computationally intensive and often consume logic and timing resources of the processor, thereby degrading its performance. Embedded systems are highly resource constrained. Most of these devices are battery powered and it is essential to have minimal energy consumption to achieve high speed and performance. The excess energy consumed by the security mechanisms degrades their performance and becomes a critical issue. As a case study, Potlapally et. al [53] have proposed a framework to analyze the energy consumption of security mechanisms and protocols. Their work primarily focuses on investigating the impact of security processing on the battery-life constraint of an embedded system. For battery powered embedded systems, the biggest challenge is the trade-off between energy and performance due to security processing. As this trade-off increases, so also does the battery gap in embedded systems. The Tables 3.1 and 3.2 show the energy consumption in μJ/Byte for most commonly known and used encryption and hashing algorithms. Hashing algorithms are most commonly employed in secure processors to achieve memory integrity verification.

In order to provide security in embedded systems, it is essential to design energy

TABLE 3.1. Energy Consumption of Encryption Algorithms [53]

	Algorithms							
	DES	3DES	IDEA	CAST	AES	RC2	RC4	RC5
Energy Consumption (μJ/Byte)	2.08	6.04	1.47	1.21	1.73	3.93	0.79	0.89

TABLE 3.2. Energy Consumption of Hashing Algorithms [53]

	Algorithms					
	MD2	MD4	MD5	SHA	SHA1	HMAC
Energy Consumption (μJ/Byte)	4.12	0.52	0.59	0.75	0.76	1.16

efficient implementations of these security mechanisms and protocols. Thus their research becomes pivotal for addressing the challenges related to energy efficient security mechanisms in battery constrained embedded systems.

The rest of this chapter is organized as follow. Section 3.2 formally defines the memory integrity property. Section 3.3 describes the attacks on memory integrity followed by a detailed description on verification mechanisms in Section 3.4. Section 3.5 presents our proposed memory integrity verification framework. We then discuss the Related Previous Research in Section 3.6. Section 3.7 presents the Conclusion of this chapter.

3.2. Memory Integrity Property

The memory integrity property can be defined as follows. A processor communicates with memory $\mathcal{M}$. Memory $\mathcal{M}$ has two attributes, addresses A and contents V. It maintains associations between addresses and contents. A read of memory at address A denoted by $\mathcal{M}[R, A]$ returns the value associated with A. A write into memory address A of value V is denoted by $\mathcal{M}[W, A, V]$. A write of A with value V immediately followed by a read of address A must return value V. As memory reads and writes have a notion of time, the model needs to associate time T with reads and writes as $\mathcal{M}[R, A, T]$ and $\mathcal{M}[W, A, V, T]$.

Definition 1. *A read of address A at time T should return the value written to address A at time $T' < T$ such that no other write to A occurs between time T' and T. In other words: $\mathcal{M}[R, A, T] = V$ if an only if $\exists \mathcal{M}[W, A, V, T']$ for $T' < T$ and $\forall\, t \in [T'+1, T-1]$, $\nexists \mathcal{M}[W, A, *, t]$. At $T = 0$ the entire memory is initialized with value 0.*

3.3. Attacks on Memory Integrity

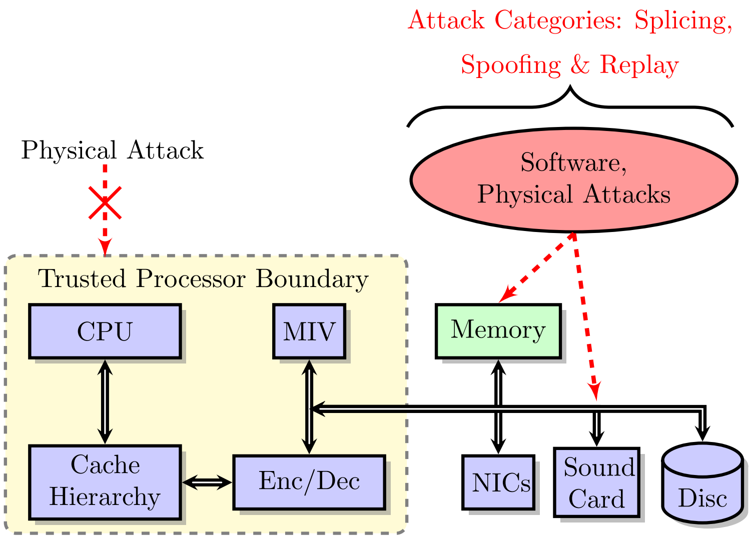


Figure 3.1. Attacks on Memory Integrity

An embedded device has to operate in an insecure environment and is thus vulnerable to a variety of attacks. The threat model is shown in Figure 3.1. The main hypothesis is that the processor chip is resistant to all physical attacks and is thus trusted. Sidechannel attacks are not taken into account in this research. As a result, it is considered that the on-chip registers and memories cannot be observed or tampered with by an adversary. The off-chip memory, peripherals and other devices are assumed to be untrustworthy. It is also assumed that the OS and its application are also untrustworthy and may be vulnerable to attacks. The attacks on an embedded system can either be software based attacks or physical attacks. These attacks can be categorized in 3 classes — splicing attack, spoofing attack and replay attack. In a splicing attack the adversary modifies the associations between the memory addresses and its values. For example, if value V_A is associated with address A and value $V_{A'}$ is associated with address A', a splicing attack would return the value $V_{A'}$ for a memory read corresponding to address A. In a spoofing attack the adversary modifies the value V_A to a random value V'_A. In a replay attack the adversary modifies the association between the value and the time. For example, if value V_A is associated with address A at time T and the value V^*_A is associated with address A at time T^*, and $T < T^*$, a replay attack would return the value V_A when a memory read is performed at time $t > T^*$.

3.4. Verification Mechanisms

In cryptography, a hash function – $\mathcal{H}$, is used to protect the integrity of a message. It takes in a input of arbitrary length and produces an output of fixed size. All hash functions possess the properties of pre-image resistance, second pre-image resistance and collision resistance. In pre-image resistance, given the hash (h) of a message (m) is known, it should be unfeasible to find the message. With second pre-image resistance, given a message (m_1), it should be unfeasible to find another message (m_2) such that $m_1! = m_2$ and $h(m_1) = h(m_2)$. Finally with collision resistance, the hash of two different messages m_1 and m_2 should never be the same. Some of the most commonly used hash functions today are SHA, SHA1, MD2, MD4, MD5 and HMAC etc.

To authenticate data, a sender creates a message authentication code (MAC) using a secret key K together with the message m as $MAC_m = \mathcal{H}(m||K)$. It sends both the message m and the corresponding MAC_m to the receiver. The receiver, which shares the secret key K with the sender, can verify the integrity of the message by recomputing the MAC. The security properties, collision resistance, pre-image resistance and second pre-image resistance of the hash function $\mathcal{H}$ ensures that any modification to message m or MAC_m go undetected with negligible probability. One solution for the memory integrity problem is to use message authentication codes. Processor generates a MAC $h_{V,A} = \mathcal{H}(V||A||K)$ for every memory write that stores a value V in address A. This MAC, $h_{V,A}$, is stored in the memory. On every memory read the processor computes the MAC $h'_{V,A}$ and verifies it against the MAC stored in the memory. This solution can protect memory integrity against splicing and spoofing attacks. A replay attack will still succeed as the MAC does not have any notion of time and hence the adversary could replay both the value and its corresponding MAC.

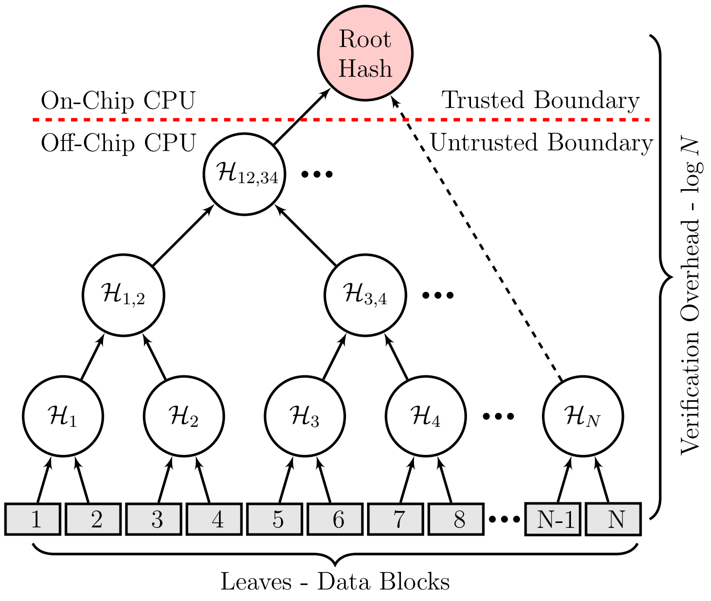


FIGURE 3.2. Merkle Hash Tree

Protecting memory against replay attacks requires the processor to have some memory about the recentness of the value. A Merkle hash tree, shown in Figure 3.2, provides an optimal solution for this problem, requiring least amount of on-chip memory. A Merkle hash tree of address range $[A, A+k]$ creates a tree of hashes with $k+1$ leaves corresponding to addresses $A, A+1, \ldots, A+k$. Any write to an address $A+i$ in this address space can modify all the hash values from the leaf node corresponding to $A+i$ upto the root of the tree. Any read from address $A+i$ needs to check the hash values along the path from the leaf node $A+i$ upto the root of the tree. The root of the tree is stored in the trusted processor storage, so that it cannot be tampered. All the other tree nodes along with the leaf nodes can be kept in the untrusted memory.

3.5. Proposed Memory Integrity Verification Framework

The motivation for proposing the memory integrity verification (MIV) framework is to simulate the behavior of a Merkle hash tree based memory integrity verification in a secure processor architecture. This is essential to determine the number of hash invocations

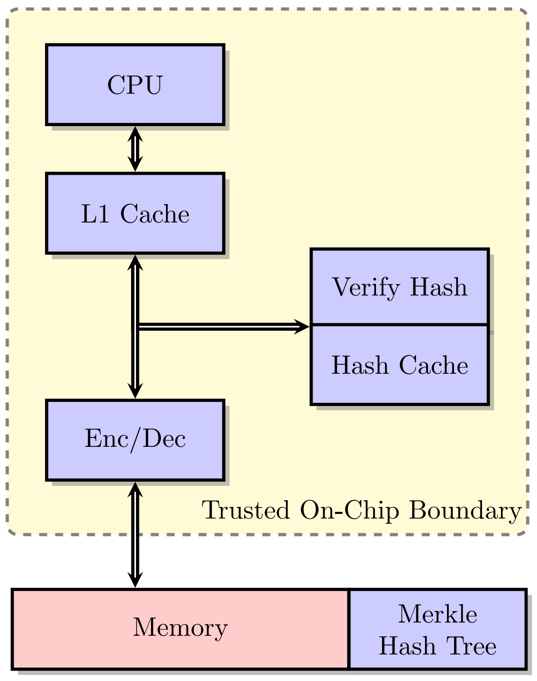
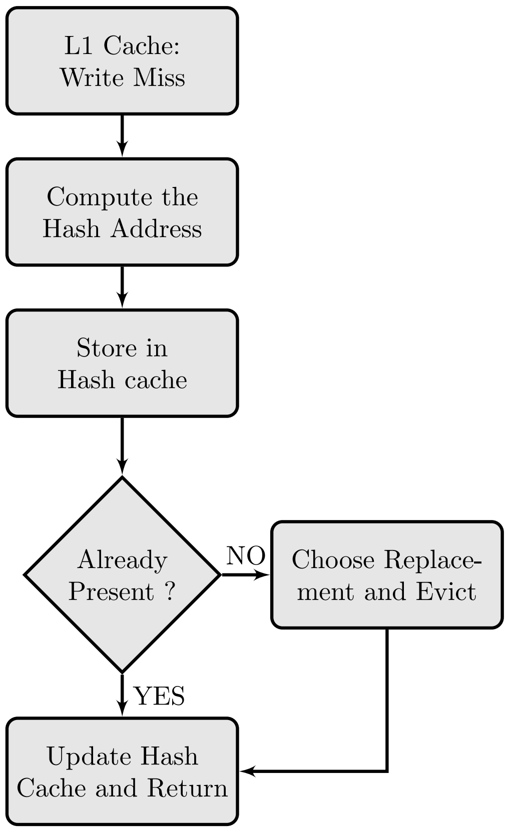


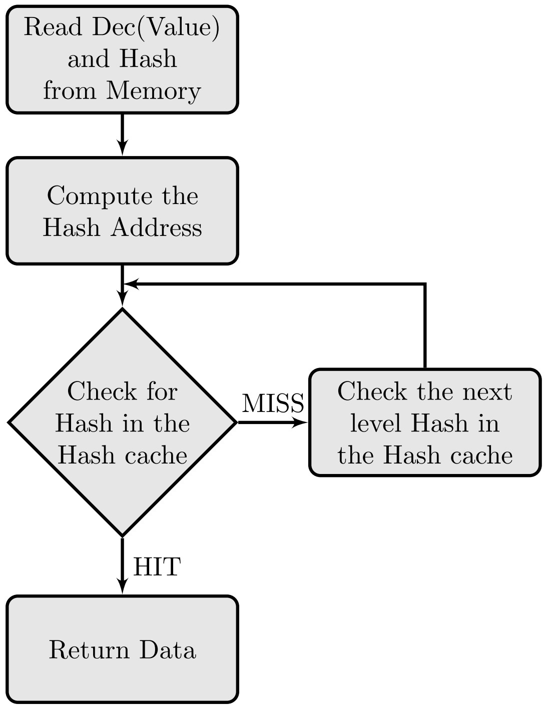
FIGURE 3.3. Memory Integrity Verification Framework

required per memory access. In this implementation, this process is being simulated by creating a custom hash cache with N levels to store the hash addresses. The configuration of hash cache is similar to the Level 1 Data cache configuration.

Figure 3.3 shows the working of memory integrity verification in a secure processor architecture. In the memory write operation, shown in Figure 3.4a, for every write miss in the level 1 data cache, the corresponding hash and hash address are calculated. This hash is stored in the hash cache in it corresponding hash address. If the hash address already exists in the cache, then it is updated or else a suitable replacement block is selected and evicted to store the hash. The data is then encrypted through the memory encryption mechanism and then stored to the off-chip untrusted memory. During a memory read operation, shown in Figure 3.4b, the data is first decrypted and the hash address of the data is recomputed. The presence of this hash address is verified against all the addresses in the hash cache. If there is a cache hit in the hash cache, the hash of the data is checked against the hash that is stored in the respective hash address memory location. If the hash matches then it is concluded that the state of the data is valid, if no then, it is concluded that the data is corrupted and the CPU aborts any operation on this data. If the hash is not present in the hash cache then the CPU checks for the next level in the hash cache until it reaches the root hash. The process of creating the Merkle hash trees and computing the hash addresses and hashes repeats recursively for all the data blocks written and read from the memory, during the execution of a process.



(A) Write operation



Read Dec(Value) and Hash from Memory
Compute the Hash Address
Check for Hash in the Hash cache
MISS
Check the next level Hash in the Hash cache
HIT
Return Data

(B) Read operation

FIGURE 3.4. Memory Integrity Verification Operations

The pseudo code of the function to compute the hash address at each level in a Merkle hash tree is given in Algorithm 2. The function takes three arguments — level of hash tree, data block address and index for the hash level. The hash block size, data block size, tree size, maximum levels of hash tree and offset address for each level are pre-initialized. At first, the block address is calculated based on the level and its offset address. Depending on the level, the block number is calculated which is then computed with tree size and index to

Algorithm 2 Function to compute Hash Address in a Merkle Hash Tree

```
Require: hash_size; block_size; tree_size
Require: max_hash_levels; BlockOffset
Ensure: level < max_hash_levels
Ensure: index < tree_size
1: block_addr ⇐ block_addr - BlockOffset[level]
2: if level == 0 then
3:     block_number ⇐ block_addr/block_size
4: else
5:     block_number ⇐ block_addr/hash_size
6: end if
7: block_number ⇐ block_number / tree_size
8: block_number ⇐ block_number ∧ index
9: hash_addr ⇐ BlockOffset[level+1] + (block_number × hash_size)
10: return (hash_addr)
```

fetch the hash address.

The overhead of MIV operation is several hashing operations, $\log N$ for N leaf nodes. One can cache some of these hash tree nodes to increase the efficiency. The granularity of a leaf node can be increased beyond a single word to an entire cache block. Despite these optimization, such hash trees are expensive primarily due to the cost of the underlying hash function. The two types of cost associated with hash function are performance cost and energy cost. AEGIS [67] charges 160 cycles for each hashing operation presumably at a cost of 2 cycles per round for 80 rounds of SHA [62]. This is an exorbitant performance cost for a memory integrity architecture that spawns many hash function instantiations for each memory read and memory write. Hence, more efficient mechanisms for memory integrity protection are required.

3.6. Related Previous Research

Blum et. al [7] have discussed the possible attacks on the contents of the memory and have demonstrated the need for memory integrity verification even if the memory contents are encrypted. They have proposed a hash tree based approach that is built on top of a trusted memory to authenticate rest of the untrusted memory. They have also proposed an offline checker that computes a running hash of all the memory reads and writes. This offline checker is used to verify the correctness of the memory after a sequence of operations are performed. The cost of their approach is $O(log(N))$.

Lie et. al have proposed a hardware secure architecture called XOM (eXecution Only Memory) [38], to authenticate the application data residing in external memory. In this approach, the application data and code are encrypted and stored in separate memory blocks. A MAC of the data and address is then computed for each memory block. This MAC is then used to verify the integrity of application data. Unfortunately, this approach does not protect replay attack, where the attacker replaces the current data value with a previously used stale value.

Suh et. al have also proposed a secure processor architecture known as AEGIS [67]. In this framework, it is assumed that only the components on the processor chip are trustworthy whereas all other components including the RAM and peripherals are considered to be untrustworthy. Hence encryption and authentication of memory contents is necessary to ensure both tamper evident and tamper resistant processing. Here, a Merkle hash tree is used to provide authentication to the data in memory. In [16], they have combined caches and hash trees to deliver the most performance efficient memory integrity verification scheme. The paper presents CHash and LHash with varying cache block sizes to analyze the performance overhead of each configuration.

Rogers et. al [59] have presented an Address Independent Seed Encryption (AISE) mechanism, which is a counter-mode based memory encryption scheme with a seed composition. It uses logical identifiers as a seed along with a counter instead of using the block address. These logical identifiers are unique across the entire physical and swap memory and over time. They have also proposed a Merkle hash tree based memory integrity verification mechanism known as the Bonsai Merkle Trees (BMT). In this approach, a MAC of encrypted data and its corresponding counter is computed and stored in the untrusted memory. Since the counters generated in the AISE mechanism have a notion of time, the integrity of data can be verified by solely comparing the MACs instead of computing the entire Merkle hash tree. Simulation results prove that this technique reduces the overhead on the system by 12% to 2% as compared to traditional approaches.

Yan et. al [74] have presented a combined memory encryption/authentication scheme. They have proposed split counters for counter-mode encryption, which eliminate counter overflow problems and reduces the size of the counter for each memory block. For the purpose of memory authentication, they have used the Galois/Counter mode of operation (GCM) that leverages counter-mode encryption to reduce the latency involved in the memory authentication process. In this approach, the memory authentication is done in parallel with encryption. Tags are updated when memory contents are modified to protect against replay attacks.

Rogers et. al [58] present an analogous tag generation approach as was proposed by [74]. The primary difference in this approach is that it applies a Parallel Message Authentication Code (PMAC) algorithm as against the GCM approach. The PMAC scheme allows for using a single hardware encryption component for both encryption and authentication, thus reducing the overall resource cost in an embedded system. Hong et. al [25] have also proposed a new tag generation approach. In this approach, a static inverse transform scheme is used for tag generation that offers a parameterized security design. They have customized the tag size and tag generation algorithm to achieve an optimal tradeoff between security and the design overhead.

Shi et. al [63] present an architecture for authenticating memory that is often shared amongst multiple processing elements in a computing system. They have proposed Authentication Speculative Execution (ASE), a new scheme to incorporate memory authentication into the processor pipeline. Here, the data and its integrity code (MAC) can be transmitted separately by virtue of tagging each data transaction. This tag is known as the transaction number. The ASE scheme is combined with one time pad (OTP) memory encryption mechanism to provide reliable security. The use of transaction number for authentication reduces memory latency, while only imposing a performance overhead of less than 5%.

Geobotys et. al [17] present optimized synthesis of new low energy masking alternatives into cryptographic softwares. The master key is initially masked with a set of masks. Masks are then periodically added throughout the life-cycle of key generation. A mask set is generated from an initial set of masks by combining all possible masks to create new masks. The final round keys are constrained to have a fixed mask. This mechanism uses the Boolean masking methodology combined with either Rijndael or the Advanced Encryption Standard (AES). Experimental results reveal that this scheme achieves an energy overhead savings of up to 2.5%.

Power-Smart System-On-Chip Architecture [69], presents an architecture for preventing sensitive information leakage via timing, power and electromagnetic channels. This architecture depends on a current sensing module to measure the power and current consumption of the system. They achieve significant success in measuring the current consumption of the system while limiting the power overhead to less than 12% of the total power.

In [57], Roger et. al describe an efficient hardware mechanism to protect integrity of softwares by signing each instruction block during program installation with a cryptographically secure signature. This technique serves as a secure and performance efficient alternative to conventional memory integrity verification module. While, Gelhert et. al [18] presents an architectural approach that protects against memory spoofing attacks. In this architecture, a secure hardware component called the FPGA guard is used, which can accelerate the execution of encrypted programs in a secure environment. The data protection techniques proposed in this paper achieves high level of security with significantly low performance overhead.

3.7. Conclusion

While designing computing systems there is always a trade-off between security and performance. A classic example of this are the embedded systems. Embedded devices are typically fast, miniaturized and specific to their application and hence often have severe energy constraints. As they now handle a lot of critical information, security becomes a crucial requirement. Therefore the need arises to design new security mechanisms so that their energy consumption is minimal while still preserving the security of the system. In this research project, the focus is specifically on reducing the energy consumption of memory integrity verification mechanisms in embedded systems. In this chapter, we have proposed a memory integrity verification framework to accurately measure the total number of hash function invocations during the execution of a program. This value is directly related to the total energy consumed by the MIV mechanism and therefore serves as baseline to compare the total energy savings offered by our proposed mechanisms presented in Chapter 4, Chapter 5 and Chapter 6.

CHAPTER 4

MEM-DNP: DETECT AND PROTECT MECHANISM [1]

4.1. Introduction

Embedded processors utilize multiple sensors to interact with their environment. These sensors can detect any physical or software attack against the memory. As an attack initiates, fluctuations can occur in the processor's physical properties (e.g. an instantaneous current spike or a jump in power or some kind of dissipation thermally and so forth). The corresponding sensors can thus measure such fluctuations. Only when the fluctuations exceed a pre-determined value is MIV executed. This use of sensors is nontraditional as an operation, for the MIV must be carried out for all the memory blocks read from the off-chip memory. Under the traditional MIV approach, for every memory write operation, the Merkle hash tree is constructed over the entire memory and a root hash is computed and stored in the on-chip memory. This process proves uncircumventable considering that the most recent root hash is always required to productively carry out the verification process during the memory read operation. Thus the processor expends huge amount of energy in the MIV process. However, in the MEM-DnP approach the Merkle hash tree based verification is only turned to when the fluctuations reach a certain threshold. Resultingly, significant amount of energy is conserved during the verification phase. Primarily, this mechanism intuitively anticipates the occurrence of an attack and only then engages the hash verification. During conditions of normalcy, therefore, exists no requirement for hash verification and so the processor can simply trust the data values read from the memory.

The use of sensors within the processor chip boundary, to measure critical physical properties has been mined by research perennially now. For example, Oh et al. [52], McGowen et al. [41], and Zhang et al. [77], deploy hardware sensors in order to measure power dissipation and thermal dissipation in the CPU. Similarly, Muresan et.al. [69] make use of a

[1]Parts of this chapter have been previously published, either in part or in full, from [47] with permission from IEEE and [49] with permission from Springer.

Current Sensor Module to forecast the power consumption of their architecture.

This chapter is organized as follows. Section 4.2 describes the architecture and the overall functioning of the MEM-DnP mechanism. Section 4.3 introduces the concept of Disjoint hash Tree with regards to MEM-DnP operation. Section 4.4 presents a theoretical basis for the sensor operation in MEM-DnP. Section 4.5 analyses the performance of MEM-DnP, and Section 4.6 presents the conclusion.

4.2. MEM-DnP Architecture

The architecture of the MEM-DnP mechanism is shown in Figure 4.1. The Sensor Module (SM), exemplifies the hardware sensors in an embedded system. The SM constantly monitors the memory bus for any vacillations in its physical properties. The architecture contains a specialized cache — hash cache for storing the generated hash addresses and hash values during MIV. The hash cache holds the hash address and the corresponding hash value of each memory block written to the off-chip memory. This hash address and the hash value are used to verify the memory's integrity. Owing to the processor chip being trustworthy, the SM and hash cache are located inside the processor boundary. Importantly to note here is that the research in this dissertation only targets MIV and not encryption/decryption mechanisms. An encryption/decryption block, componential in the general architecture of the MEM-DnP, merely represents a broader view of a secure processor architecture.

The functioning of the MEN-DnP can be illustrated per its two fundamental memory operations — memory write operation and memory read, shown in Figure 4.2.

in terms of its memory operations, as shown in Figure 4.2. Here, there are two basic operations — memory write operation and memory read operation. During the memory write operation (Figure 4.2a), prior to writing a memory block to the off-chip memory, the CPU computes a corresponding hash value and hash address and stores it in the hash cache. This memory block is then encrypted and stored in the off-chip memory. During the memory read operation (Figure 4.2b), the memory block ascertained from the memory is first decrypted. The SM continuously monitors the memory bus to detect an attack, keeping a Threshold value (V_T) for the fluctuations. If the systems fluctuations exceed this V_T, that an anomaly

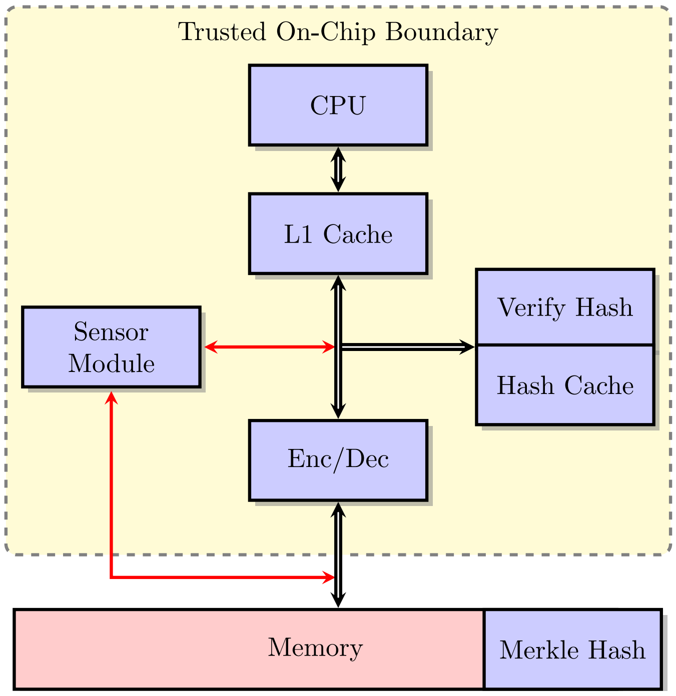
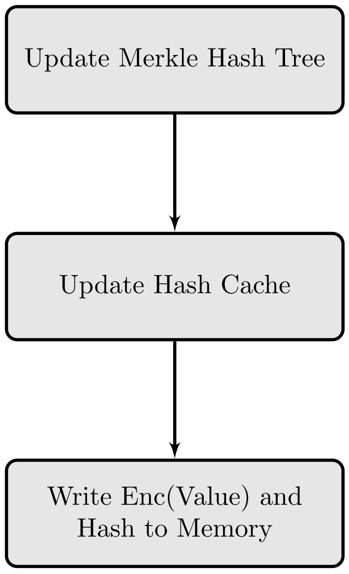
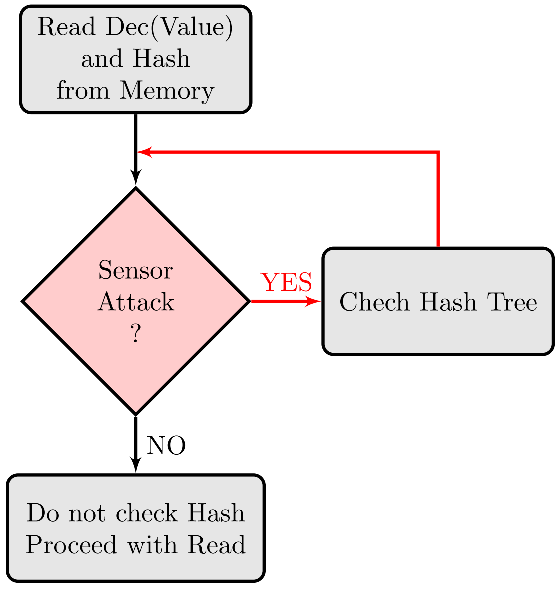


FIGURE 4.1. MEM-DnP Architecture

lies in the system stands apparent. In the case of such an anomaly, the hash address of the data is re-computed. The hash address is checked against all the addresses in all hash cache levels. If there is a cache hit in the hash cache, the hash of the data is corroborated against the hash that is stored in the hash cache. If hash matches hash, then the state of the data is valid; else if the hashes do not equate, the data is corrupted and there is an attack on the system. In such a case, the CPU aborts any operation on this data. This process is recursive and continues for all the memory reads performed in the window of fluctuations exceeding V_T . During this time, if the fluctuations equilibrialize and then drop below the V_T, then the threat has subsided. At this point, the hash address verification process ceases. The SM and the MIV (hash cache) modules are in enjoinment with each other so that the MIV is only performed when required, i.e., at the time of unconventional changes or steep discrepancies. The experimental results in Section 4.5, prove that this mechanism yields impactful energy savings as compared to a traditional MIV mechanism.



(A) Write operation


Read Dec(Value) and Hash from Memory
Sensor Attack ?
YES
Chech Hash Tree
NO
Do not check Hash Proceed with Read

(B) Read operation

FIGURE 4.2. Memory Operations in MEM-DnP Architecture

4.3. Disjoint Hash Trees

The SM continuously superintends the memory bus, while the CPU operates on the memory blocks. The MIV module (hash cache) works closely with the SM to commence verification when discrepancies in the readings are found. From this point, the MIV module starts recomputing the hash address and the corresponding hash for the verification purpose. This is cyclic process and lasts until the fluctuations stabilize. To reduce the energy consumption in the verification phase, this we present a novel mechanism of Disjoint hash trees, evinced in Figure 4.3. The cerebration to this approach is to divide the original Merkle hash tree into smaller trees. According to the location of the infected memory block, a suitable Disjoint Tree is relied on for the verification process. The MEM-DnP mechanism trusts the precision of the SM to accurately locate the infected memory block. The primary advantage of this technique is that the hash tree is not re-constructed over the entire memory; instead, what is re-constructed is only that particular Disjoint tree which contains the infected memory block. At this point, importantly what needs to be emphasized is that the number of Disjoint Trees built in the memory forthrightly relates to the number of secrets able to be stored on-chip. In a traditional implementation of a Merkle hash tree, only one root hash i.e. one secret is kept on-chip. Whereas in the proposed mechanism, multiple root hashes, i.e. numerous secrets corresponding to each Disjoint Tree are placed on-chip. Hence this approach exploits the availability of on-chip memory space to provide energy saving returns during the integrity verification process.

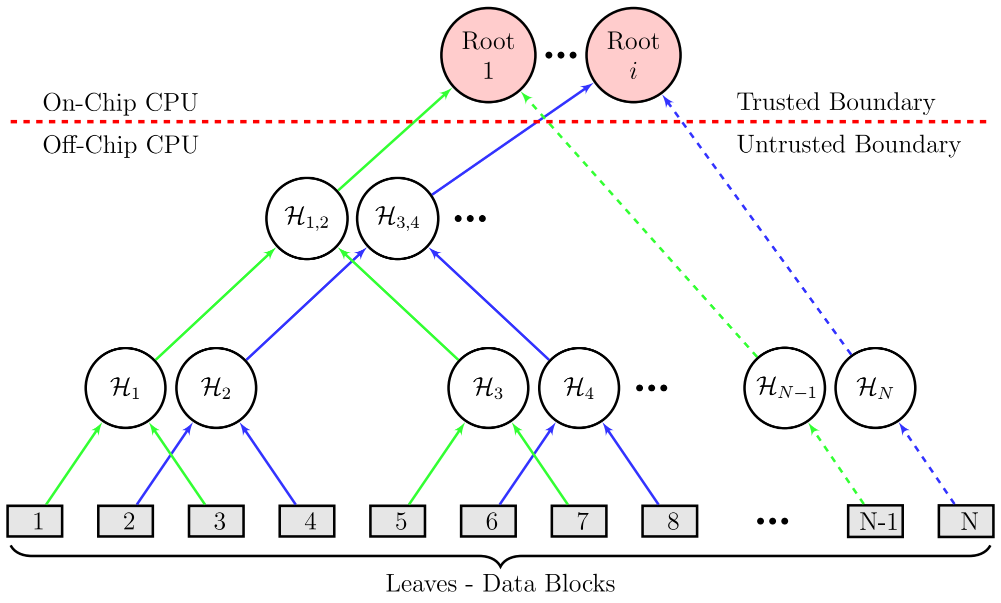


FIGURE 4.3. Disjoint Hash Tree

4.4. Relation between Process Variations in Sensors and V_T

The measurements recorded from a sensor manufacture an error profile that can be accredited to the distribution of noise that impacts the sensor's readings. What is responsible for this noise owes itself to Process Variations [20] [19] [45] that occur in the VLSI circuit of a sensor. Zhang et al. [75] have shown that the noise related to the process variations is the phase noise of the circuit. The assumption that "all transistors are alike" is not valid when considering nanochips. It is essential that the variations in two transistors in a chip or else two or more different chips having the same design are considered in any design decisions for making circuits robust and for improving the target outcome of the Design for Manufacturing (DFM). The device parameters, chip performance, and the chip yield are influenced by industrial/engineering/fabrication process variations. The electrical parameters and the wholesale performance of any chip is decisively affected by the process variations and the effects from inconsistencies become evident in the variations in power and delay and other attributes that reside with and are demonstrated by the chip. The process variations can be either inter die or intra die, can fall as random or as systematic, can be correlated or uncorrelated, or can be spatial or temporal. If such variations as these (and other potentially) are not considered, such a design oversight may lead to significant design errors and yield losses.

Therefore, recently, researchers have explored approaches to designing alternatively to guarantee that the yield of the VLSI circuit undergoes minimal impact from process variations. The three main types of approaches for modeling process variations are as follows: statistical design approach, post silicon process compensation/correction, and avoidance of variation induced failures. Statistical design methodology has been widely looked into as an efficient method for yield under process variation. Whereas the post-silicon process compensation/correction has also been extensively explored, it detects process variations using the on-chip process sensors and it may also involve manufacturing tests and compensating/-correcting circuit parameters that may have deviated due to process variations. And finally avoidance of variations-induced failures is based on the concept of critical path isolation that makes a circuit subject to voltage scaling while still being robust to parametric variations.

For this dissertation, the statistical modeling technique for modeling the process variations in a sensor (directly related to the sensor noise) has been chosen. A sensor's noise samples are generally modeled as continuously valued random variables, while the noise waveforms are modeled as random processes. A continuous random variable X with non-negative density F, is specified by its probability density function (PDF) and is given in Equation 1.

$$f_X(x) = \int_{-\infty}^{\infty} f_X(x)\,\mathrm{d}x. \tag{1}$$

Given a PDF for a random variable, it is then possible to calculate the mean (μ) and the variance (σ^2) for the random variable. Both μ and σ^2 estimate the power of noise or the random variable X and are given in Equations 2 and 3.

$$\begin{aligned} \mu_X &= \int_{-\infty}^{\infty} x f(x)\,\mathrm{d}x, \\ \mu_X^2 &= \int_{-\infty}^{\infty} x^2 f(x)\,\mathrm{d}x \end{aligned} \tag{2}$$

$$\sigma_X^2 = \mu_X^2 - (\mu_X)^2. \tag{3}$$

The most common random variable used to model noise is the Gaussian. Also, the noise distribution in sensors follow a Gaussian distribution with a zero mean [15], shown in Figure 4.4. The PDF in a Gaussian distribution is given in Equation 4

$$f(x) = \frac{1}{\sigma_X\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu_X}{\sigma_X})^2} \tag{4}$$

In the case of MEM-DnP approach, the sensor module measures many independent signals

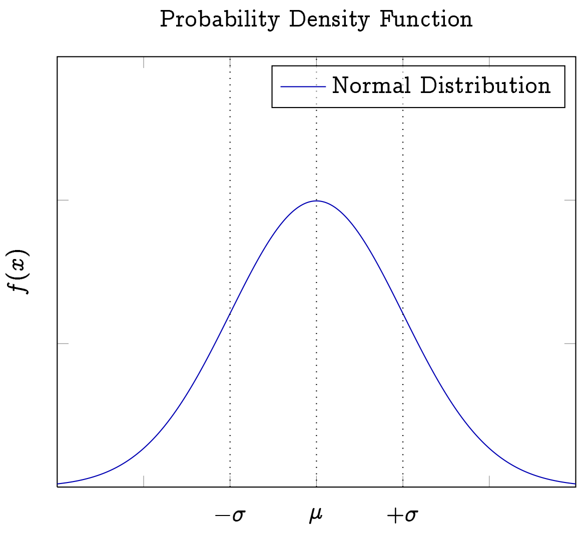


FIGURE 4.4. Gaussian Distribution Curve

each having its own individual error/noise profiles. Thus mathematically a sensor module can be viewed as a Gaussian process P with 1...N individual continuous valued random variables. While modeling the V_T for the sensors, key is to know the relationship between individual signals at different time instants. For ease of understanding, let us consider X and Y as two individual normally-distributed random variables with zero mean. Let F(X) and F(Y) be the PDFs, respectively, of X and Y. The co-variance of X and Y is given in Equation 5.

$$COV(X,Y) = F(XY) - F(X)F(Y) \tag{5}$$

However, since both X and Y are individual random variables, $F(XY) = F(X)F(Y)$. Therefore $COV(X,Y) = 0$. Thus the two individual random variables X and Y in the same Gaussian process stand uncorrelated. The similar also shows true for 1...N random variables in the same Gaussian process. Now, since the individual random variables are uncorrelated, the resultant variance of the Gaussian process (σ_P^2) is nothing but the product of variances of N individual random variables in the process, given in Equation 6.

$$\sigma_P^2 = \sigma_1^2 * \sigma_2^2 * \sigma_3^2 .. * .. \sigma_N^2 \tag{6}$$

Regarding sensor, there are false positives and false negatives affecting its reading. A false positive measurement will indicate an attack when in reality none exists, whereas a false negative measurement is the one where an actual attack goes undetected by the sensor. While modeling such a system both false positives and false negatives must stay within a minimal allowable range. Using the MEM-DnP approach, the emphasis lies on false positives. As for the MEM-DnP simulations, a Random value (RV) is embedded to simulate one such false positive that indicates fluctuations in the readings. The details about RV is discussed in Section 4.5.3. The Threshold value V_T is directly related to an RV. For an N-bit RV the probability that a fluctuation occurs is given by $\frac{1}{2^N}$. Also in the case of sensors, the mean (μ) and variance (σ^2) of a sensor is modeled during its design process so as to accommodate the random noise in a sensor's readings. Thus given the values of μ, σ and RV, to compute the tolerance (τ) of the sensor circuit becomes doable. The tolerance τ of a sensor circuit is a point on the Gaussian distribution curve, beyond which we consider that the false positives occur in the system, a system then considered under an attack. This can be optimally explained by the following example.

As a case study, Figure 4.5 represents a statistical characterization of process variation

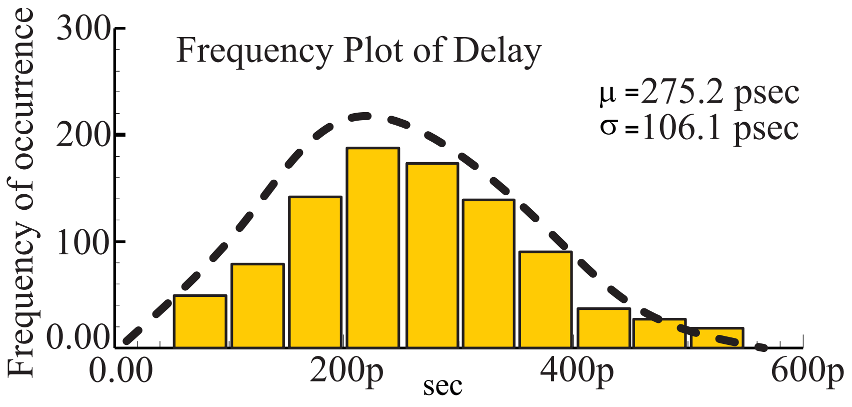


FIGURE 4.5. Example: Propagation Delay PDF in a 2-input NAND Gate

in a 2-input NAND logic gate. This example illustrates a plot of propagation delay in the NAND gate. The NAND gate is designed, tested and realized in a 45 nm CMOS technology. Monte Carlo simulations [8] are executed to translate the process and design variations at the input side, to the propagation delay at the output side. The CMOS transistor parameters in the NAND gate, considered during these simulations are channel length (L), device width (W), gate oxide thickness (T_{ox}), device threshold voltage (V_{th}) and supply voltage (V_{dd}). It is observed that the propagation delay follows a Gaussian distribution. For ease of simulation, modeling and understanding we have only considered a single 2-input NAND gate. However, realistically a circuit may contain several NAND gates. Since the propagation delay PDF is Gaussian in nature, the input distribution at each gate is statistically independent. Therefore in the case of N NAND gates in a sensor circuit, the mean (μ) and the variance (σ^2) of the distribution is given in Equation 7.

$$(7) \qquad \mu_{SENSOR} = \sum_{i=1}^{N} \mu_{NAND_i} \qquad \sigma_{SENSOR} = \sqrt{\sum_{i=1}^{N} \sigma^2_{NAND_i}}$$

Similarly, a sensor circuit may contain several logic. But since the output distribution of each gate is Gaussian in nature, the overall PDF of the sensor will also be Gaussian in nature. For instance if $X, Y ... N$ are the Gaussian output distributions of the logic gates in a sensor circuit, the overall PDF of circuit $= X + Y + ... + N || X * Y * .... * N || X \oplus Y \oplus .... \oplus N$ is also Gaussian in nature. This curve has a μ=275.2 psec and σ=106.1 psec. As discussed earlier, since the value of μ and σ is known, it is possible to compute the τ for the sensor circuit for RV=16, 20 and 24. In a Gaussian distribution curve this can be achieved using a cumulative distribution function (CDF) given in Equation 8.

$$CDF = \frac{1}{2}[1 + erf(\frac{x-\mu}{\sqrt{2\sigma^2}})] \tag{8}$$

Here erf stands for error function that is typically used for measurement in probability and statistics. For a random variable X, the CDF represents the probability of X as - $P(X) < x$ on the Gaussian curve. However, to calculate τ we will fix the probability as - $P(X) > x$. Thus this probability will be (1 - CDF). Therefore by applying to Equation 8, we will get Equation 9.

$$1 - CDF = 1 - \frac{1}{2}[1 + erf(\frac{x-\mu}{\sqrt{2\sigma^2}})] = \frac{1}{2}[1 - erf(\frac{x-\mu}{\sqrt{2\sigma^2}})] \tag{9}$$

Now for RV=16, the probability of a false positive occurring in the sensor's reading is: P(FP)= $\frac{1}{2^{16}}$. In order to compute the value of τ, we will fix this probability. Hence consider that (1 - CDF) = P(FP) = $\frac{1}{2^{16}}$. Also the co-ordinate "x" on the Gaussian curve will represent the value of tolerance - τ of the sensor, where the false positives will occur. Hence given the values of RV, μ and σ, the goal is to compute the value of τ.

$$\frac{1}{2^{16}} = \frac{1}{2}[1 - erf(\frac{\tau-\mu}{\sqrt{2\sigma^2}})] \tag{10}$$

Hence by solving Equation 10, the value of τ can be computed. Therefore solving Equation 10 gives us the value of τ as: $\tau = \mu + 435.01 \times 10^{-12} = 710.21$ psec. Similarly the tolerance of the sensor can be computed for RV=20 and 24 as well. The tolerance value is a crucial parameter that indicates, where on the Gaussian curve, a false positive reading might be recorded, based on the μ, σ and RV. At this point it is important to understand that the values of μ, σ and RV should be optimally selected during designing and modeling the sensor circuit, so that the value of τ is low. A higher value of τ may lead to severe issues. This is shown in Figure 4.6. This figure shows a Gaussian distribution curve for a typical sensor circuit. It is represented as an original PDF with mean μ_o and standard deviation σ_o. Now

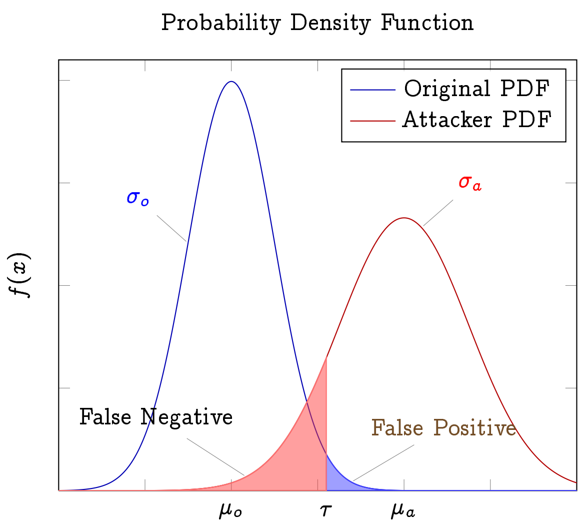


FIGURE 4.6. Tolerance of Sensor as compared to Attacker PDF

consider that an attacker is assaulting the system and the sensor is responsible for measuring the fluctuations in the system. The attacker distribution also follows a Gaussian distribution, represented by an attacker PDF with mean μ_a and standard deviation σ_a. In such scenarios, the value of tolerance τ factors significantly in the overall security of the system. As the τ is closer to μ_o, the probability of false positives is more than the probability of false negatives. But as the τ moves further away from μ_o, the overlap of attacker PDF on original PDF increases and thus the probability of false negatives increases over the probability of false positives. This is a critical security condition as the sensor will fail to detect a legitimate attack measurement. Therefore the value of τ should be designed depending on the value of μ_o, σ_o and RV.

4.5. Experimental Evaluation

This section presents an analysis of the energy consumption in a traditional MIV process as opposed to the energy consumption of the proposed MEM-DnP mechanism.

4.5.1. Simulation Framework

The simulation framework derives from Simplescalar Tool Set [4], which is configured to execute ARM binaries. Since the primary goal here is to demonstrate the energy efficiency

Table 4.1. Cache Configurations

Cache	Specifications
L1-D Cache	8KB, 2-way, 32B Line
L1-I Cache	16KB, 2-way, 32B Line
L2-D Cache	None
L2-I Cache	None

of the proposed MEM-DnP mechanism in embedded systems, MiBench [23] embedded benchmark suite has been used, that best replicates the variety of practical applications run on embedded devices. The results obtained from different benchmark programs are presented, to demonstrate the efficiency of the proposed mechanism. All the simulations performed are cache based simulations, using the sim-cache simulator in simplescalar. The cache configurations used for the simulations is given in Table 5.1. Here, the level 1 cache configurations are selected to match the typical configurations of embedded ARM processors [3].

4.5.2. Baseline Energy Consumption

As described in Section 3.4, MIV is performed by constructing a Merkle hash tree in the memory and storing the final hash, known as the root hash in the on-chip memory storage. For every memory read, the Merkle hash tree is re-constructed and the resulting root hash is compared with the one already present on-chip. If both the values match, then it is concluded that the memory block is verified; otherwise, else, it is infected. Typically, this is implemented by partitioning the memory into fixed sized blocks (usually the same in size as those of the cache) and generating multiple levels of hashes. Hence energy is consumed while generating the hash from the memory all the way up to its root. To measure this, we have exhibited Algorithm 2 in Section 3.5. Here the algorithm is used to measure the number of hash invocations required per data miss in the level 1 data cache. Hence, given that the energy consumption per hash invocation is known, the results obtained from the algorithm can be used to calculate the total energy consumption of the MIV mechanism.

TABLE 4.2. Baseline Simulations showing Total Hash Invocations and the Average Hash Rate

Benchmark	DL1 Misses	Total Hash Invocations	Avg. Hash Rate
dijkstra_small	1501134	21007122	13.99
fft_large	6570198	91732597	13.96
jpeg_large	2923393	40925500	14.00
lame_large	39221353	549097030	14.00
patricia_large	9138017	127930138	14.00
qsort_large	7114792	99603844	14.00
math_large	1318621	18090557	13.72
sha_large	264246	3287898	12.44
stringsearch_large	76794	1070293	13.94
susan_large	109933	1537169	13.98
blowfish_large	5074709	70618620	13.92
crc_large	11237338	97672098	8.69

Table 4.2 shows the DL1 Misses, Total Hash Invocations and the Average Hash Rate for 12 embedded benchmark applications. The average hash rate is calculated by dividing the total hash invocations by the DL1 Misses in each of the benchmark applications. Since the average hash rate is directly related to the total hash invocations, it is also directly related to the energy consumption in the MIV mechanism. The more the average hash rate, the more are the total hash invocations and thus the more is the energy consumed by the MIV mechanism and vice versa. Therefore, here, in this research, the emphasis is on reducing the average hash rate by employing a sensor module to detect possible attacks on the system. This is explained in detail in Section 4.5.3 along with the results.

4.5.3. MEM-DnP Energy Consumption

The SM and the MIV module work together for the memory: to detect for it and protect it from physical attacks, as explained in Section 4.1. Sensor accuracy is responsible for sleuthing with precision the infected memory blocks. The functioning of MEM-DnP mechanism results in two false positives. The first false positive arises due the action of sensors. A typical non-ideal sensor would manifest certain abnormal fluctuations. This non-ideal sensor characteristics is modeled in the simulation framework by using a Random Value (RV), which is a random number generator that can generate 16 bit, 20 bit, or 24 bit random numbers. The RV represents the probability of occurrence of such fluctuations in the sensor readings. For an N-bit RV the probability that a fluctuation occurs is given by $\frac{1}{2^N}$. The fluctuations may, as should be stressed, arise even if there is no potential attack on the system. The second false positive again relates to the activity of the sensor. At the time of fluctuations in the readings, a sensor paradigmatically has the enablement to auto-correct itself and stabilize its readings. For an ideal sensor the time required is 0 for auto-correction. In practice, however, a finite amount of time is always mandatory if the sensor is to auto-correct itself. This finite time is modeled in the simulation framework through the use of Window Size. Window Size corresponds to the number of memory accesses required for the sensors to stabilize. The Window Size accommodates 3 possible values — 2000, 8000, and 15000 memory accesses. Finally, the Disjoint Trees correspond to the number of secrets or root hashes that can be stored on-chip. Therefore the number of Disjopint trees depend on the amount of memory available in the on-chip storage. The number of Disjoint Trees are tested for three values — 16, 64, and 128.

The simulation Algorithm 3 displays the steps for the simulation of MEM-DnP mechanism. A predetermined Attack Seed of the same size as that of the Random Value is embedded in the simulation framework. During the simulations, when the values of Random Value and Attack Seed match, it can be concluded that fluctuations in the sensor readings have exceeded V_T and that an anomaly rests within the system. At this point, the MIV module generates the Disjoint hash tree and computes the hash address and hash of the memory blocks and proceeds with the verification process. This process is recursive i.e. the Random Value and the Attack Seed are constantly checked to detect if the match exists. If again a match comes up, the reappearance/reoccurrence indicates that the anomaly persists and the hash verification process continues. If the sensor measurements stabilize, the MIV module halts the hash verification process.

Algorithm 3 Simulation Algorithm

```
Require: Random Value, Window size, Disjoint Trees.
1: SM will monitor the system to detect an attack.
2: if Random attack seed = Random Block. (Fluctuations >$V_T$. Anomaly exists)
3: Invoke MIV for given Window size and Generate Disjoint Trees.
4: Keep repeating steps 2 & 3.
5: If there is match again. (Anomaly persists)
6: Extend the Window size; keep generating Disjoint Trees.
7: Else, otherwise, attack has subsided.
8: Stop the MIV process.
9: Keep repeating step 2.
```

The MEM-DnP mechanism is tested for all possible combinations of the two false positives: Random Value and Window Size and the parameter Disjoint trees. Using Algorithm 3, simulations were performed on the same embedded benchmarks applications that are used in basecase simulations and the new average hash Rate was recorded. The new average hash rate was compared with the one in basecase simulations. Note, the lower the average hash rate, the greater are the energy savings. Figures 4.7a, 4.7b and 4.7c show the reduction in average hash rate for Disjoint trees = 16, 64, and 128 respectively. The reduction in average hash rate plotted for all possible values of Window Size and Random Value. Here the reduction in average hash rate is almost 1.00, i.e., 100% for Random Value - 24. This is expected as the probability that the random number generator would generate a Random Value matching the Attack Seed is the least for value - 24 as compared to 16 and 20. Also except for Disjoint tree-128, the reduction in average hash rate is the least for Random

Value-16 and Window Size-15000. This is due to fact that the Random Value-16 will result in a higher probability of a match with the Attack Seed and the number of memory accesses (Window Size) for which MIV functions are predominantly higher. Hence the reduction in average hash rate will be lower.

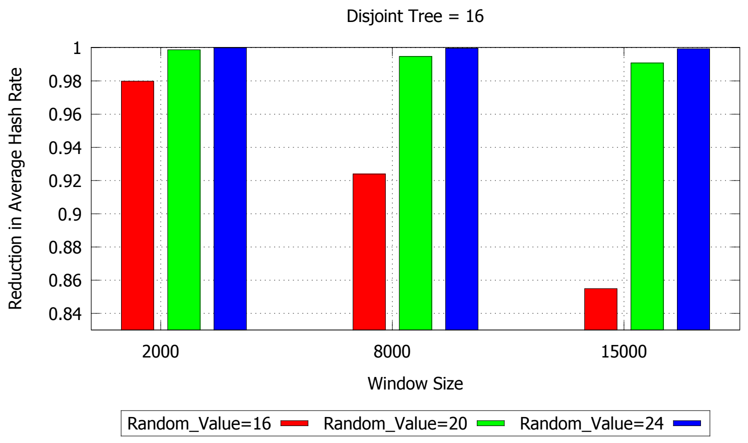
Figures 4.8a, 4.8b, and 4.8c show the reduction in average hash rate for Window Size = 2000, 8000 and 15000 respectively. The reduction in average hash rate is plotted for all possible values of Disjoint tree and Random Value. Here again the reduction in average hash rate is almost 1.00 i.e. 100% for Random Value - 24, whereas for Random Value - 16 it is the least.

Finally Figures 4.9a, 4.9b, and 4.9c show the reduction in average hash rate for Random Value = 16, 20 and 24 respectively. The reduction in average hash rate is plotted for all possible values of Window Size and Disjoint trees. Here the reduction in average hash rate comes in highest for Window Size-2000. This is expected as the number of memory accesses for which the MIV functions reside significantly low and hence the average hash rate resides low.

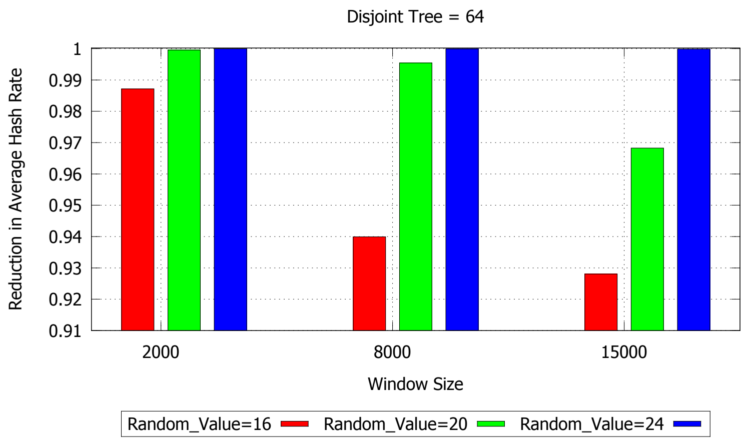
Based on the reduction in average hash rate, the average energy savings with regards to the basecase simulations, are calculated. Tables 4.3, 4.4 and 4.5 indicates the average energy savings for Disjoint trees = 16, 64 and 128 respectively. The energy savings are least for Disjoint tree = 16, Window Size = 15000 and Random Value = 16 with approximately 85.5%. While the energy savings are the highest for Disjoint trees = 64, Window Size = 2000, and Random Value = 24 with 99.998%.

4.6. Conclusion

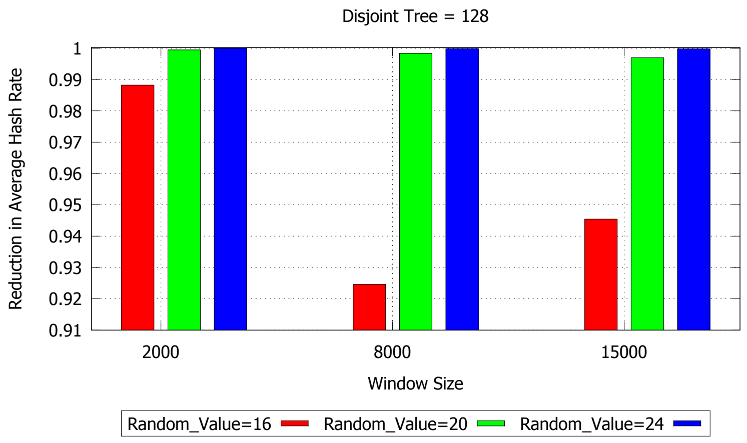
This chapter presents the *MEM-DnP* mechanism, a novel approach to achieve energy efficient memory integrity verification in embedded systems. This mechanism relies on a sensor module to detect any kind of a memory based attack on the system. The MIV operation is only carried out in the event of an anomaly i.e. when the fluctuations rise above the Threshold value V_T. Hence a lot of energy is saved by de-coupling the MIV process during normal execution of the system. The simulation results prove that the average energy



(A) for Disjoint Trees-16

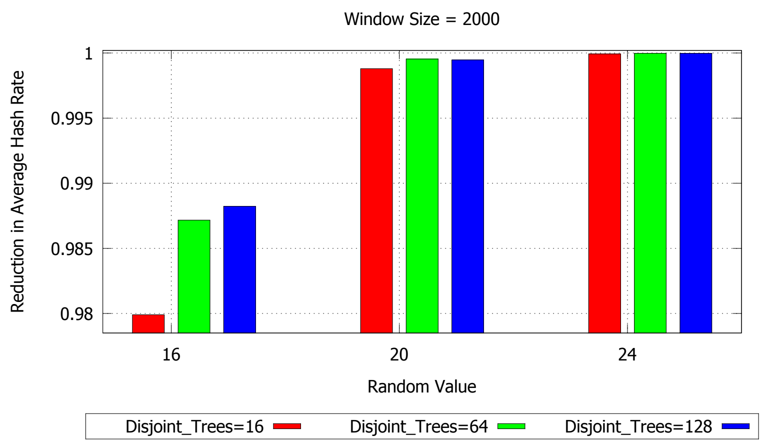


(B) for Disjoint Trees-64

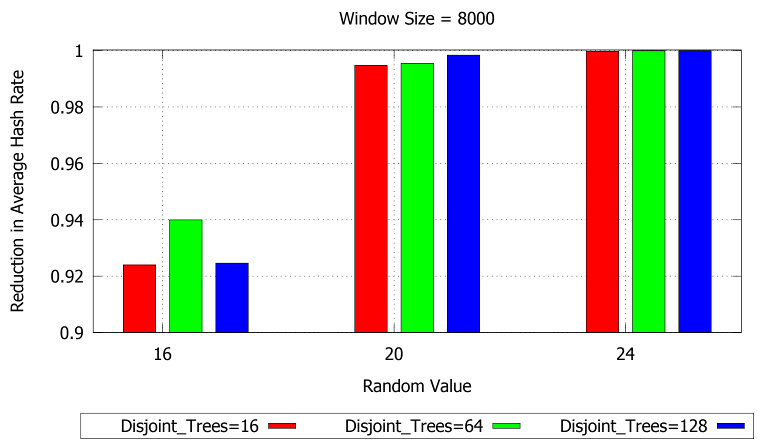


(C) for Disjoint Trees-128

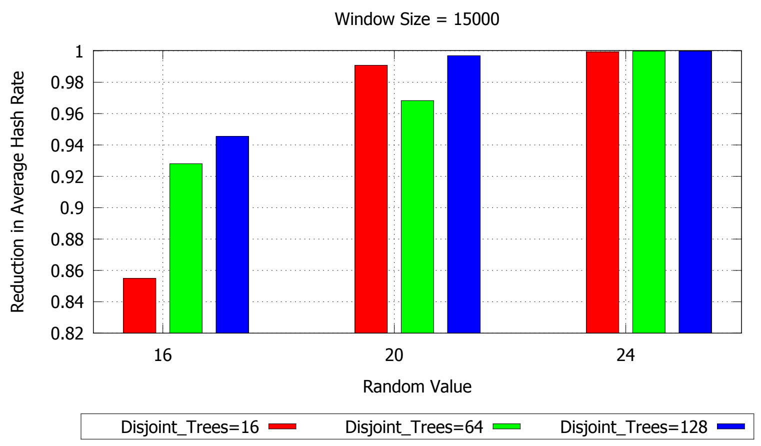
FIGURE 4.7. Reduction in Average Hash Rate for different Disjoint Trees.



(A) for Window Size-2000

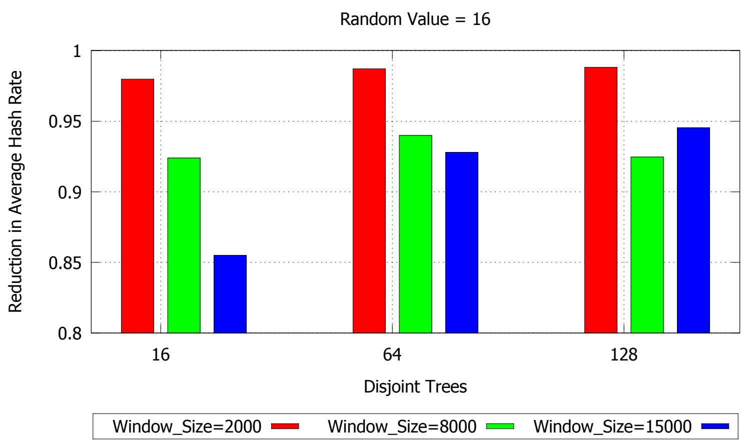


(B) for Window Size-8000

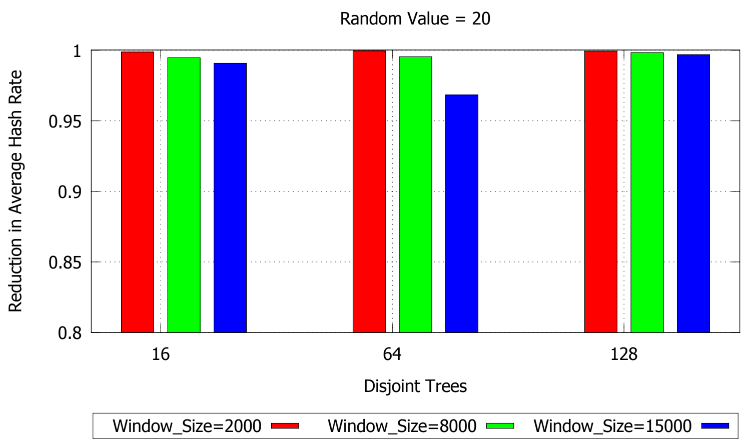


(C) for Window Size-15000

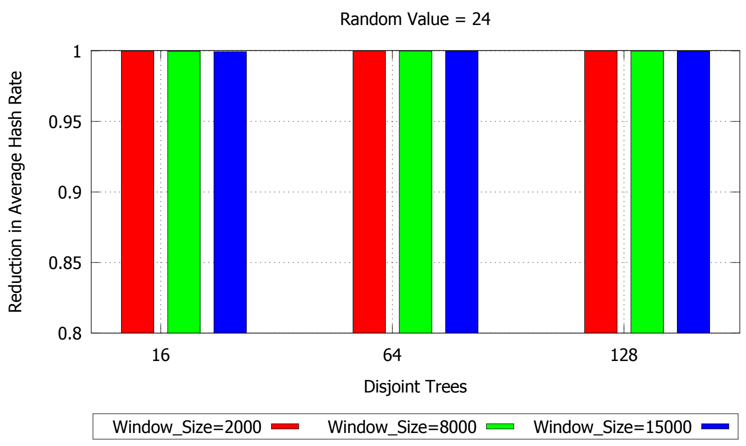
FIGURE 4.8. Reduction in Average Hash Rate for different Window Sizes.



(A) for Random Value-16



(B) for Random Value-20



(C) for Random Value-24

FIGURE 4.9. Reduction in Average Hash Rate for different Random Value.

Table 4.3. Average Energy Savings for Disjoint Tree = 16

Disjoint Tree = 16		
Window Size	Random Value	Average Energy Savings
2000	16	97.990
	20	99.880
	24	99.993
8000	16	92.394
	20	99.475
	24	99.971
15000	16	85.492
	20	99.080
	24	99.934

savings are in the range of 85.5% to 99.998% as compared to the basecase simulations with traditional MIV techniques.

TABLE 4.4. Average Energy Savings for Disjoint Tree = 64

Disjoint Tree = 64		
Window Size	Random Value	Average Energy Savings
2000	16	98.717
	20	99.954
	24	99.998
8000	16	93.995
	20	99.543
	24	99.989
15000	16	92.804
	20	96.828
	24	99.985

Table 4.5. Average Energy Savings for Disjoint Tree = 128

Disjoint Tree = 64		
Window Size	Random Value	Average Energy Savings
2000	16	98.823
	20	99.947
	24	99.998
8000	16	92.461
	20	99.837
	24	99.989
15000	16	94.546
	20	99.696
	24	99.984

CHAPTER 5

TSV: TIMESTAMPS VERIFICATION MECHANISM [1]

5.1. Introduction

The Timestamps Verification (TSV) mechanism runs on the principle of locality. Most programs display this behavior of utilizing a small set of addresses, names a working set, during an epoch of execution. The Working Set Model [13], offered by Peter J. Denning, models program behavior in computing systems. It was observed that during computational operations, programs obey the principle of locality (i.e. a program's formerly referenced pages) and may predict which pages will likely be revisited/re-referenced. To this end, the working set model was suggested as the one to provide the solution to system's general resource allocations. From the program's standpoint, the working set of information W(t, τ) of a process at time t is the assemblage of information referenced by the process in time interval (t - τ,t), where τ is the working set parameter. The cerebration here is to encourage the processor to recall the timestamps of any one working set of addresses. These timestamps can then aid in verifying the integrity of the data instead of re-constructing the Merkle hash tree for integrity checking. In this way, large energy conservings can be realized during the verification phase. In Chapter 4, we proposed the MEM-DnP mechanism that relies on on-chip sensors to detect and protect against any attacks on memory integrity. Here our goal is to present a mechanism that is independent of any sensors, thereby completely eliminating the false-positives introduced by the sensor operation.

The rest of this chapter is organized as follows. Section 5.2 describes the architecture and the overall functioning of the TSV mechanism. Section 5.3 analyses the performance of TSV and Section 5.4 presents the conclusion.

[1]Parts of this Chapter have been previously published, either in part or in full, from [48] with permission from Elsevier.

5.2. TSV Architecture

In the proposed TSV architecture, shown in Figure 5.1, the processor timestamps every value on a memory write operation and encryptedly stores the timestamp along with the data. The timestamps can be generated via pseudorandom number generators (e.g. block ciphers). These timestamps are kept on-chip having space limitations; this cache is the Timestamp Cache, as it is called. When the cache fills up, older timestamps, must, as a result, be evicted; therefore, merely a confined set of timestamps remain stored. In this mechanism, two operations exist: memory write and memory read, per, respectively, Figures 5.2a and 5.2b.

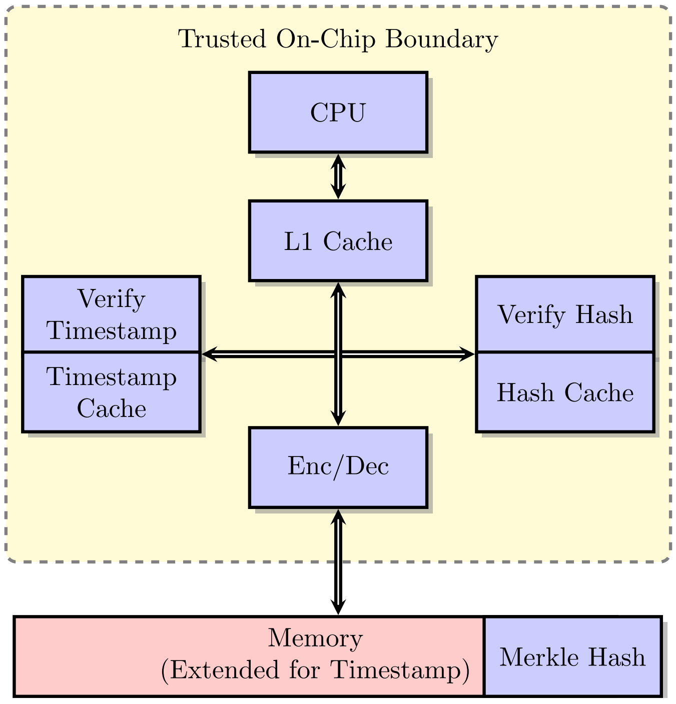
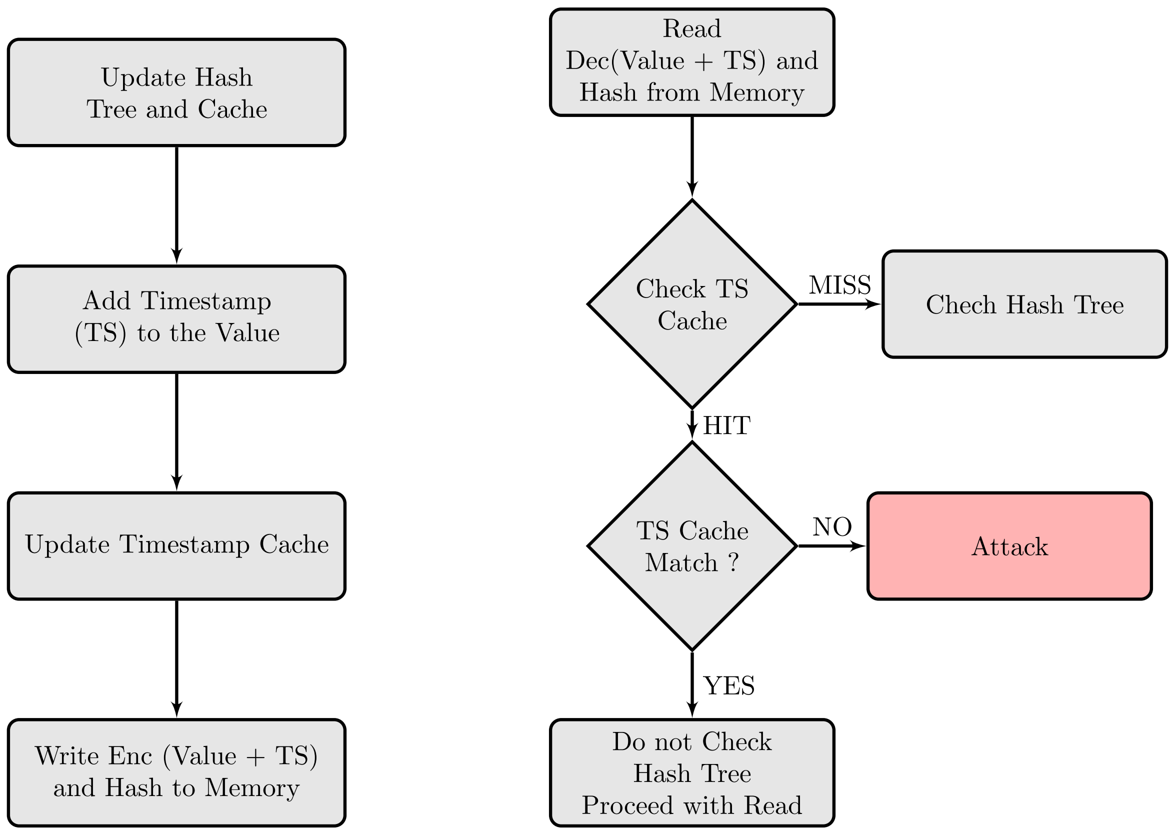


FIGURE 5.1. Timestamp Verification Architecture (TSV)

During memory write, the CPU initially constructs the hash tree and saves the root hash in the hash cache. Subsequently, an unique timestamp, which is itself warehoused in the Timestamp Cache for later verification, get created for each memory block. The memory block and the given timestamp are then encrypted through the encryption module and written to the main memory. During the read operation, the CPU first fetches the decrypted



(A) Write operation (B) Read operation

FIGURE 5.2. Memory Operations in TSV Architecture

memory block and its timestamp. The CPU then compares the associated timestamp against those preexisting in the Timestamp Cache. If it comes up a miss, then the processor performs the Merkle hash tree verification as an integrity validation. But it falls on the side of being a hit, then the processor compares timestamp values. If both the values match, the processor conclusively affirms that the integrity of the data is intact; otherwise, if a mismatch is appellated, an attack on the system is identified and the processor aborts any further operations regarding that line of data. The security of the integrity protection owes its effectiveness to the security of the encryption function. Any splicing, spoofing, and replay attack will be detected because the decryption function will generate a pseudorandom timestamp very less likely to match the timestamp stored on-chip. The security is proportional to the size of the timestamp. For an N-bit timestamp the probability that any attack on memory will not be detected (attack succeeds) is $\frac{1}{2^N}$.

The advantage primarily to this approach is in the hash verification phase. If the timestamp is available in the Timestamp Cache, then the processor is no longer required to re-route its way along the hash tree to ascertain the integrity of each memory block it acquires from the memory. Also, while verifying the hash, the CPU has to expend energy in likening the hash values at each level, until a hit results. This expending may be required until the root hash is verified. Hence, an amplitude of energy is used up by the processor in this undertaking. To remedy this taxation, the timestamps mechanism tries to mitigate both performance and energy burdens during the verification phase. The Timestamp Cache is a much more energy efficient data cache, whose configuration is set to optimum, by its execution of a series of performance simulations. This approach requires the memory to be modified to store the timestamps. This modification is transparent to the processor.

5.3. Experimental Evaluation

This section details about the simulation framework/test bench and specifies the configurations needed to arrive at the given results. It then presents the baseline results illustrating the performance of traditional Merkle hash tree verification followed by the performance of the proposed TSV scheme. Finally the performance analysis section concludes with an elaboration of both the baseline and TSV energy consumption results, in comparison.

5.3.1. Simulation Framework

The simulation framework is based on Simplescalar Tool Set [4], which is configured to execute ARM binaries. With this dissertation having for its main goal to demonstrate the energy efficiency of the proposed memory integrity verification mechanism in embedded systems, what has been used, that best replicates the variety of practical applications run on embedded devices, is MiBench [23] embedded benchmark suite. Here, the results obtained from different benchmark programs are presented to demonstrate that the efficiency of the proposed TSV mechanism is thorough. All the simulations performed are cache based, and employ the sim-cache simulator in simplescalar. The cache configurations used for the simulations, presented in Table 5.1, have the configuration of the Level 1 Data Cache optimally chosen to complement the typical configurations of an Embedded ARM processor [3]. Moreover, the TSV mechanism is straightly implemented for data only and hence neither to its instructions or to the simulation's I-cache is applied any emphasis.

Table 5.1. Cache Configurations

Cache	Specifications
L1-D Cache	4KB, 1-way, 32B Line
L2-D Cache	None

5.3.2. Baseline Energy Consumption

The adoption of the Merkle hash tree in MIV leads to excesses in energy consumed. To measure this computation, an MIV architecture has been proposed in Section 3.5. Here the algorithm is wielded to compute and count the number of hash invocations required per data miss in the Level 1 Data Cache, in order to verify data integrity. Hence, given that the energy consumption per hash invocation is known, the results obtained from the algorithm can serve to calculate the total energy consumption of the MIV mechanism. For comparison purposes, the configuration of hash cache is left akin to that of the L1 Data cache. Therefore, the hash cache is 1 way associative i.e. direct mapped cache. At this point it should be noted that the number of hash invocations grows incrementally as onward proceeds the verification from the first level to the root level.

Table 5.2 shows a relationship between the DL1 misses and Hash verification at each level, for 14 embedded benchmark applications. The Merkle hash tree constructed is a $4-ary$ hash tree with 14 levels. The hash level (HL) indicates the number of times the hash verification was invoked. Here the total DL1 misses are distributed amongst 14 hash levels to indicate in which level the miss was verified. Hence in general, the total DL1 misses is equal to the summation of misses verified at each hash level as given in equation 11. Here n

Table 5.2. Hash Accesses at different Levels

Benchmarks	HL1	HL2	HL3	HL4	HL5	HL6	HL7	HL8	HL9	HL10	HL11	HL12	HL13	HL14
dijkstra_small	93	46	626	0	2	0	0	11	0	3	1	0	8	1500344
jpeg_large	65	35	56	10	0	0	0	0	0	0	7	0	0	2923220
lame_large	66	21	44	12	15	6	0	1	0	0	0	4	1	39221183
lame_small	66	21	44	12	15	6	0	1	0	0	0	4	1	3135942
patricia_large	130	19	9	4	1	0	0	0	0	0	6	8	0	9137840
patricia_small	130	19	9	4	1	0	0	0	0	0	6	8	0	1529667
qsort_small	61	14	21	18	0	0	121	0	0	0	0	0	8	1876005
math_large	74	30735	18	7	7	0	0	0	0	0	0	12	0	1287768
sha_large	31556	18	94	3	2	0	0	1	0	0	4	0	2	232566
sha_small	2950	18	94	3	2	0	0	1	0	0	4	0	2	23269
stringsearch_small	125	140	21	0	0	0	0	3	0	0	1	198	0	1756
bitcount_large	113	20	37	3	1	0	0	0	0	0	0	0	0	719
bitcount_small	112	21	38	3	1	0	0	0	0	0	0	0	0	706
dijkstra_large	93	46	626	0	2	0	0	11	0	3	1	0	8	7020003

represents maximum Hash level.

$$\text{Total DL1 Misses} = \sum_{i=1}^{n} \text{Misses at each HL}_i \tag{11}$$

For example, in the case of bitcount_large application, there are a total of 893 misses in DL1 cache ($= HL1 + HL1 + HL2 + ... + Hl14$). Out of these misses, 113 are verified in Level 1. The number of hash invocation in Level 1 are $113 * 1 = 113$. Similarly, 719 misses are verified in Level 14 accounting for $719 * 14 = 10066$ hash invocations. What this brings to bear is that the total number of hash invocations for a particular benchmark can be calculated using Equation 12.

$$\text{Total Hash Invocations} = \sum_{i=1}^{n} \text{Misses at each HL}_i \times i \tag{12}$$

This can then be related to average energy consumption of the integrity verification hash function per invocation to calculate the total energy consumption of the integrity verification module as given in Equation 13.

$$\begin{aligned}\textsf{Total Energy} =& \textsf{Total Hash invocations} \quad \times \\ & \textsf{Energy per invocation}\end{aligned} \tag{13}$$

For instance, from Table 3.2, if it is assumed that the energy consumed per hash invocation by the SHA-1 algorithm is 0.76 μJ, then for *bitcount_large* application, the energy consumption for hash verification will be $0.76 * 10066 = 7.65mJ$ at Level 14 alone. Hence the total energy consumption arrws over to the addition of energy consumption at each hash level. From this discussion, it is evident that as the verification process traverses up the levels, the energy consumption increases rapidly. Moreover, for all the applications in the above simulation, a majority of the DL1 misses are verified in the last level-14, thus consuming the maximum energy possible. These statistics render themselves as a baseline for comparisons with the proposed Timestamps Verification mechanism.

5.3.3. TSV Energy Consumption

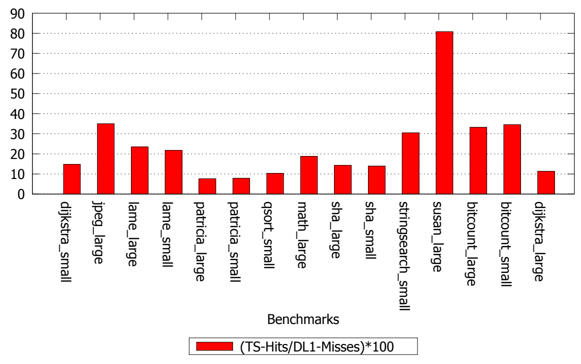
In the timestamps mechanism, a TS cache is created to store unique timestamps associated with each memory block. This timestamp is later taken by the processor to verify integrity of the block, before reverting to the orthodox approach of a Merkle hash tree. The timestamps generated are, as far as size, small and so the size of the TS Cache is likewise small, as compared to the DL1 cache. This is a significant advantage for embedded systems that are stringent with size and energy requirements. The timestamps stored in the TS Cache may possess 8 or 16 Bytes. Therefore, the TS Cache size may vary depending on the size of the timestamps. The performance of the TS cache is analyzed depending on the number of timestamps it can store. These can be 8, 16, 32, and 64. For each type, four separate configurations are analyzed, based on the number of ways and number of sets in the TS cache. Various configurations are detailed in Table 5.3. The TS cache size is equal to the product of the number of timestamps stored in the cache and the size of each timestamp. Hence assuming the size of timestamps is 8 Bytes, the size of TS cache of 8 timestamps is

64 Bytes. Similarly the size of TS cache storing 16, 32 and 64 timestamps is 128, 256 and 512 Bytes, respectively.

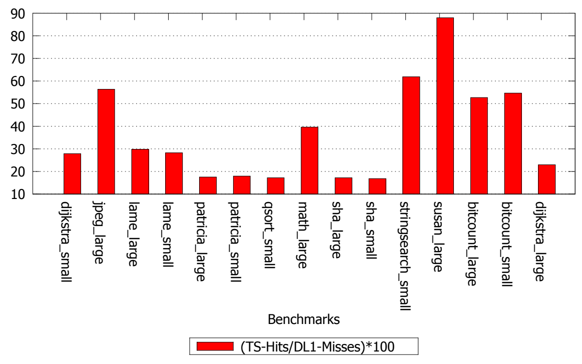
Table 5.3. TS Cache Configurations

# Timestamps stored in TS Cache	Configurations
8	1 Set-8 Ways; 2 Sets-4 Ways; 4 Sets-2 Ways; 8 Sets-1 Way
16	1 Set-16 Ways; 2 Sets-8 Ways; 4 Sets-4 Ways; 8 Sets-2 Way
32	1 Set-32 Ways; 2 Sets-16 Ways; 4 Sets-8 Ways; 8 Sets-4 Way
64	1 Set-64 Ways; 2 Sets-32 Ways; 4 Sets-16 Ways; 8 Sets-8 Way

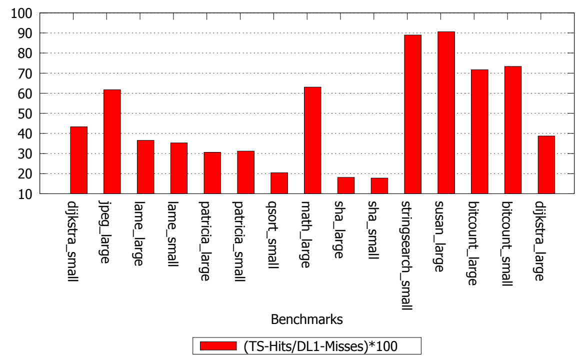
For each TS cache capable of storing different number of timestamps, detailed performance results have been presented along with their relative performance. Figure 5.3a, shows the percentage of TS hits w.r.t the DL1 misses for a TS cache capable of storing 8 timestamps, for 15 embedded benchmark applications. To revisit the discussion in Section 5.2, a hit in TS cache reduces the energy consumed by the integrity verification as the values in the TS cache can be trusted, without any Merkle hash tree verification required. In this case, the percentage of TS hits stands highest for benchmark application - *susan*, with almost 81% and the least for benchmark application *patricia*, with almost 8%. The average percentage of TS hits = 36.5%. Figure 5.3b, reveals the percentage of TS Hits w.r.t the DL1 misses for a TS cache capable of storing 16 timestamps. Here, the percentage of TS hits is highest for benchmark application - *susan*, with 88% and the least for benchmark application *patricia*, with almost 17%. The average percentage of TS hits is almost 56%. Figure 5.3c, shows the percentage of TS hits w.r.t the DL1 misses for a TS cache able to store 32 timestamps. Here, the percentage of TS hits is highest for benchmark application - *susan*, with 91% and the least for benchmark application *sha* with almost 18%. The average percent of TS hits is almost 71%. At the last, Figure 5.3d shows the percentage of TS hits w.r.t the DL1 misses for a TS cache of 64 timestamps. Here, the percentage of TS hits is highest for benchmark application - *stringsearch*, with 96%, and the least for benchmark application *sha*, with almost 18%. The average percentage of TS hits is almost 77%.



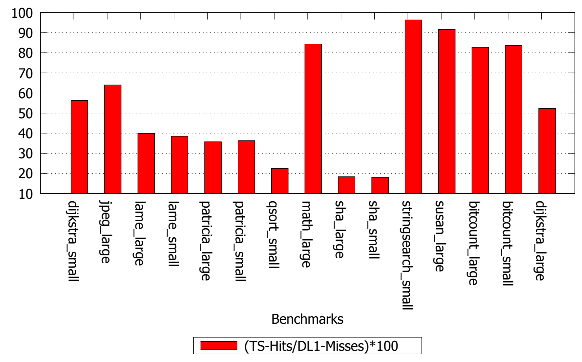
(A) Average percentage of TS Hits for TS Cache with 8 Timestamps



(B) For 16 Timestamps



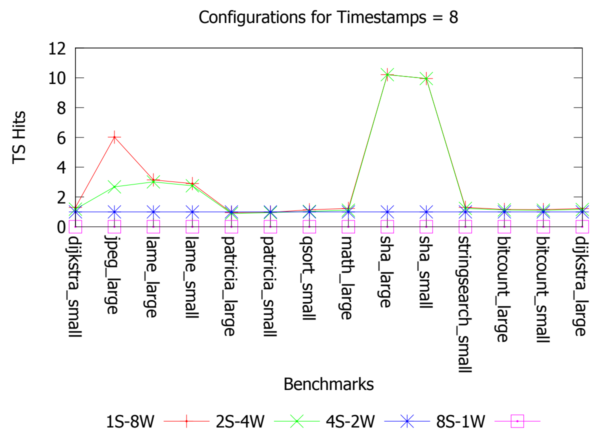
(C) For 32 Timestamps



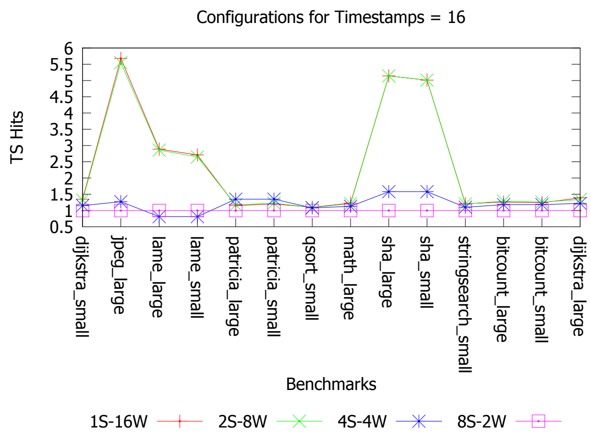
(D) For 64 Timestamps

FIGURE 5.3. Average percentage of TS Hits

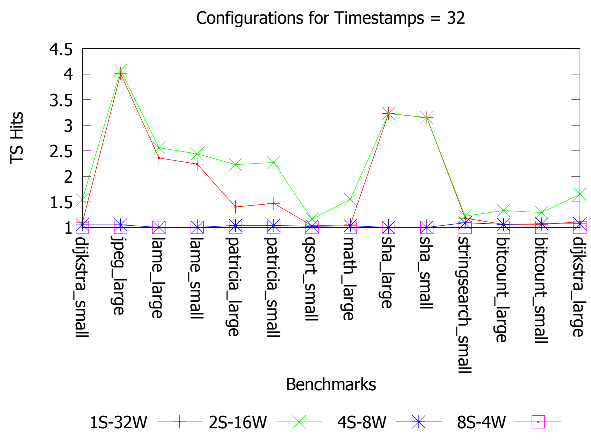
The results about the four individual configurations of each timestamp - 8, 16, 32 and 64 are presented. These spell out a generality of which configuration could offer the best energy savings for a particular TS cache. Figure 5.4a, shows the relative performance of the 4 possible configurations of TS cache made to store 8 timestamps. Here, the configuration 1Sets-8Ways results in higher TS hits and thus dramatic energy savings for the majority of the benchmark applications. For most of the benchmarks, the configuration 2Set-4Ways yields similar TS hits as compared to 1Sets-8Ways. Contrarily, the configuration 8Sets-1Ways results in no TS hits and thus no energy savings for any benchmark applications. The TS hits abound more for configuration 2Sets-4Ways than 4Sets-2Ways. Therefore it can be said that the impact of number of ways on TS hits is more than number of sets in the



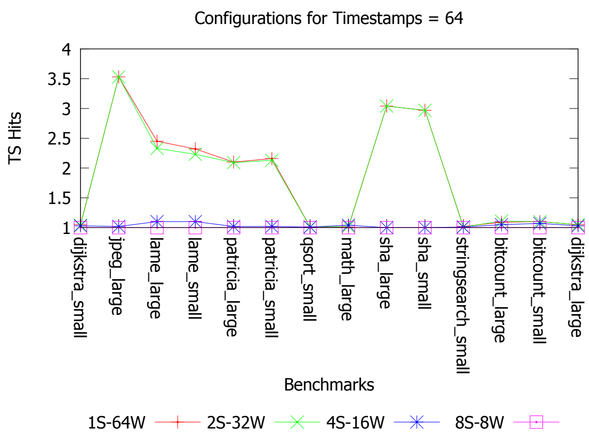
(A) Relative performance of four configurations for TS cache with 8 timestamps



(B) For 16 timestamps



(C) For 32 timestamps



(D) For 64 timestamps

FIGURE 5.4. Relative performance of various TS cache configurations

case of the TS cache with 8 timestamps. The TS hits and in turn energy savings go up with a rise in number of ways.

Figure 5.4b, shows the relative performance of the 4 configurations of TS cache capable of storing 16 timestamps. Here, the configurations 1Sets-16Ways and 2Sets-8Ways results in the highest TS hits and thus the highest energy savings for most of the benchmark applications, whereas, the configuration 8Sets-2Ways results in the least TS hits for most of the benchmark applications. Here again, the TS hits are more with an increase in the number of ways. Figure 5.4c indicates, the relative performance of the 4 configurations of TS cache capable of storing 32 timestamps. Here again, the configurations 1Sets-32Ways and 2Sets-16Ways result in the highest TS hits and thus highest energy savings for most of the benchmark applications. On the contrary, the configurations 4Sets-8Ways and 8Sets-4Ways yield similar results with lower TS hits for most of the benchmark applications. For the benchmark application of math_large, the 3 configurations of 2Sets-16Ways, 4Sets-8Ways and 8Sets-4Ways result in similar TS hits. Therefore again in this case, the TS hits occur more with an increase in the number of ways. Figure 5.4d, shows the relative performance of the 4 configurations of TS cache capable of storing 64 timestamps. Here, the configurations 1Sets-64Ways and 2Sets-32Ways provide similar results with higher TS hits whereas the configurations 4Sets-16Ways and 8Sets-8Ways provide similar results with lower TS hits. For the benchmark application math_large, all the 4 configurations yield similar results. Here again, the TS hits tally higher when put with an increase in the number of ways.

Based on the discussion in Sections 3.5 and 5.3.2, the percentage of TS hits are directly related to the energy savings. Recall, that for each miss in the DL1 cache, an amplitude of energy is consumed in re-constructing the Merkle hash tree and verifying the root hash. But in the case of Timestamp mechanism, for every hit in the TS cache corresponding to a DL1 miss, to spend any energy on Merkle hash verification becomes unneeded. Hence these hits directly relate to energy savings in an embedded system. At this point, it is important to emphasize that an increasing amount of energy is lost during Merkle hash verification as it moves from level 1 to level 14. To show this effect, the hash invocations at each level are computed, for TS cache storing 8, 16, 32 and 64 timestamps using the same approach described in Section 5.3.2. This is used to calculate the weighted averages of all the benchmarks in the timestamps simulation. The geometric mean of these weighted averages for TS = 8, 16, 32 and 64 is compared with that of basecase simulations. This is shown in Table 5.4. Here, the geometric mean of weighted averages of hash invocations steadily decreases for timestamps configurations as compared to basecase simulations. Thus fewer invocations means bigger energy savings.

TABLE 5.4. Geometric Mean of Weighted Averages of Hash Invocations

Basecase	TS = 8	TS = 16	TS = 32	TS = 64
48672.78	30537.76	27925.43	10106.65	7674.06

Using the equation 13 and the values in Table 5.4, we present a synopsis of average energy savings in the TSV approach when compared to basecase simulations. The energy savings for TS cache with 8 timestamps comes in at the atleast, at 36% and that for TS cache with 64 timestamps comes up with the most, at 81%, as shown in Table 5.5. The energy savings vouchsafed here are the averages of all the configurations in a particular TS cache. Section 5.3.5 analyzes the energy savings in each configuration and its impact on the entire system. Also, it is important to emphasize that the goal of this research is to present a variety of options for using the TSV mechanism instead of just suggesting/exhibiting the best option. It ultimately lies with the chip designer to evaluate all the possible options and each of their impacts on the system before selecting the most suitable option.

TABLE 5.5. Energy Savings in Timestamps Approach

Timestamps (TS Cache)	Energy Savings
8 timestamps	36%
16 timestamps	62%
32 timestamps	73%
64 timestamps	81%

5.3.4. Theoretical Evaluation for TSV Mechanism

In this section, we present a theoretical basis for the TSV mechanism and aim at theoretically justifying our results. Much prior research [55, 2, 60, 65, 66] has focused on modeling caches and their performance. We leverage this body of research to give here a model that encapsulates TS cache and its performance. In the same vein as the cache organization Agrawal et.al [2] describe, TS cache organization - TS_C, is denoted as (S, A, B), where S is the number of sets, A is the degree of associativity and B is block size. The TS cache size in bytes is the product of number of sets, degree of associativity and the block size. The number of blocks (i.e. the timestamps) and each block size (i.e. the timestamp size) is fixed in the TSV mechanism. Therefore the number of sets and the degree of associativity manifests the working set of the TS cache. Further, analysis is presented to show how altering the size of the working set influences the energy savings of the TS cache.

If considering the TS cache, the working set is dependent on the possible combinations of the number of sets and the degree of associativity. This is represented by the TS cache configurations. Also, in the TSV mechanism, the time factor in the working set model is represented by the total number of accesses. It is important to note that the TS cache configurations resemble any general purpose cache configurations - Direct Mapped, Fully Associative and Set Associative. For instance 1 sets - 8 ways, 1 sets - 16 ways, 1 sets - 32 ways, 1 sets - 64 ways resemble a fully associative cache, whereas, 8 sets - 1 ways resembles a direct mapped cache with the rest as set associative.

The Independent Reference Model [55] was proposed by Rao to analyze the performance of a cache. This model is analytically tractable and presents miss rate estimates for direct-mapped, fully-associative, and set-associative caches using the arithmetic and geometric distributions for page reference probabilities. The effectiveness of the TSV mechanism is analyzed by the energy savings offered by the TS cache. The energy savings correspond with TS cache hit rate (or 1 - TS cache miss rate). Therefore the Independent Reference Model serves best to analyze the TS cache performance in terms of its miss rate for varying TS cache size.

With reference to [55], the miss rate or the fault rate in a cache is written as

$$\mathsf{F}_f = \sum_{t=1}^{n} \mathsf{p}_t \mathsf{q}_t \tag{14}$$

where F is the fault rate, f is the replacement policy, n is the logical pages in the backing store, p_t is the page reference probability and q_t is the probability of not finding a page in the cache.

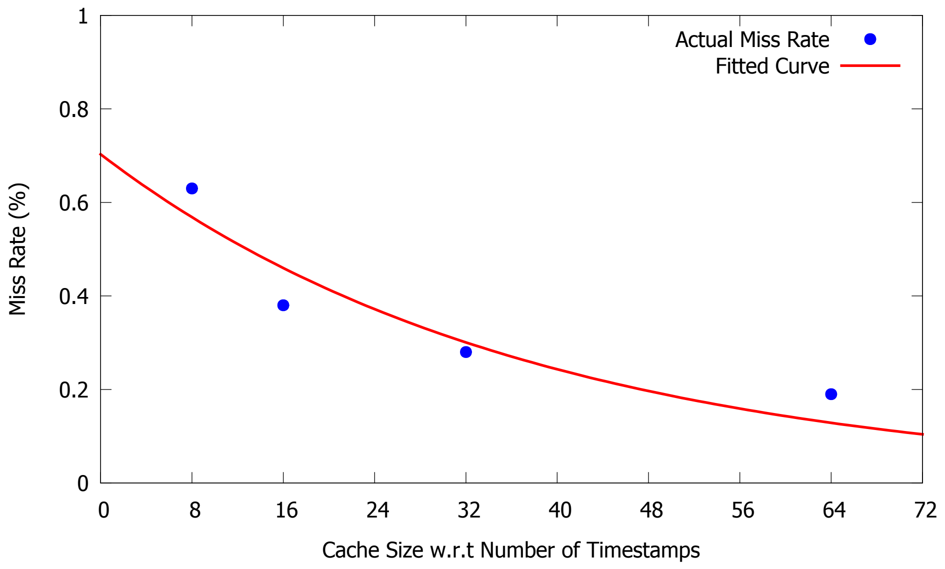


FIGURE 5.5. Miss Rate vs. TS Cache Size in terms of Number of Timestamps

If a graph of the fault rate/miss rate is to be plotted versus the cache size, then what is observed is that the fault rate decreases exponentially as increase the cache size. This trend is also exhibited in the TS cache and appears here in Figure 5.5, which represents a graph of miss rate vs. TS cache size in terms of number of timestamps. Since the block size is kept constant at 8 bytes, the TS cache climbs from 64 Bytes for 8 timestamps to 512 Bytes for 64 timestamps. The miss rate depicted (5.5) is calculated by averaging the miss rates obtained from all the configurations i.e. working sets in a particular TS cache. It is therefore evident that the miss rate decreases exponentially as increases the size of the the TS cache.

5.3.5. Overhead Evaluation of TSV Mechanism

The proposed timestamps mechanism does not mandate a unique timestamp to be stored in the TS cache to serve readily all the data blocks accessed by the L1 Data Cache. An L1 Data cache of size 4KB with 32 Byte block size contains a total of 128 cache blocks. However, the TS cache is configured to house only 8, 16, 32 or 64 timestamp blocks. More-

Table 5.6. Energy Savings vs. Area Overhead in TS Cache

Number of Timestamps	Size of TS Cache with Block Size = 8 Bytes	TS Cache Configurations	Energy Savings (%)	TS Cache Area Overhead w.r.t L1 Data Cache of size 4KB (%)
8	64	1 Sets-8 Ways	56.92	1.56
		2 Sets-4 Ways	53.40	
		4 Sets-2 Ways	33.11	
		8 Sets-1 Ways	0.00	
16	128	1 Sets-16 Ways	74.10	3.12
		2 Sets-8 Ways	73.94	
		4 Sets-4 Ways	53.23	
		8 Sets-2 Ways	46.91	
32	256	1 Sets-32 Ways	83.50	6.25
		2 Sets-16 Ways	89.57	
		4 Sets-8 Ways	59.78	
		8 Sets-4 Ways	57.46	
64	512	1 Sets-64 Ways	92.36	12.50
		2 Sets-32 Ways	92.10	
		4 Sets-16 Ways	70.02	
		8 Sets-8 Ways	68.84	

over, a timestamp is unique data pertaining to a particular cache block and hence its block size is significantly smaller than that of the cache block. The timestamp block size can be either 8 Bytes or 16 Bytes as compared to 32 Bytes in a L1 Data cache. In this research, the timestamp block size is set to 8 Bytes. Table 5.6 below presents an analogy between the size of the TS cache, its energy savings and its area overhead with respect to L1 Data cache of size 4KB. The area overhead is calculated using the equation 15 below. For simplicity, we have ignored the size of the cache tag in both L1 Data Cache and TS Cache configuration. Nonetheless, it is important to stress that the size of tag will be higher in L1 Data Cache as compared to that in the TS Cache. Thus the area overhead will be even less in practical scenario.

$$\text{TS Cache Area Overhead} = \frac{\text{Area of TS Cache}}{\text{Area of L1 Data Cache}} \times 100 \tag{15}$$

Since the size of each TS block is fixed at 8 Bytes, the size (in bytes) of the TS cache storing 8, 16, 32 and 64 timestamps is 64, 128, 256 and 512 respectively. For each TS cache mentioned above, the energy savings (in percentage) for every configuration and its area overhead (in percentage) is presented. The area overhead for the TS cache of 64, 128, 256 and 512 bytes is 1.56%, 3.12%, 6.23% and 12.50%, respectively. Therefore the overhead of the TS cache is significantly low as compared to the energy savings it can offer.

The energy savings of TS cache with 8 timestamps is in the range of 33.11% to 56.92% with an area overhead of 1.56%. The energy savings of TS cache with 16 timestamps falls in the range of 46.91% to 74.10% with an area overhead of 3.12%. The energy savings of TS cache with 32 timestamps is in the range of 57.46% to 89.57% with an area overhead of 6.25%. And finally the energy savings of TS cache with 64 timestamps is in the range of 68.84% to 92.36% with an area overhead of 12.5%. Importantly, we here emphasize that the affect of number of ways or associativity on the energy savings is profoundly higher than that of number of sets. Energy savings only decrease as the associativity decreases. This attribute is expected as a general characteristic in caches.

To stress the importance of the energy savings offered by the proposed TSV mechanism and to provide a comparison between the configurations of various TS caches, we calculate a new parameter — the Energy Savings/Area Overhead factor, in short the E/O factor. The value of the E/O factor stands in the range of 0% to 100% and it should ideally be as high as possible. Table 5.7, shows the E/O factor for all the TS cache configurations. Where the E/O factor is the highest is for the TS cache configuration of 1 sets-8 ways (Fully Associative), at 36.49%, and where the E/O factor is lowest is for TS cache configuration of 8 sets-1 ways (Direct Mapped), at 0%. This nonce reading of 0% indicates that this configuration did not receive TS hits and consequently resulted in no energy savings. But if this configuration is ignored, the E/O factor decreases steadily with 5.51% as the lowest for the

Table 5.7. Energy Savings/Area Overhead (E/O) factor for TS Cache Configurations

TS Cache Configurations	E/O Factor (%)
1 Sets-8 Ways	36.49
2 Sets-4 Ways	34.23
4 Sets-2 Ways	21.23
8 Sets-1 Ways	0.00
1 Sets-16 Ways	23.75
2 Sets-8 Ways	23.70
4 Sets-4 Ways	17.06
8 Sets-2 Ways	15.03
1 Sets-32 Ways	13.36
2 Sets-16 Ways	14.33
4 Sets-8 Ways	9.57
8 Sets-4 Ways	9.19
1 Sets-64 Ways	7.39
2 Sets-32 Ways	7.37
4 Sets-16 Ways	5.60
8 Sets-8 Ways	5.51

TS cache configuration of 8 sets-8 ways. This trend implies that the energy savings offered by the TS cache does not increase at the same rate at which its area overhead does. Since area overhead bears a greater impact, it gradually pulls down the E/O factor. Figure 5.6 presents the average E/O factor as well as the lower and upper bounds for TS cache with 8, 16, 32 and 64 timestamps. The fitted curve shows that the average E/O factor decreases exponentially as increases the TS cache size. The fluctuation in the upper and lower bounds, also known as the error fluctuation, falls as the TS cache increases. This trend points to the Law of Diminishing Returns. It is important to stress that as the TS cache size increases, the impact of associativity on the energy savings decreases. Accordingly, for a smaller TS cache, this trend possesses importance. It is key to select the fully associative cache configuration to arrive higher energy savings. Notwithstanding, for a larger TS cache, even the direct mapped configuration could yield savings in energy comparable to that gained by the fully associative cache configuration.

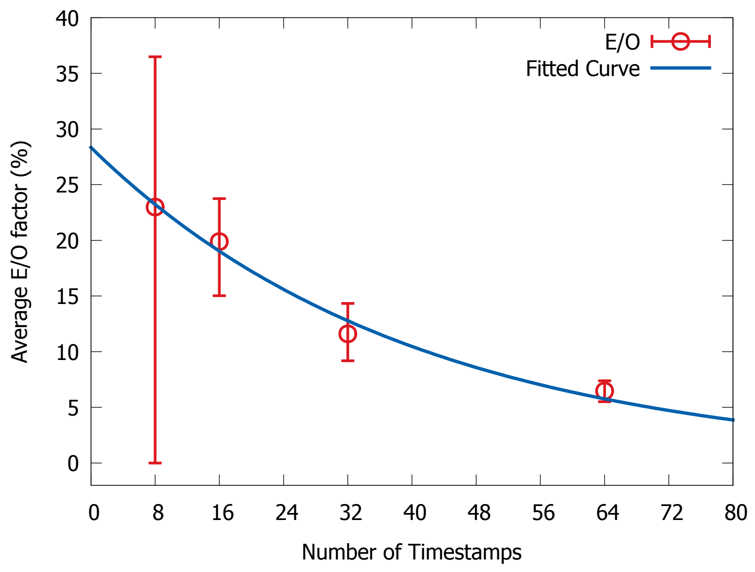


FIGURE 5.6. Average E/O factor for each TS cache

5.4. Conclusions

This chapter presents a novel mechanism of Timestamps Verification - TSV. This scheme is based on the principle of locality, also known as the working set of the program in execution. TSV uses timestamps to take advantage of the locality principle and reduce the energy consumption of the Merkle hash tree during the phase of integrity verification. The simulation results show that the energy savings with TSV mechanism can range from 36% to 81%, compared to baseline results. We have also detailed here our theoretical analysis to prove the simulation results. What we have done last, we have also presented the E/O factor to consider the area overhead imposed by the Timestamp Cache.

CHAPTER 6

HASH FUNCTION-LESS MEMORY INTEGRITY VERIFICATION

6.1. Introduction

Secure architectures employ both encryption and memory integrity verification, as protecting integrity without confidentiality does not provide any security. A cryptographically secure encryption function, by definition, is a pseudo-random function [30]. Any modification to the ciphertext C, resulting in C', corresponding to a message M ($C = E_k(M)$) will cause every bit in the decrypted message $M' = D_k(C')$ to flip with a probability of $\frac{1}{2}$. Another distinctive characteristic of memory integrity problem is that the sender is always the receiver. In other words, processor which writes a value in the memory is the ultimate consumer of the value. This allows the processor to store some partial information about the message (data to be stored in the memory) and use the partial information to verify the integrity.

The rest of this chapter is organized as follows. Section 6.2 describes the architecture and the overall functioning of the Hash Function-Less Memory Integrity Verification mechanism. Section 6.3 analyses the performance of Hash Function-Less Memory Integrity Verification mechanism and Section 6.4 presents the Conclusion.

6.2. Hash Function-Less MIV Architecture

The overall architecture is shown in Figure 6.1. In the figure the function that extracts partial information is referred as f_p. For example, for a 256-byte block a simple partial extraction function could be extracting every 16^{th} bit to extract 128-bit partial information. The security of the solution is *independent* of the partial extraction function, as encryption function should affect every bit equally likely. This partial information is stored in a dedicated cache known as the f_p cache. Such an operation creates a tree of partial data known as the Merkle f_p tree, as shown in Figure 6.3. The root of this tree is stored in the f_p cache whereas the rest of the encrypted tree values are stored in the off-chip untrusted memory.

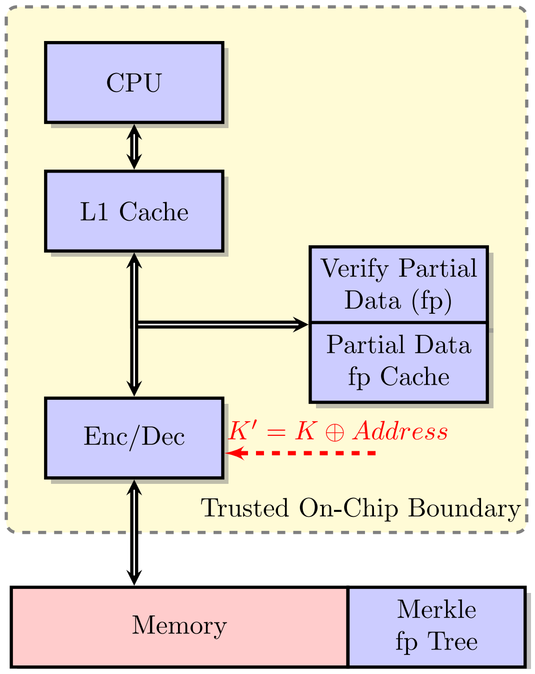
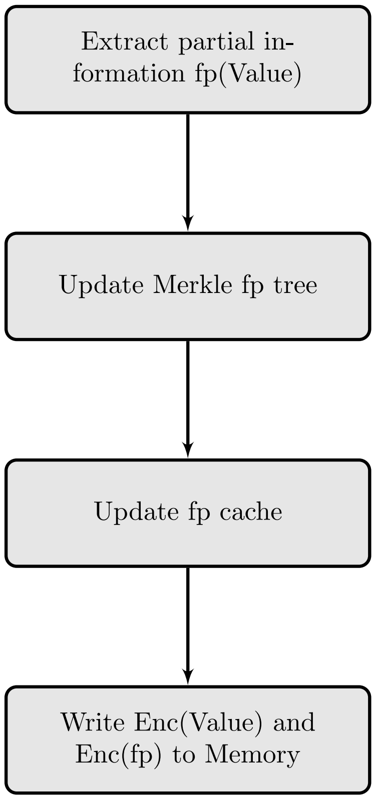


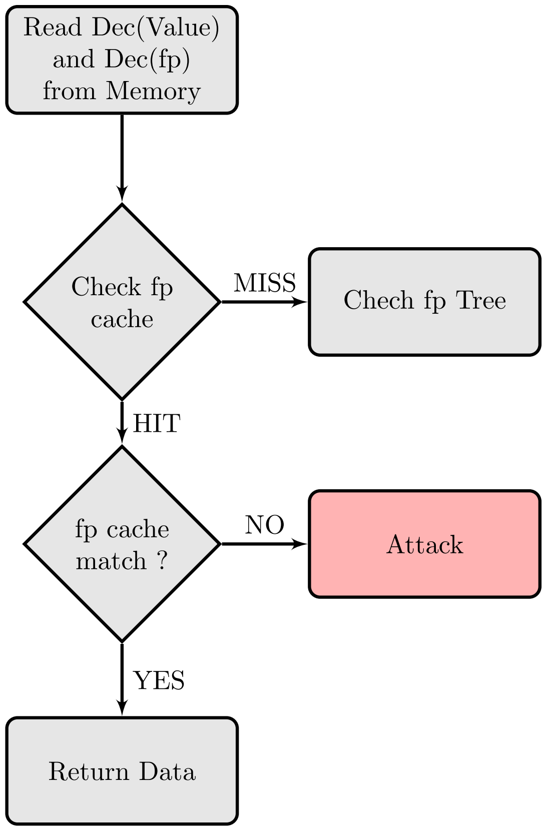
FIGURE 6.1. Hash Function-Less Verification Architecture

The primary difference between a hash tree and the tree of partial values is that the partial values are encrypted and stored in the memory. In our earlier work [21] we have proven that, in such a setup, security against integrity attacks can be guaranteed only with collision resistance. Since the value is stored encrypted on memory, the pseudorandomness property of the encryption function provides the necessary collision resistance. The solution is to build a tree using partial information and store the root on-chip. Hence this will provide security against splicing and spoofing attacks. However, since there is no notion of time in this setup, a potentially powerful adversary could perform a replay attack. In such an attack, a memory block located at a given address is recorded and inserted at the same address at a later point in time. Thus the processor is made to process a stale value instead of the most current one. To prevent this attack and to add a notion of time to the memory address and value, the encryption function key (K) is combined with the memory block address – $K' = K \oplus Address$. Any change in the memory address and its associated value should be detected by the encryption function and hence should protect against the replay attack.

The memory write and read operations are shown in Figures 6.2a and 6.2b respec-


Extract partial information fp(Value)
Update Merkle fp tree
Update fp cache
Write Enc(Value) and Enc(fp) to Memory

(A) Write Operation



(B) Memory Read

FIGURE 6.2. Memory Operations in Hash Function-Less Verification Architecture

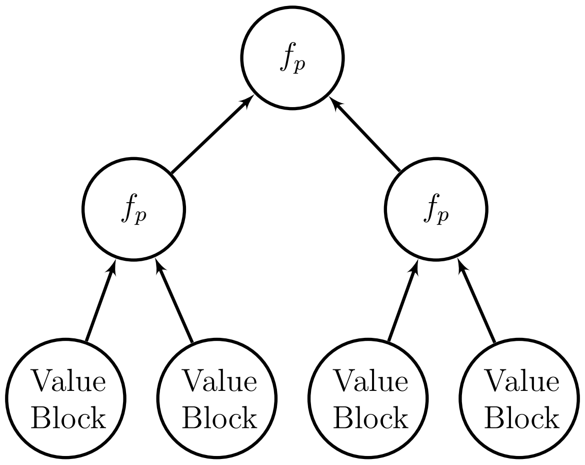


FIGURE 6.3. Merkle f_p Tree

tively. During a write operation, the processor first extracts the partial data using the partial extraction function. It then updates the merkle f_p tree and the root value in the f_p cache. Finally, it writes the encrypted data and encrypted partial data value to the off-chip memory. During a read operation, the data value and partial data values are first decrypted. The processor then checks the f_p cache to determine if the partial data value is already present in the cache. If the value is not present i.e. if there is cache miss, then it re-computes the entire tree for verification. In an event where the value is present i.e. if there is cache hit, then the processor compares the two values. A match of two values proves that there was no tampering done to the data and so the processor continues its operation. On the contrary, if the values do not match, it is concluded that there was an attack on the system and the processor aborts further operation on that data.

LEMMA 6.1. *The proposed hash-function less memory integrity scheme detects replay, splicing, and spoofing attack on memory with probability* $1 - \frac{1}{2^{128}}$.

PROOF. Consider two data blocks D_A and D_B stored at addresses A and B respectively. Let the encrypted values stored in the memory by C_A and C_B and their partial values are f_{p_A} and f_{p_B} respectively.

Splicing attack: Splicing attack changes the association of addresses and data. The decrypted value after a read from address A would be $D'_A = D_{K \oplus A}(C_B)$. Since the encryption function is a pseudorandom function the change in the key and the data value will result in pseudorandom output. Thus $P[f'_{p_A} = f_{p_A}] = \frac{1}{2^{128}}$. The attack succeeds if the partial data matches, which happens with probability $\frac{1}{2^{128}}$.

Spoofing attack: Spoofing attack randomly modifies the data. The decrypted value after a read from address A would be $D'_A = D_{K \oplus A}(C'_A)$. Since the encryption function is a pseudorandom function the change in the key and the data value will result in pseudorandom output. Thus $P[f'_{p_A} = f_{p_A}] = \frac{1}{2^{128}}$. The attack succeeds if the partial data matches, which happens with probability $\frac{1}{2^{128}}$.

Replay attack: Replay attack replaces the data in an address from its previous instance. The partial data are constructed as a merkle tree and their root is stored in the trusted boundary on-chip. This root $f_{p_{root}}$ will result in a different value when any of its leaf nodes are modified. Thus $P[f'_{p_{root}} = f_{p_{root}}] = \frac{1}{2^{128}}$. The attack succeeds if the partial data root matches, which happens with probability $\frac{1}{2^{128}}$. □

6.3. Experimental Evaluation

6.3.1. Simulation Framework

To measure the power dissipation in our research, we are using Sim-Panalyzer[32], a cycle accurate power simulator for ARM instruction set architecture. Specically, it simulates the StrongArm SA1100 processor and is widely used for power estimations in ARM processors. The latest version of Sim-Panalyzer provides very detailed power estimation models for various components on System-on-Chip (SoC). Some of the power models include - cache power model, datapath and execution unit power model, clock tree power model and I/O power model etc. Sim-Panalyzer computes the power dissipation in a program, based on counting the number of transitions in these parts of the processor. Since this research is primarily targeted towards embedded systems, we have used the MiBench[23], an industry-standard embedded benchmark suite. MiBench is divided into multiple classes which are representative of different embedded application domains such as automotive, consumer, networking and security etc. The results presented in rest of this section were obtained by running power simulations on these benchmark applications.

The processor specifications used in the simulation test bench are listed in Table 6.1. These specifications are mimicked to represent an ARM Cortex A9 embedded processor. The first set of results, shown in Figure 6.4, presents the average power (%) consumed by various components of an embedded processor. Here, the cache hierarchy consumes the most amount of power (75%) followed by the CLOCK (14%) and IRF (8%). Whereas, negligible power is consumed by Others (1%), constituted of the ALU, MULT, FPU, LOGIC and FPRP. These results form a base line for the MIV power dissipation results presented in the next section.

Table 6.1. Embedded Processor Specifications

Feature	Value	Property
Technology	0.18 μm CMOS	
Supply Voltage	1.2 V to 1.8 V	
Clock Frequency	233 MHz	
I-Cache	32 KB	4-Way Set Associative. Line Length = 8 words Block Size = 32 Bytes
D-Cache	32 KB	4-Way Set Associative. Line Length = 8 words Block Size = 32 Bytes
L2 Cache	512 KB	8-way set-associative
Instruction TLB	32 Entries	Fully Associative
Data TLB	32 Entries	Fully Associative

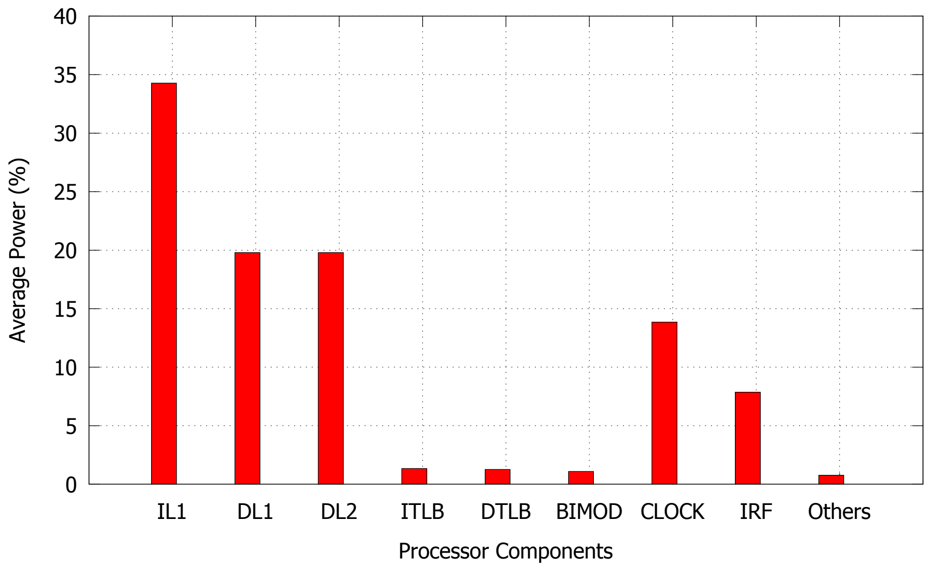


Figure 6.4. Power consumption in various components of a Processor

6.3.2. Power Dissipation in MIV

During the MIV process, there are two stages where power is consumed. First, when a hash function is invoked to create the hash of the data block and second, while caching some of these hashes in the on-chip memory. Hence, if $P_{HashFunc}$ resembles the power consumed by the hash function and $P_{HashCache}$ resembles the power consumed by the cache that stores these hashes, then the total power (P_T) consumed by the MIV process is:

$$\mathsf{P}_T = \mathsf{P}_{HashFunc} + \mathsf{P}_{HashCache} \tag{16}$$

$P_{HashFunc}$ can be calculated using the formula given in Equation 17.

$$\begin{aligned} \mathsf{P}_{HashFunc} &= \frac{\mathsf{E}_{HashFunc} \times \mathsf{H}_{SZ} \times \mathsf{H}_T \times \mathsf{C}_F}{\mathsf{S}_C} \\ \text{where } \mathsf{E}_{HashFunc} &= \text{Energy consumption per data bit} \\ \mathsf{H}_{SZ} &= \text{Size of Input to the Hash function} \\ \mathsf{H}_T &= \text{Total number of Hash function Invocations} \\ \mathsf{C}_F &= \text{Clock Frequency} \\ \mathsf{S}_C &= \text{Total Simulation cycles} \end{aligned} \tag{17}$$

As discussed in Section 3.4, during a Merkle hash tree based verification, hashes are generated from the leaf nodes and compared against the hashes that are present in the hash cache. For every hash cache miss, this process is repeated at all the levels of the hash tree up until the root hash is calculated. Thus if L represents the total number of levels in a Merkle hash tree and M represents the hash misses then H_T can be calculated using Equation 18.

$$H_T = \sum_{L=1}^{N} \text{Misses at each } \mathsf{M}_L \times L \tag{18}$$

We have simulated a hash cache structure using the hash address algorithm proposed in Section 3.5. This allows us to measure the hash cache misses at each level of the tree (M_L) so that we can calculate H_T. We rely on Sim-Panalyzer to provide the $P_{HashCache}$ value. In order to get the $E_{HashFunc}$, we have referred to the implementation of BLAKE [24], a cryptographic hash function that is one of the five finalist at the NIST SHA-3 competition.

In [33], the authors have analyzed the performance of all the round 2 candidates in the NIST SHA-3 competition. According to their results, BLAKE consumes 4.66 pJ/bit of dynamic energy i.e. $E_{HashFunc} = 4.66$ pJ/bit. The hash tree simulated in this research is a 4-ary tree with 14 levels and each data block as 32 bytes. Hence $H_Z = 32 \times 4 \times 8 = 1024$ bits. The value of C_F is 233 MHz while H_T and S_C can be calculated from the simulation results.

Using the equations, algorithm and parameter values mentioned above, we have measured the $P_{HashFunc}$ and $P_{HashCache}$ and shown in Figure 6.5. Here, $P_{HashFunc}$ is represented as HFunc and $P_{HashCache}$ is represented as HCache. As seen from the results, the hash function consumes 922.85 mW of power while the hash cache consumes 139.29 mW of power during the entire simulation cycle. This is significantly higher than the power consumed by rest of the processor components.

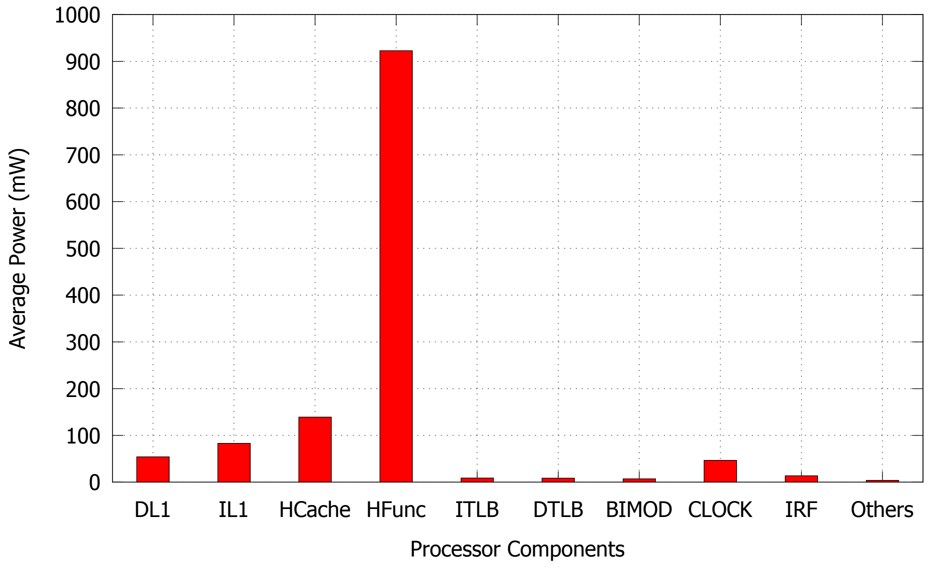


FIGURE 6.5. Power consumption of Hash Cache and Hash Function in a Processor

6.3.3. Power Savings in Hash Function-Less Verification

In the proposed Hash Function-Less MIV mechanism, we have completely eliminated the use of hash function to generate the hashes. Instead, we have proposed the use of an existing encryption function or a pseudo random function to generate the hashes. Since a hash function and a pseudo random function posses similar characteristics, the security of the system does not get comprised. To measure the power savings offered by our mechanism, we have measured the power consumed by an encryption function to compare against that of the hash function. In [36], the authors have analyzed the power consumption of AES-128 encrytpion function at 1.8 V input voltage and 0.18μm CMOS technology. Here, AES-128 consumes energy of 0.432 nJ/Byte. Similar to the MIV power dissipation in Section 6.3.2, we have calculated the power dissipation of AES-128 encryption function. At this point, it is important to stress two things. First, the number of encryption function accesses at level 1 will be required in regular encryption/decryption operation. Hence energy consumed at level 1 is not considered. The energy overhead occurs when the encryption function is accesses from levels 2 to 14. Second, the hash cache structure is required to store the partial data tree blocks. The Figure 6.6 shows that the average power consumed by AES-128 to generate and verify hashes is 185.66 mW as opposed to 922.85 mW required for BLAKE hash function.

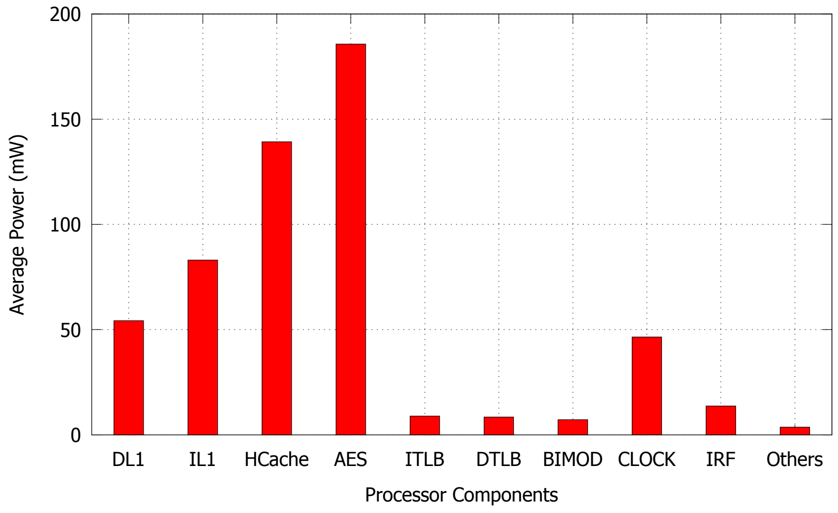


FIGURE 6.6. Power consumption of AES-128 Encryption Function in a Processor

The Table 6.2 shows that the total power consumed by an embedded processor with a hash function is 1287.8 mW whereas the total power consumption is 550.61 mW when an encryption function is used to perform the hash verification process. Thus, our proposed mechanism achieves a total power savings of 57.24%, which is significant in battery operated, energy constrained embedded systems.

TABLE 6.2. Total Power savings in an Embedded Processor

Total Power with Hash Function(mW)	Total Power with Encryption Function(mW)	Power Savings(%)
1287.80	550.61	57.24

6.4. Conclusion

This chapter presents the Hash Function-Less Memory Integrity Verification mechanism for embedded systems. The goal of this approach is to completely eliminate the use of a hash function during the process of memory integrity verification. Here, an encryption function can be used to perform memory integrity verification. Our cryptographic analysis proves that since the security properties of an encryption function are similar to that of a hash function, the overall security of the system is not compromised. The simulation results prove that the proposed mechanism leads to a total power savings of 57.24%, which can significantly improve the battery life of an embedded system.

CHAPTER 7

CONCLUSION

The modern day computing devices have pervaded in all facets of human life. They are empowered to store, track and relay essential data over the network. This data may often contain sensitive information that must be protected from leaking out to unwanted entities. Thus security becomes a primary concern. Now a days, the attacks may originate from a variety of sources and may evolve into various sophisticated forms, due to the computing power available to the hackers. These may be software based attacks or even physical attacks on the system. Physical attacks are more prevalent on embedded systems like smart cards, smart phones, PDAs, network sensors and so forth as they comparatively easily accessible. Thus the traditional software-only security solutions fail to provide adequate protection against these attacks. To achieve reliable and robust security, it is essential to have hardware support for security. Hardware support for security facilitates the construction of trustworthy secure systems. However, a significant disadvantage of hardware security mechanisms is that they require modification to the micro-architecture of the processor. This is an extremely expensive and time consuming process and cannot be conceived unless the security mechanisms are thoroughly tested. Another serious concern with hardware security is that they consume system resources, thereby causing a huge performance bottleneck. Owing to these concerns, the adoption of hardware security mechanisms is hampered in modern computing devices. With an emphasis on hardware security mechanisms, the objective of this dissertation is to propose solutions to answer the two concerns related to hardware security mechanisms.

Therefore, we first propose the Virtualization Based Secure Framework (vBASE) that takes advantage of the virtualization technology to realize hardware security architectures inside the virtualization layer. The vBASE framework has two forms: Testing and Execution framework, based on its mode of operation. The Testing framework serves as a generic platform to test the security implementations of hardware architectures. The Execution framework, constitutes the Secure Hypervisor (SecHYPE) framework and the Cloud Trust

(CTrust) architecture, which presents a security solution for cloud computing platforms. With the motivation of improving the performance of cryptographic memory integrity verification mechanism, we have proposed three novel mechanisms: Memory Detect and Protect mechanism (MEM-DnP), Timestamps Verification (TSV) mechanism, and Hash Function-Less Memory Integrity Verification mechanism. The MEM-DnP approach yields energy savings of approximately 86% to 99% during the integrity verification phase. The TSV mechanism offers energy savings in the range of 36% to 81% during the integrity verification phase. Finally, the Hash Function-Less verification reduces the power consumption of an embedded processor by 57.24%. These energy and power savings are significant, considering most embedded devices are battery powered.

BIBLIOGRAPHY

[1] A. Thakwani. Process-level Isolation using Virtualization. `http://repository.lib.ncsu.edu/ir/handle/1840.16/2031`, 2010.

[2] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7(2):184–215, May 1989.

[3] ARM Architecture Reference Manual. `http://www.altera.com/literature/third-party/ddi0100e_arm_arm.pdf/`, 2000.

[4] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59 –67, feb 2002.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[6] E. Barker, M. Smid, D. Branstad, and S. Chokhani. A framework for designing cryptographic key management systems. Technical report, National Institute of Standards and Technology - NIST, 2012.

[7] Manuel Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 90–99, 1991.

[8] R. Burch, F.N. Najm, P. Yang, and T.N. Trick. A monte carlo approach for power estimation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 1(1):63 –71, march 1993.

[9] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.

[10] C. Christian, K. Idit, and S. Alexander. Trusting the cloud. *SIGACT News*, 40:81–86, June 2009.

[11] C. Cowan, F. Wagle, Calton Pu, S. Beattie, and J. Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119 –129 vol.2, 2000.

[12] Cyber Security Watch Survey. `http://www.sei.cmu.edu/newsitems/cybersecurity_watch_survey_2011.cfm`, 2011.

[13] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[14] J. Dwoskin, M. Gomathisankaran, Y. Chen, and R. Lee. A framework for testing hardware-software security architectures. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 387–397, New York, NY, USA, 2010. ACM. Acceptance rate ¡ 20%.

[15] Eiman Elnahrawy and Badri Nath. Cleaning and querying noisy sensors. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, pages 78–87, New York, NY, USA, 2003. ACM.

[16] B. Gassend, G.E. Suh, D. Clarke, M. van Dijk, and S. Devadas. Caches and hash trees for efficient memory integrity verification. In *High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings. The Ninth International Symposium on*, pages 295 – 306, feb. 2003.

[17] C.H. Gebotys. Low energy security optimization in embedded cryptographic systems. In *Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on*, pages 224 – 229, sept. 2004.

[18] O. Gelbart, E. Leontie, B. Narahari, and R. Simha. Architectural support for securing application data in embedded systems. In *Electro/Information Technology, 2008. EIT 2008. IEEE International Conference on*, pages 19 –24, may 2008.

[19] D. Ghai, S.P. Mohanty, and E. Kougianos. Design of parasitic and process-variation

aware nano-cmos rf circuits: A vco case study. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(9):1339 –1342, sept. 2009.

[20] D. Ghai, S.P. Mohanty, and E. Kougianos. Variability-aware optimization of nano-cmos active pixel sensors using design and analysis of monte carlo experiments. In *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*, pages 172 –178, march 2009.

[21] M. Gomathisankaran and A. Tyagi. Relating boolean gate truth tables to one-way functions. In *Electro/Information Technology, 2008. EIT 2008. IEEE International Conference on*, pages 1 –6, may 2008.

[22] Mahadevan Gomathisankaran and Akhilesh Tyagi. Architecture Support for 3D Obfuscation. *IEEE Trans. Computers*, 55(5):497–507, 2006.

[23] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3 – 14, dec. 2001.

[24] L. Henzen, J.-P. Aumasson, W. Meier, and R.C.-W. Phan. Vlsi characterization of the cryptographic hash function blake. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(10):1746–1754, Oct 2011.

[25] Mei Hong, Hui Guo, and Sharon X. Hu. A cost-effective tag design for memory data authentication in embedded systems. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '12, pages 17–26, New York, NY, USA, 2012. ACM.

[26] C. Hunt and J. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[27] A. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy. Cloudsec: A security monitoring appliance for virtual machines in the iaas cloud model. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 113 –120, sept. 2011.

[28] Into the Cloud, Out of the Fog. `http://www.ey.com/Publication/vwLUAssets/`

`Into_the_cloud_out_of_the_fog-2011_GISS/$FILE/Into_the_cloud_out_of_the_fog-2011\%20GISS.pdf`, November 2011.

[29] J. Rutkowska. Qubes OS Architecture. `http://qubes-os.org/files/doc/arch-spec-0.3.pdf`, 2010.

[30] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.

[31] E. Keller, J. Szefer, J. Rexford, and R. Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA, 2010. ACM.

[32] Nam-Sung Kim, Taeho Kgil, V. Bertacco, T. Austin, and T. Mudge. Microarchitectural power modeling techniques for deep sub-micron microprocessors. In *Low Power Electronics and Design, 2004. ISLPED '04. Proceedings of the 2004 International Symposium on*, pages 212–217, 2004.

[33] M. Knezevic, K. Kobayashi, J. Ikegami, S. Matsuo, A. Satoh, U. Kocabas, Junfeng Fan, T. Katashita, T. Sugawara, K. Sakiyama, I. Verbauwhede, K. Ohta, N. Homma, and T. Aoki. Fair and consistent hardware evaluation of fourteen round two sha-3 candidates. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(5):827–840, May 2012.

[34] Srujan D. Kotikela, Satyajeet Nimgaonkar, and Mahadevan Gomathisankaran. Virtualization based secure execution and testing framework. In *Parallel and Distributed Computing and Systems*, pages 65–72, 2011.

[35] Elias Kougianos, Saraju P. Mohanty, and Rabi N. Mahapatra. Hardware assisted watermarking for multimedia. *Computers & Electrical Engineering*, 35(2):339 – 358, 2009. Circuits and Systems for Real-Time Security and Copyright Protection of Multimedia.

[36] H. Kuo, I. Verbauwhede, and P. Schaumont. A 2.29 gbits/sec, 56 mw non-pipelined rijndael aes encryption ic in a 1.8 v, 0.18 mu;m cmos technology. In *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*, pages 147–150, 2002.

[37] X. Li, C. Jiang, J. Li, and B. Li. Vminsight: Hardware virtualization-based process

security monitoring system. In *Network Computing and Information Security (NCIS), 2011 International Conference on*, volume 1, pages 62 –66, may 2011.

[38] David Lie, Chandramohan A. Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John C. Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.

[39] Q. Liu, C. Weng, M. Li, and Y. Luo. An in-vm measuring framework for increasing virtual machine security in clouds. *Security Privacy, IEEE*, 8(6):56 –62, nov.-dec. 2010.

[40] J. McCune, L. Yanlin, Q. Ning, Z. Zongwei, A. Datta, G. Virgil, and P. Adrian. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 143–158, Washington, DC, USA, 2010. IEEE Computer Society.

[41] R. McGowen, C.A. Poirier, C. Bostak, J. Ignowski, M. Millican, W.H. Parks, and S. Naffziger. Power and temperature control on a 90-nm itanium family processor. *Solid-State Circuits, IEEE Journal of*, 41(1):229 – 237, jan. 2006.

[42] Z. Min, Y. Miao, X. Mingyuan, L. Bingyu, Y. Peijie, G. Shang, Q. Zhengwei, L. Liang, C. Ying, and G. Haibing. Vasp: virtualization assisted security monitor for cross-platform protection. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 554–559, New York, NY, USA, 2011. ACM.

[43] Mobile and Smart Device Security Survey 2011. `http://images.tmcnet.com/tmc/whitepapers/documents/whitepapers/2011/4683-mobile-smart-device-security-survey-2011.pdf`, 2011.

[44] Saraju P. Mohanty. A secure digital camera architecture for integrated real-time digital rights management. *Journal of Systems Architecture*, 55(1012):468 – 480, 2009.

[45] Saraju P. Mohanty and Elias Kougianos. Simultaneous power fluctuation and average power minimization during nano-cmos behavioral synthesis. In *Proceedings of the 20th International Conference on VLSI Design*, pages 577–582, 2007.

[46] National Institute of Standards and Technology. *The NIST Definition of Cloud Computing*, nist special publication 800-145 edition, January 2011.

[47] S. Nimgaonkar and M. Gomathisankaran. Energy efficient memory authentication mechanism in embedded systems. In *Electronic System Design (ISED), 2011 International Symposium on*, pages 248–253, 2011.

[48] Satyajeet Nimgaonkar, Mahadevan Gomathisankaran, and Saraju P. Mohanty. Tsv: A novel energy efficient memory integrity verification scheme for embedded systems. *Journal of Systems Architecture*, 59(7):400 – 411, 2013.

[49] Satyajeet Nimgaonkar, Mahadevan Gomathisankaran, and SarajuP. Mohanty. Memdnpa novel energy efficient approach for memory integrity detection and protection in embedded systems. *Circuits, Systems, and Signal Processing*, pages 1–24, 2013.

[50] Satyajeet Nimgaonkar, Srujan Kotikela, and Mahadevan Gomathisankaran. Ctrust: A framework for secure and trustworthy application execution in cloud computing. In *Cyber Security (CyberSecurity), 2012 International Conference on*, pages 24–31, 2012.

[51] Satyajeet Nimgaonkar, Srujan Kotikela, and Mahadevan Gomathisankaran. Ctrust: A framework for secure and trustworthy application execution in cloud computing. *Academy of Science Journal*, pages 152–164, 2012.

[52] Dongkeun Oh, Nam Sung Kim, C.C.P. Chen, A. Davoodi, and Yu Hen Hu. Runtime temperature-based power estimation for optimizing throughput of thermal-constrained multi-core processors. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 593 –599, jan. 2010.

[53] Nachiketh R. Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Analyzing the energy consumption of security protocols. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 30–35, New York, NY, USA, 2003. ACM.

[54] R. Sailer and R. Perez and S. Berger and R. Cceres and L. Van Doorn. Towards Enterprise-level Security with Xen, 2006.

[55] Gururaj S. Rao. Performance analysis of cache memories. *J. ACM*, 25(3):378–395, July 1978.

[56] T. Ristenpart, E. Tromer, S. Hovav, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[57] A. Rogers, M. Milenkovic, and A. Milenkovic. A low overhead hardware technique for software integrity and confidentiality. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*, pages 113 –120, oct. 2007.

[58] Austin Rogers and Aleksandar Milenkovi. Security extensions for integrity and confidentiality in embedded processors. *Microprocessors and Microsystems*, 33(56):398 – 414, 2009.

[59] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic. Using address independent seed encryption and bonsai merkle trees to make secure processors os- and performance-friendly. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 183 –196, dec. 2007.

[60] E. Rothberg, J.P. Singh, and A. Gupta. Working sets, cache sizes, and node granularity issues for large-scale multiprocessors. In *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, pages 14 –25, may 1993.

[61] J. Sahoo, S. Mohapatra, and R. Lath. Virtualization: A survey on concepts, taxonomy and associated security issues. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222 –226, april 2010.

[62] *Secure Hash Standard.* National Institute of Standards and Technology, Washington, 2002. Federal Information Processing Standard 180-2.

[63] W. Shi, H.-H.S. Lee, M. Ghosh, and C. Lu. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Parallel Architecture and Compilation Techniques, 2004. PACT 2004. Proceedings. 13th International Conference on*, pages 123 – 134, sept.-3 oct. 2004.

[64] P. Siani. Taking account of privacy when designing cloud computing services. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, CLOUD '09, pages 44–52, Washington, DC, USA, 2009. IEEE Computer Society.

[65] A.J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *Software Engineering, IEEE Transactions on*, SE-4(2):121 – 130, march 1978.

[66] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, ICS '01, pages 1–12, New York, NY, USA, 2001. ACM.

[67] G.E. Suh, C.W. O'Donnell, and S. Devadas. Aegis: A single-chip secure processor. *Design Test of Computers, IEEE*, 24(6):570 –580, nov.-dec. 2007.

[68] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, C. Fernando, A. Andrew, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38:48–56, 2005.

[69] H. Vahedi, S. Gregori, Y. Zhanrong, and R. Muresan. Power-smart system-on-chip architecture for embedded cryptosystems. In *Hardware/Software Codesign and System Synthesis, 2005. CODES+ISSS '05. Third IEEE/ACM/IFIP International Conference on*, pages 184 –189, sept. 2005.

[70] W. Vogels. Beyond server consolidation. *Queue*, 6:20–26, January 2008.

[71] S.R. White and L. Comerford. Abyss: an architecture for software protection. *Software Engineering, IEEE Transactions on*, 16(6):619 –629, jun 1990.

[72] Xen Cloud Platform. `http://www.xen.org/download/xcp/index_1.1.0.html`.

[73] C. Xiaoxin, G. Tal, E. Lewis, P. Subrahmanyam, C. Waldspurger, B. Dan, J. Dwoskin, and D. Ports. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *SIGOPS Oper. Syst. Rev.*, 42(2):2–13, March 2008.

[74] Chenyu Yan, B. Rogers, D. Englender, D. Solihin, and M. Prvulovic. Improving cost, performance, and security of memory encryption and authentication. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 179 –190, 0-0 2006.

[75] Chi Zhang, A. Srivastava, and Hsiao-Chun Wu. Hot-electron-induced effects on noise and jitter in submicron cmos phase-locked loop circuits. In *Circuits and Systems, 2005. 48th Midwest Symposium on*, pages 507–510 Vol. 1, 2005.

[76] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA, 2011. ACM.

[77] Lide Zhang, L.S. Bai, R.P. Dick, Li Shang, and R. Joseph. Process variation characterization of chip-level multiprocessors. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 694 –697, july 2009.

[78] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. Hide: an infrastructure for efficiently protecting information leakage on the address bus. *SIGARCH Comput. Archit. News*, 32(5):72–84, October 2004.