ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

IMPLEMENTATION OF A LIGHTWEIGHT TRUSTED PLATFORM MODULE

M.Sc. THESIS

Mehmet Akif ÖZKAN

Department of Electronics and Communications Engineering

Electronics Engineering Programme

MAY 2014

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL OF SCIENCE ENGINEERING AND TECHNOLOGY

IMPLEMENTATION OF A LIGHTWEIGHT TRUSTED PLATFORM MODULE

M.Sc. THESIS

Mehmet Akif ÖZKAN
(50411209)

Department of Electronics and Communications Engineering

Electronics Engineering Programme

Thesis Advisor: Assoc. Prof. Dr. Sıddıka Berna Örs

MAY 2014

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ FEN BİLİMLERİ ENSTİTÜSÜ

GÖMÜLÜ SİSTEMLER İÇİN GÜVENİLİR PLATFORM MODÜLÜ TASARIMI

YÜKSEK LİSANS TEZİ

Mehmet Akif ÖZKAN
(50411209)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Doç. Dr. Sıddıka Berna Örs

MAYIS 2014

Mehmet Akif ÖZKAN, a M.Sc. student of ITU Graduate School of Science Engineering and Technology 50411209 successfully defended the thesis entitled **"IMPLEMENTATION OF A LIGHTWEIGHT TRUSTED PLATFORM MODULE"**, which he prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor :	**Assoc. Prof. Dr. Sıddıka Berna Örs** Istanbul Technical University	.....
Jury Members :	**Assoc. Prof. Dr. İlker Hamzaoğlu** Sabancı University	.....
	Assoc. Prof. Dr. Güneş Karabulut Kurt Istanbul Technical University	.....

Date of Submission : **5 May 2014**
Date of Defense : **23 May 2014**

Dedicated To the Reader,

FOREWORD

The main part of this thesis was done at the COSIC group in KU Leuven University. I would like to thank my promotor Prof. Dr. Ingrid Verbauwhede and my assistant Anthony Van Herrewege for their guidance in KU Leuven.

I would like to thank my bachelor and master supervisor Assistant Prof. Dr. Berna Örs for being a wonderful person and not keeping her guidance and support from me any time I needed.

My sincere gratitude goes to my family. I especially thank them for teaching me the value of knowledge and always being with me no matter how far they are.

I would also like to thank the jury for reading the text.

May 2014

Mehmet Akif ÖZKAN
(Engineer)

TABLE OF CONTENTS

Page

ABBREVIATIONS

AES : Advanced Encryption Standard
ASLR : Address Space Layout Randomization
ASM : Assembly Programming Language
CRP : Challange Response Pairs
DMA : Direct Memory Access
DOS : Denial of Service
FIPS : Federal Information Processing Standard
FPGA : Field Programmable Gate Array
HMAC : Keyed-Hash Message Authentication Code
I2C : Inter-Integrated Circuit
IMDs : Implantable Medical Devices
ISR : Instruction Set Randomization
LPC : Low Pin Count
LUT : Look-up Tables
MAC : Message Authentication Codes
MTM : Mobile Trusted Module
NIST : National Institute of Standards and Technology
PC : Program Counter
PCR : Platform Configuration Register
PLC : Programmable Logic Controllers
PUF : Physical Unclonable Function
RAM : Random Access Memory
RC : ROM Code of SMART Attestation
RF : Radio Frequency
ROM : Read-Only Memory
SDCC : Small Device C Compiler
SHA : Secure Hash Algorithm
SM : Small Model
SMART : Secure and Minimal Architecture for Root of Trust
SMBus : Message Authentication Codes
SMM : System Management Mode
SPM : Self-Protecting Modules
SU : Stack Using Model
TCG : Trusted Computing Group
TPM : Trusted Platform Module

LIST OF TABLES

Page

LIST OF FIGURES

Page

IMPLEMENTATION OF A LIGHTWEIGHT TRUSTED PLATFORM MODULE

SUMMARY

Today's computing platforms are becoming more and more mobile and networked, while their tasks get increasingly critical. Therefore, the need to verify and identify that a local or remote computing platform behaves as expected has become an important challenge. Software and hardware attestation protocols have been proposed to solve this problem in the past few years. While many vulnerabilities and attacks have been discovered against the proposed software based solutions, hardware based solutions are too costly for lightweight embedded devices. Recently, lightweight solutions requiring minor hardware changes have been proposed for the low-end embedded devices. One of the state of the art approaches is SMART (Secure and Minimal Architecture for Root of Trust), in which memory accesses are controlled by looking at the program counter (PC), proposed by El Defrawy et al.

In this thesis, SMART is designed for 8051 platform on a Spartan 6 CGS324 FPGA. This is the first implementation of SMART architecture that shows the cost of the modifications to hardware in terms of area and speed. In addition, a novel hardware architecture that changes memory access layout in order to provide better attestation program in terms of code size and speed is proposed. It is shown that the size of additional program code decreased by 36% and attestation speed is increased by 12% while the area and speed of hardware is optimized.

Moreover, 224 bit SHA3 (KECCAK[r=40, c=160]) software is implemented for 8051 by using assembly (ASM) and C programming languages with SDCC compiler. The trade off between code size and speed of SHA3[r=40, c=160] is presented with several implementations. No other SHA3[200] implementation for 8051 can be found in the literature at the time of this thesis is written.

GÖMÜLÜ SİSTEMLER İÇİN GÜVENİLİR PLATFORM MODÜLÜ TASARIMI

ÖZET

Gömülü sistemler, özel bir amaca hizmet etmek amacıyla gerçek zaman ihtiyaçlarına cevap verebilecek şekilde genellikle kısıtlı donanımlar üzerinde tasarlanan, güç sınırlamalarına uyan ve çevre birimleri ile uyumlu olarak çalışan bilgisayar platformlarıdır. Düşük performans gereksinimli gömülü sistemler haberleşme, tıp, savunma, otomotiv gibi hemen hemen her alanda görülmektedir.

Kullanım alanı her geçen gün daha fazla genişleyen gömülü sistemler kalp pilleri, otomobil frenleri, fabrika otomasyon sistemleri gibi kritik uygulamalarda daha fazla yer almaktadır. Artan kritik görevlerinin yanında gömülü sistemler daha çok uygulamada internet veya yerel bir ağ üzerinden başka bilgisayarlara bağlı olarak kullanılmaktadır. Genel amaçlı bilgisayar platformlarının güvenliğinin sağlanması ve sistemin güvenliğinin doğrulanması üzerine birçok çalışmanın bulunmasına rağmen yakın zamanlara kadar düşük performans gereksinimli gömülü sistemler bu konuda ihmal edildi. Ancak bu sistemleri hedef alan birçok saldırı literatürde bulunmaktadır. Vücuda yerleştirilen insulin pompası medikal cihazlarına yapılan saldırılar, arabaların kontrolör ağına (CAN) yapılan saldırılar ve Stuxnet bilgisayar kurdunun Iran nükleer tesislerine bulaşarak programlanabilir mantıksal denetleyici (PLC) sistemlerinin yazılımını değiştirmesi bunların örneklerindendir.

Günümüzde bilgisayar platformları fiziksel olarak güvenli yerlerde korunabilmesine rağmen, ağ üzerinden yapılan saldırılara ve zararlı yazılımlara karşı daha hassastır. Sistem güvenliğinin sağlanması amacıyla önerilen önemli özelliklerden biri uzaktan doğrulamadır. Yerel veya uzakta çalışan bir bilgisayarın kimliğinin doğrulanması ve çalışmasının beklenildiği gibi olmasının doğru bir şekilde kontrol edilebilmesi güvenli bir uzaktan doğrulama protokolü ile sağlanabilmektedir.

Uzaktan doğrulama yöntemi, doğrulayıcı (verifier) olarak isimlendirilen güvenilir bir bilgisayar sistemi ve sağlayıcı (provider) olarak isimlendirilen, ağ üzerinden bağlanılan, güvenilirliğinin kontrol edilmesi gereken bilgisayar sisteminden oluşur. Uzaktan doğrulama protokolü doğrulayıcının isteği üzerine başlar. Doğrulayıcı her bir hesaplamayı taze tutmak amacıyla sağlayıcıya yeterli büyüklükte rastgele bir sayı iletir. Ardından sağlayıcı; sistem durumunu, iletilen rastgele sayıyı ve sadece doğrulayıcının bildiği bir şifreyi kullanarak bu girişlere özel bir sağlama toplamı (checksum) üretir. Sağlama toplamı, her bir girişi için yeterince farklı bir sonuç üreten tek yönlü bir fonksiyon sonucudur. Sağlayıcının sistem durumu, hafızaları (memory), saklayıcıları (register) gibi yazılımın saklanmasını ve işletilmesini gösteren donanımlarının kontrol edildiği andaki içerikleridir. Sağlama toplamının şifre kullanılarak üretilmesi hesaplama sonucunu sağlayıcıya özel yapar ve hesaplamanın

simüle edilebilmesini engeller. Son adım olarak sağlama toplamı doğrulayıcıya iletilir. Doğrulayıcı, kendisine iletilen sağlama toplamının beklediği gibi olması durumunda sağlayıcının güvenli bir şekilde çalıştığı sonucuna varır.

Uzaktan doğrulama statik ve dinamik olmak üzere ikiye ayrılır. Dinamik doğrulamada sistem çalışırken dahi güvenilirlik protokolü gerçekleştirilebilirken, statik doğrulamada ise sistem sadece başlangıçta kontrol edilebilir.

Yakın zamanlara kadar düşük performanslı gömülü sistemlerde uygulanması için önerilen uzaktan doğrulama protokolleri sadece yazılım veya sadece donanım tasarımlarına yönelikti. Sadece yazılımın kullanıldığı yöntemler genellikle sağlama toplamının hesaplanması sırasında geçen zamanın da doğrulayıcı tarafından kontrol edilmesi fikrine dayanır. Bu sistemlerde, sağlama toplamının beklenen zaman aralığında hesaplanmaması durumunda fazladan işlem yaptırıldığı ve sağlayıcının zararlı bir yazılım tarafından ele geçirildiği varsayılır. Sadece donanım tasarımına yönelik yöntemler, sağlama toplamının hesaplanması için sağlayıcı platforma ek bir işlemci veya Güvenilir Platform Modülü (Trusted Platform Module, TPM) olarak isimlendirilen özelleştirilmiş bir yonga eklenmesi fikrine dayanır. Sadece yazılımın kullanıldığı sistemlere karşı başarılı birçok saldırının literatürde yer almasının yanında sadece donanımın kullanıldığı yöntemler bu çalışmada hedef olarak belirlenen işlemciler için pahalı olmaktadır.

Şimdilerde donanım ve yazılımın birlikte tasarımıyla daha düşük performans özelliklerine sahip gömülü sistemleri hedef alan yöntemler geliştirildi. Bu metotlar donanımda ufak değişiklikler önererek yazılım tarafından güvenli bir doğrulama protokolü gerçeklemesini mümkün kılmaktadır. SMART (Secure and Minimal Architecture for Root of Trust, Güvenilirlik Kökü için Güvenli ve Minimum Tasarım) bu alandaki en güncel ve önemli çözümlerden biridir.

SMART yönteminde, doğrulama programını saklamak üzere ve sağlayıcı şifresini saklamak üzere işlemciye program bellekleri eklenir. Doğrulama programının veya şifrenin değişmeyeceği durumlarda salt okunur bellek tercih edilir. Güncellenebilir bellekler kullanabilmek içinse işlemcinin yazılımı tarafından değiştirilemeyecek güvenli bir protokol eklenir. Ayrıca işlemci ve veri bellekleri arasına bellek erişimlerini kontrol etmek üzere ufak bir denetleme devresi eklenir. Eklenen denetleme devresi işlemcinin program sayıcısının değerlerini takip ederek bellek erişimlerini kontrol eder. Uzaktan doğrulamada kullanılmak üzere saklanan şifrenin, sadece doğrulama yazılımı tarafından okunmasını sağlar. Ayrıca doğrulama programının sadece ilk direktifinden uyarılabilmesini ve bölünmeden çalıştırılıp son direktifinden çıkabilmesini sağlar. Herhangi bir ihlalde sistemi sıfırlar ve üçüncü bir kişiye veri akışını engeller. Böylece doğrulama yazılımının beklenildiği gibi çalışması zorlanmış olunur. Doğrulama yazılımı doğrulayıcının isteği üzerine çalıştırılır ve uzaktan doğrulama protokolüne uyarak şifreyi, sistem durumunu ve kendisine iletilen rastgele sayıyı kullanarak bir sağlama toplamı hesaplar. Hesaplanan sağlama toplamı doğrulayıcıya iletilir.

Bu çalışmada düşük performans özelliklerine sahip gömülü sistemler için dinamik uzaktan doğrulama yöntemi olarak önerilen SMART, Intel 8051 işlemcisi için tasarlanmıştır. 8051 işlemcisi standarda uygun olarak tasarlanan Dalton çekirdeği kullanılarak Spartan 6 CSG324 Sahada Programlanabilir Kapı Dizileri (Field

Programmable Gate Array, FPGA) üzerine gömülmüştür. İşlemcinin donanımı SMART yapısına uygun olarak değiştirilmiştir.

Doğrulama yazılımı tasarımında sağlama toplamını üretmek üzere 224 bit SHA3[r=40, c=60] (Secure Hash Algorithm, Güvenli Hash Algoritması) seçilmiştir. Hash algoritması herhangi uzunluktaki bir veriden sabit uzunluktaki bir değeri tek yönlü olarak hesaplar ve farklı girişler için farklı çıkışlar üretir. Güvenli hash algoritmaları, Amerika Birleşik Devletleri Standart ve Teknolojiler Enstitüsü (NIST) tarafından belirlenmiş standartlardır. SHA3 algoritması, 2007 yılında NIST tarafından duyurulan ve 2012 ye kadar süren bir yarışmanın kazananıdır. Bir şifre kullanarak sağlama toplamı üreten hash algoritmalarına mesaj doğrulama kodları (MAC, message authentication code) denir. SHA3 algoritması, mesaj doğrulama kodu üretecek şekilde yapılandırılıp SDCC derleyicisi kullanılarak 8051 için tasarlanmıştır. Program önce C programlama dili kullanılarak tasarlanmış ve optimize edilmiştir. Daha sonra assembly (ASM) kullanılarak hedeflenen platform için hızlandırılmıştır. Sonuç olarak SHA3[200] ün kod boyutu ve hızı arasındaki oran tasarlanan kodlarla gösterilmiştir. Bu çalışma SHA3[200] algoritmasının 8051 üzerinde gerçeklemesini göstermesi açısından da bir ilk olma özelliği taşır.

Bu tez SMART sisteminin donanım ve yazılım üzerindeki alan ve hız açısından yükünü gösteren ilk çalışmadır. Sistemin herhangi bir işlemci için gerçeklenmesi sırasında karşılaşılabilecek zorluklar gösterilmiştir. Buna ek olarak SMART sisteminin bellek yapısını değiştirerek alan ve hız açısından daha hızlı bir program tasarımına olanak tanıyan ve SMART sisteminin platforma olan yükünü azaltan bir yöntem önerilmiştir. Önerilen yöntemde doğrulama yazılımına rezerv edilmiş bir veri belleği eklenmiştir. Bu sayede doğrulama yazılımı küçültülebilmiş ve daha hızlı çalıştırılabilmiştir. Sonuç olarak önerilen sistem için tasarlanan doğrulama yazılımının % 36 daha küçük ve % 12 daha hızlı olduğu gösterilmiştir.

1. INTRODUCTION

Embedded devices are being used more and more widely and they are becoming more and more networked while their tasks are getting more critical [10]. Unfortunately, any programmable device is vulnerable to malware and a significant fraction of the devices that are connected to internet is infected with malware [10].

Medical, automotive and banking applications can be given as examples where embedded devices are highly used in a networked scheme while strong security requirements are necessary. Implantable medical devices (IMDs), like insulin pumps, can be designed to be accessed via home readers through a Radio Frequency (RF) channel [7]. In-car systems are networked and can be connected to internet. Confidential data and cryptographic keys are handled in payment terminals.

E-Services, outsourcing of services, mobile devices or digital rights management can be given as additional examples where system security is vital. Integrity is essential in an election using e-voting mechanism [11]. Confidentiality and security of medical records must be provided when they are used in e-services. Sale, usage and enterprise of digital contents are other examples of security needed systems. Sale of products using e-banking needs secure money transfer over the public internet. Supporting controlled usage like limited time of access to the product or protecting private copies from fraud is only possible with a secure system running on a trusted platform.

There are a lot of work about the security of general purpose devices but embedded devices are neglected. However, some attacks have appeared targeting low cost embedded devices recently. The computer worm Stuxnet [12], targeting the Siemens Programmable Logic Controllers (PLC), spread to Iran's nuclear facilities and broke the automation system. In addition to that a lot of attacks against embedded systems, e.g. insulin pumps hack [13], automotive controllers hack [14], implantable cardioverter defibrillator hack [15], are presented in the literature. A good summary of attack methods on embedded systems can be found in [16, 17].

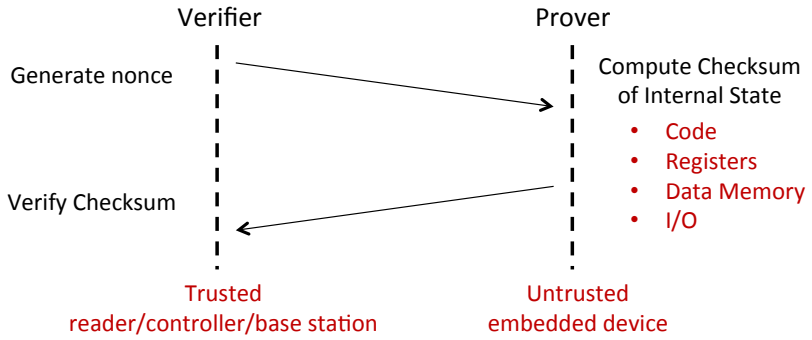

Figure 1.1: Remote Attestation Scheme.

Secure execution of the code in a platform is crucial. Likewise, the need to verify and identify that a local or remote computing platform behaves as expected has become an important challenge in order to gain trust [11]. One common solution to this challenge is remote attestation. Most of the attacks targeting embedded devices are deployed over the network instead of physical action. These concerns also motivate building a secure remote attestation mechanism.

1.1 Remote Attestation

Remote attestation is a two party protocol used to gain assurance that a local or remote computing platform behaves as expected. A basic remote attestation scheme can be seen in Figure 1.1. The scheme is based on challenge response mechanism. Prover is the device needed to be attested. A secure fresh evidence about provers internal state is produced and sent to verifier over a network when it is requested.

Basic steps of the remote attestation protocol that are given in Figure 1.1 are as follows: (1) Verifier produces a random number nonce, which is for the freshness of the attestation result, and sends it to the prover. (2) Prover calculates a checksum, which is generally calculated using cryptographic hash functions, using platform specific properties and nonce. The checksum is calculated in a way that it is specific to the platforms state and its unfeasible to be produced with a simulation of the scheme. (3) Verifier checks the checksum, then identifies and verifies the prover. As a result any corruption of the platform is detected. In a real life scenario verifier can be a trusted server and the prover can be an embedded device that is deployed to the field.

Remote attestation protocols can be classified as static and dynamic attestation. The state of a device can be verified at the run-time in the dynamic attestation. Different remote attestation protocols are proposed from purely hardware based to software based approaches for high end devices to low cost embedded devices.

1.2 Related Work

Different techniques are proposed for higher end devices in order to strength security. Using hardware security features to provide strong software isolation and securely support remote attestation are some significant research lines. Deploying hardware features to provide privileged levels and virtual memory support for operating system is one solution. Operating system can support isolated processes and guard their interactions in this way. Creating memory safe virtual machines or deploying hypervisors, which monitors the execution, with the support of hardware are other research lines aiming to provide isolation of programs sharing same physical device.

Existing hardware based approaches are based on adding a chip that cannot even be compromised by the owner. Trusted Computing Group (TCG) is a recent industrial initiative towards the standardization of attestation specified Trusted Platform Module (TPM) [18] as the standard for reporting of a platform's software state. One can make provable statements about the platform configuration and protect keys from multiple users by using TPMs and secure co-processors. However, these approaches are focused on general-purpose hardware and they are still too costly for low-cost embedded devices like wireless sensor networks. Dynamic root of trust mechanism is provided by the TPM specification so attestation can be done not only before but also after the boot process. Intel TXT [19] and AMD SVM are some commodity product examples.

Software based solutions are proposed in order to be used in resource constrained platforms [20–24]. These methods provides attestation capability without relying on any hardware. A checksum is computed using a software function that uses side effects of computation like the execution time and any unexpected overhead is detected. However, current software based remote attestation methods rely on too restrictive assumptions for many realistic applications and expect no collusion which means that

the verifier should be linked directly to the prover without using a network [25]. In addition, many vulnerabilities and attacks have been discovered against them [25, 26].

Recently, lightweight solutions requiring minor hardware changes have been proposed for the low-end embedded devices [2, 5–7]. Schultz [5] et. al. proposes a physical unclonable function (PUF) [27], physical one way function, hardware based mechanism with combination of software based attestation. Strackx et al. [2] proposes self-protecting module (SPM) structure and gives the idea of program counter (PC) based memory controller to create isolated processes. El Defrawy et. al. [7] developed this idea and designed SMART (Secure and Minimal Architecture for Root of Trust), which provides dynamic root of trust requiring only a small additional hardware that checks the PC and controls the memory accesses. Sancus [6] is another method that offers extensibility using the same PC based idea to create isolated processes. All of these methods are very promising and very new.

1.3 Conclusion

The background information is given in Section 2, and the term software vulnerability is explained according to cryptography in Section 3. Then, defense and attestation mechanisms are summarized in Section 4. The security analysis and abstract design of the SMART architecture is discussed in Section 5. Finally, software and hardware implementations of SMART scheme are shown in Section 6 and Section 7, respectively.

2. BACKGROUND

2.1 Introduction

Necessary background information will be given in this section. Cryptographic hash functions, secure hash algorithms, message authentication code and hash-based message authentication code will be explained briefly.

2.2 Cryptographic Hash Functions

A cryptographic hash function is a one-way procedure that takes an arbitrary length of data as input and produces a fixed-length bit string as output in a way that a small change of the data causes change of the output [28]. Generally, the input of the function is called the message and the output is called the hash value or the message digest. An ideal hash function computes the hash value for any given message easily. A cryptographic hash function must withstand all known types of cryptanalytic attacks and minimally it must have the following properties [28]:

A hash function

$$h : X \to Y \tag{2.1}$$

- **Preimage resistance:** Difficulty of finding any message

  $$x \in X \tag{2.2}$$

  from the given hash value

  $$h(x) = y \tag{2.3}$$

  (impossible in ideal one-way function).

- **Second preimage resistance:** Difficulty of finding another message

  $$x' \in X \tag{2.4}$$

that has the same hash value

$$h(x') = h(x) \in Y \tag{2.5}$$

for the given message

$$x \in X \tag{2.6}$$

.

- **Collision resistance:** Difficulty of finding two different messages

$$x \neq x' \in X \tag{2.7}$$

with the same hash value

$$h(x) = h(x') \in Y \tag{2.8}$$

.

For an ideal cryptographic hash function, it is infeasible to break these resistance properties and easy to compute the hash value of any given message. The integrity can be assured by checking if the message is changed. This can be done using a trusted verification system for the hash value of a message that works even the message is in an insecure place [28].

2.3 Secure Hash Algorithms

The National Institute of Standards and Technology (NIST) published a family of cryptographic hash function standards, called as Secure Hash Algorithms (SHA), as a U.S. Federal Information Processing Standard (FIPS) [29]. First, SHA was designed and specified in FIPS-180 (1993) [29]. Then, it was revised and specified as SHA-1, which produces 160-bit hash values, in FIPS-180-1 (1995). Next, SHA-256, SHA-384, SHA-512 algorithms are issued in FIPS 180-2 (2001) [30]. With these algorithms the security strength is increased by using Advanced Encryption Standard (AES) [31] and more compatibility is provided with different output bit lengths. After some weaknesses of SHA1 [32] and possible threats to SHA2 had been discovered, NIST announced a public competition to develop a new standard in 2007. 64 candidates were

submitted in October 2008; 51 of them are selected for the first round in December 2008. They were downsized to 14 by the second round in July 2009 and 5 finalist, BLAKE [33], Groestl [34], JH [35], Keccak [36], and Skein [37], were chosen in December 2010. Finally, the Keccak algorithm, designed by Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche, was selected as the SHA3 standard in October 2012.

2.4 Message Authentication Code

Providing a way to check the integrity of information that is stored or transmitted between unreliable parties is a very important need especially in communication. This mechanism is provided by message authentication codes (MAC). A message authentication code (MAC) is a hash function that produces a MAC value using a key [28]. It is also called keyed hash function. The MAC algorithm can both provide the authenticity and integrity assurance. Authenticity assurance can be gained by sharing a secret key between parties and checking the MAC of the message for that key. The integrity assurance comes from the hash function.

One common way to construct a MAC is to concatenate the key with the message in an unkeyed hash function. However, there are additional security requirements for a MAC compared to an unkeyed hash function. As an example, an attacker must not have the ability of guessing the MAC of any other messages without performing infeasible amounts of computation, even he has ability to generate MACs for the chosen messages [28]. Therefore constructions and padding schemes, e.g. NMAC and HMAC, have been designed to avoid such weaknesses [38]. Otherwise, length extension attack can be applied to SHA1 and SHA2 and some information can be learned when the message and the length of the key is known. Therefore, they have to be used in a padding scheme like HMAC to build a keyed hash function. However, Keccak as SHA3 standard does not have any length extension weakness, so a MAC algorithm can be constructed by just prepending the key with the message [36].

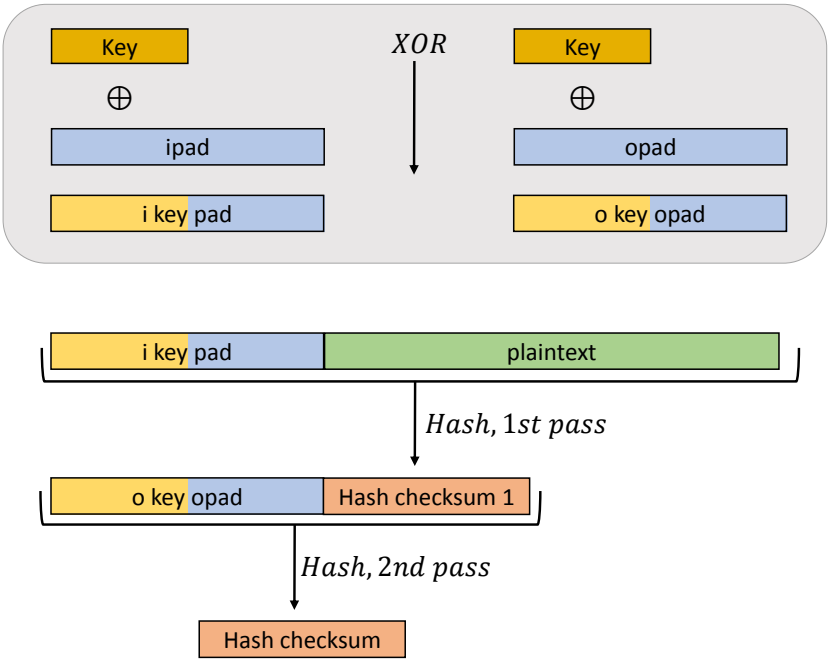

Figure 2.1: HMAC-SHA1 Construction .

2.5 Hash-Based Message Authentication Code

Keyed-hash message authentication code (HMAC) is a specific construction that provides a secure mechanism to produce a MAC, in which a hash function is used with a combination of key. HMAC construction was first published in 1996 by Mihir Bellare, Ran Canetti, and Hugo Krawczyk and standardizied as RFC 2104 [39].

The goals of the HMAC padding scheme is explained by the writes in five substances:
(1) Provides a secure use for all working hash functions in a MAC constructed way.
(2) Preserving the performance of hash function when it is MAC constructed.
(3) Providing an easy way to handle keys.
(4) Providing a well understood proof for the strength and security of a MAC construction for a secure hash function.
(5) Providing replaceability of the hash function of a MAC construction.

Any cryptographic hash function can be used with HMAC function like MD5 or SHA-1 and the construction will be named as HMAC-MD5 or HMAC-SHA1 respectively. The cryptographic strength of the HMAC depends upon the cryptographic

strength of the underlying hash function, the size of its hash output, and on the size and quality of the key. Any size of key can be used but the values smaller than the hash output size decreases the strength of the function while the longer sized keys does not increase [39].

HMAC-SHA1 is given in Figure 2.1 as an example of the construction. The mathematical operations of the algorithm are shown in the Equation (**2.9**) using the language of standard [39]:

ipad = the byte 0x36 repeated until the one-block length is reached (0x36..36),
opad = the byte 0x5C repeated until the one-block length is reached (0x5C..5C),
"||" denotes the concatenation operation,

$$HMAC(text) = H(K\ XOR\ opad\ ||\ H(K\ XOR\ ipad\ ||\ text)) \qquad (\mathbf{2.9})$$

3. OVERVIEW OF SOFTWARE VULNERABILITIES

Software vulnerability in terms of security, types of attacks using them and countermeasures are explained in this chapter. The purpose of this chapter is to give the idea to the reader about the security of software and why they are vulnerable. Simple examples are given to present how easily a software can be vulnerable, what kind of attacks can be exploited to them. The main countermeasures to attacks are mentioned. The main research tracks are summarized.

This chapter is written in an introduction manner and details are not given. The methodologies in this chapter are mainly given on high end computing devices since the research has made mostly considering them and low end devices also have similar properties in some subjects. The active research areas and the needs for the security of computing devices against software vulnerabilities are presented.

3.1 Attacks Using Software Vulnerabilities

Frequently, computing platforms are threatened by external attacks, which can aim to gather sensitive information, disrupt computer operation, gain access to private systems. In computer security these attacks have become one of the most compelling challenges because of their combined effects [40]. Generally, they come from the communication channel and locate some code to the program memory then occupy the control of the system by using the low-level software vulnerabilities [40].

A software vulnerability is a bug that can be triggered by an attacker with possibly disastrous consequences [1]. An introductory background information about such vulnerabilities is given in this topic. Their initiation to a software system is shown and their difficulty of elimination is discussed.

Baltopoulos and Gordon have proposed a principle called *"source-based reasoning"* to provide a software low-level security [41]. The principle indicates that security

properties of a software system should follow the review of the source code and its source level semantics, and should not depend on details of the compiler or execution platform [41]. This means that a programmer should only care at the source code level and let the compiler and run-time system worry about the low-level execution platform.

Many mechanisms have been proposed for defending against these attacks [40]. However, secure high level abstraction of the platform have not been achieved in high level programming languages [40]. Still, consequences of a bug in the software can not be understand without considering many details of the execution platform.

Both attacking such vulnerabilities and defending against such attacks depend on low level details of the software and the attacked computing platform. Although source based reasoning is provided in safe languages like Java, more powerful attacks can be made in the case that attacker can interact with the code at the machine code level. As an example, a browser extension can attack any web page visited or a malicious kernel module can install a root kit [40].

Attacker models are distinguished into two models, which are ***interactive attacker model*** and ***in-process attacker model***, in the paper [40].

The interactive attacker can interact with the program by giving input and reading output. He uses the failures of source-based reasoning principle of the unsafe languages such as C and C++. For examples, he can use a ***buffer overflow*** vulnerability of a software that is designed in unsafe language such as C or C++ or exploit a logic flaw against a program written in a safe language. This is the case of an attacker trying to undermine a network service that is protected from a server [40].

The in-process attacker can load some machine code while the execution of the program. For example; he can scan or change the memory for secrets, change the control flow although the program is designed in a safe language. Applications built from components coming from different stakeholders or the binary run time plugins of the applications is an in-process attacker model [40].

An overview of the interactive attack model are summarized in the paper [1] and given four selected methods of applying these attacks: 1. Corrupting the return address of a function on the stack. 2. Corrupting function pointers stored in the heap. 3. An

Table 3.1: Example stack area for the Algorithm 1 [1]. The attacker changes the function return address to 0x0012ff48, which he placed an infinite loop as seen in the Table 3.2.

address	content		
0x0012ff5c	0x00353037	argument second pointer	
0x0012ff58	0x0035302f	argument first pointer	
0x0012ff54	0x00401263	return address	0x48 0xFF 0x12 '\0'
0x0012ff50	0x61666473	saved base pointer	's' 'd' 'f' 'a'
0x0012ff4c	0x61666473	tmp continues	's' 'd' 'f' 'a'
0x0012ff48	0x61666473	tmp continues	0xCD 0x2E 0xEB 0xFE
0x0012ff44	0x612f2f3a	tmp continues	':' '\' '\' 'a'
0x0012ff40	0x656c6966	tmp array:	'f' 'i' 'l' 'e'

existing code can be executed after pointers are corrupted. 4. Control flow can be changed via corruption of data values that determine behavior.

Algorithm 1 Example of stack-based buffer overflow vulnerability: The program flow can be conducted by the attacker when the function returns if the strings are chosen by the attacker [1].

```
int unsafe( char* first, char* second ){
// Must have strlen(first) + strlen(second) < MAX_LEN
    char tmp[MAX_LEN];

// Style-1
    char* b = tmp;
    for( ; *first != '\0'; ++first, ++b ) *b = *first;
    for( ; *second != '\0'; ++second, ++b ) *b = *second;
    *b = '\0';

// Style-2
    // strcpy( tmp, first );
    // strcat( tmp, second );

return strcmp( tmp, "file://foobar" );
}
```

The function in the Algorithm 1 takes 2 string pointers, concatenate them and returns the result of the comparison with "file://foobar". The same code can be written as the style-2, which is commented in the Algorithm 1. Both of the codes have stack-based *buffer overflow* vulnerability. If the input strings are chosen from the attacker return address of the function can be changed and the attacker can gain the full control of the program.

Table 3.2: The ASM program of the attacker in the Table 3.1 [1].

Machine Code Opcode	ASM instructions
0xcd 0x2e	int 0x2e ; system call to the operating system
0xeb 0xfe	L: jmp L ; a very short, direct infinite loop

An example interactive attack to the Algorithm 1 can be made as shown in the Table 3.1. In the case of a regular operation of the function in the Algorithm 1, the concatenation of two input strings would be stored to the addresses between 0x0012ff40 and 0x0012ff4c. However an attacker can overflow the reserved area and can change the return address as he wants. In the case of Table 3.1, he changed return address to 0x0012ff48 where he placed an infinite loop as can be understand from the Table 3.2.

This attack is called ***return-address clobbering*** and it was widely effective a decade ago on the codes compiled with C , C++. Blaster worm, which affected a majority of Internet users in 2003 [42], was built using this vulnerability. Since the worm was using loops it couldn't be detected by the analyses tools. Although this attack is not prominent still, this example gives the idea of software vulnerabilities and shows how easy to have one.

A safer version of the program in the Algorithm 1 is given in the Algorithm 2.

Algorithm 2 More Secure design of the Algorithm 1 [1].

```
int safer( char* first, char* second ){
    char tmp[MAX_LEN] = { '\0' };
    strcpy_s( t, _countof(t), a );
    strcat_s( t, _countof(t), b );
    return strcmp( t, "abc" );
}
```

Two more examples given in the paper [40] for the interactive attack, are shown in here to present other basic software vulnerabilities for the interactive attacker model.

The program in the Algorithm 3 takes the username and password from the user and makes privileged operations if they have a correct match. Although the memory reservation for the username variable is 24 size of char, the program allows to take 28 size of char. This error causes a *buffer overflow* vulnerability. Typically the compiler will put other variables next to username. This can cause to change the result of valid

credentials by making authenticated variable different than zero. Therefore, one can make privileged operations by using the compiler and execution platform details.

Algorithm 3 A buffer overflow vulnerability for the interactive attacker coming from the error of the memory reservation for the username variable.

```
int do_maintenance() {
    int authenticated = 0;
    char username [24];
    char password [28];

    fgets(username , sizeof(password), stdin);
    fgets(password , sizeof(password), stdin);

    if (valid_credentials (username, password) == 1)
        authenticated = 1;

    if ( authenticated )
        //Do privileged operations
}
```

The program in the Algorithm 4 takes a string from the command line and initializes the src buffer then transforms it with an assigned operation and writes the result to dst buffer. Since there is no check for the src buffer a ***code injection attack*** can be realized on this code. One can input more than 256 bytes and overflow the src and dst buffers and change the function pointer that is assigned to capitalize function. As an example he can direct the program to address of src after he injected his own code to it. As seen in this example, an interactive attacker can become an in-processor attacker by inserting an arbitrary code.

The class of bugs, which is produced because of unreferenced pointers, can be used to put the program into an unintended execution flow. This causes a vulnerability referred as ***Dangling Pointer***, which can be exploited by an attacker to reallocate the memory and launch a buffer overflow attack [43].

For these examples given in the Algorithm 1, 3, 4 although code semantics shows the state of the program as undefined, an attacker can use vulnerabilities by relying on the compiler and execution platform details such as layout of the variables on the memory and Von Neumann architecture, which executes the code in the data memory.

Algorithm 4 Attack2

```
struct data_node {
    char src [128];
    char dst [128];
    int (* transform_func )( char *, char *);
};

int main (int argc, char *argv[]) {
    struct data_node *n;
    int i;
    n = malloc(sizeof(struct data_node ));
    n->transform_func = capitalize;

for(i=0; argv[1][i] != '\0'; i++)
        n->src[i] = argv[1][i];

(*n->transform_func )(n->src, n->dst);
};
```

More examples can be given for the in-processor attacker. A malicious plugin that comes from an extension for the web browser. He can access to all of the stored secret keys or passwords. Although source code semantics of these variables are private, the code can access these by scanning the memory.

Although the problems of constructing source based reasoning is well understood for the interactive attacker model, it is difficult and still an open problem for the in-process attacker model [40].

3.2 An Overview of Countermeasures to Attacks

The main countermeasures to the attacks using the software vulnerabilities will be discussed in this topic in order to show the evolution of defenses.

3.2.1 Defenses to interactive attacks

Providing source based reasoning for the program is highly studied in the literature and there are two main countermeasures against interactive attacks [40]. One approach is to use ***safe languages*** to design and another is to ***execute unsafe languages more defensively***.

3.2.1.1 Safe languages

High level languages, e.g. Java, C#, Scala, are accepted as safe languages if they provide very restricted access to unsafe features. This is achieved by ruling out dangerous language features, compile-time and run-time check. However unsafe languages like C and C++ are used widely and they are not expected disappear in a near future, safe languages are only a partial solution to interactive attacks [44].

For instance, accessing to the out of bounds of an array is not possible in safe languages. The code either can not be compiled for the bug in the code or a run time error occurs for an exception causing a violation. Moreover, a dangling pointer can not be created because of language constructions like garbage collectors.

3.2.1.2 Safer execution of unsafe languages

Many techniques have been developed to gain low-level security for the unsafe languages. Most of them are explained in the Chapter 1. One method is to design the program as tamper resistant with ***Additional Run Time Checks***. In these methods, behaviors of the execution is controlled and the program is terminated in case of any violation. Placing some random variables called ***stack canaries*** to the stack boundaries and checking whether they are changed as in ***Stack Guard [45]*** is one of the early important example. Making ***data memory non executable*** to prevent code injection attack is another approach in this context.

Another way is to use ***Randomization*** in order to make the executable code meaningless to attacker or make the malicious code of attackers nonsense by changing. ***Address Space Layout Randomization (ASLR) [46]*** and ***Instruction Set Randomization (ISR) [47]*** are main methods.

Although a lot of developed cautions to interactive attacks are widely used in operating systems they are still not cure for the vulnerabilities. A lot of attacks are realized against the ASLR [48–50]. It is proved that the canary based systems like Stack Guard can be bypassed [51] and non-executable stack or heap can be tricked with return to libc attack [52] or return oriented programming [53]. As a result, there are

still evolved interactive attacks that can defeat known countermeasures to corrupt a vulnerable system.

Return to libc attack is a method to exploit buffer overflow vulnerability of a system that has a non-executable stack. This attack starts with a buffer overflow and continues by changing address with an address of a function that is in the binary or shared library. The shared library called "libc" provides the C runtime on UNIX style systems.

As a result, the return address is changed with one of a function in libc then correct arguments are passed and have it executed for the attacker in return to libc attack. Therefore stack protection is bypassed.

Return oriented programming is a method that allows an attacker to execute code although the system security is strength with non executable memory segments and code signing. *Code signing* is a security method that digitally signs executables and scripts by calculating hash so that any corruption can be detected by the author.

In the return oriented attack, a malicious code execution can be realized by using gadgets instead of programs that are in the stack or libc after the program control flow is corrupted. The term gadget is used for a carefully chosen machine instruction sequence. A gadget can start from the middle of existing instructions and different instructions can be simulated. All the job an attacker should make is to find usable byte sequences in the executable memory and realize return chaining to follow each other in order to perform arbitrary operations.

To summarize, a fully functional "language" is provide in return-oriented programming so that an attacker can use it to make a compromised machine perform any operation desired.

3.2.2 Defenses to in-process attacks

As discussed before, the in-process attacks can load and execute machine code. The attacks of this class are more powerful and realistic.

A countermeasure approach to this type of attacks is to monitor an untrusted code by a trusted software, which uses some methods like ***software fault isolation*** [54] to detect and change a malicious software, before it enters to the process. The weakness

of these methods is that although the main program is protected from its untrusted modules, modules are not protected from host. For instance, a plugin still can access to secret keys protected in web browser [40].

In the past few years, several methods have been proposed based on isolated execution of modules in order to protect modules and host from each other. The aim is to provide a secure execution for the modules even the host is compromised. These methods ranging from hardware-only to software-only.

As discussed in the chapter 1, these methods can be summarized as ***techniques based on randomization***, ***protected software modules*** and ***Trusted Computing Base (TCB)***. In addition it is mentioned that more lightweight solutions, especially targeting low cost embedded systems, for isolated execution are became popular in these few years. ***Tiny operating systems*** [55–60] and small hardware additions for ***program-counter based memory access control*** [2, 5, 7] are the most promising research lines.

3.3 Conclusion

The concept of software vulnerability and attacks to them are mentioned in this chapter. It is told that source based reasoning is a necessary property of a software although it doesn't guarantee the low level security of a software. The reason is that a mallware can change the program or inject an arbitrary code at the machine code level although high level security primitives like being private is preserved after the compilation of software.

Common known attack methods as corrupting the return address of a function on the stack, corrupting function pointers stored in the heap, executing an existing code by corrupting pointers, changing program flow by corrupting data values that determine behavior are told. In addition, dangling pointer vulnerability, buffer overflow attack, code injection attack, return to libc attack and return oriented programming attack are explained.

The attacks are investigated using the methodology of two types as interactive attacks or in-process attacks. In-process attacks can inject some arbitrary code while interactive attacks can only give input to the system and read the output. It is told that,

still evolved attacks that defeat the modern defending systems appear although they are highly studied in literature. Moreover, they are still important since interactive attacks can turn themselves to in-process attacks. The in-process attacks are more powerful ones and countermeasures to them are highly studied in this decade around the idea of providing strong isolation of code and data. Defending systems in the range of software-only to hardware-only have been proposed. Systems using software modules depending on the Trusted Computing Base (TCB) are popular recently. In addition to their implementation on high end devices, another approaches for embedded devices are appeared recently including tiny secure operating systems and adding small memory access control circuits for the low level.

Since any computing device can be corrupted by an uncalculated attack, verifying the state of a computing device is very important. Therefore realizing attestation schemes in a secure way in order to acquire trustworthiness of a computing device is vital.

4. REMOTE ATTESTATION AND DEFENSE SYSTEMS

Providing a secure attestation and execution for the modules even when the host is compromised has became an important issue for the security of programs. A lot of research have been made in this decade and some methods are proposed. The proposed methods are mainly using *tamper resistant software* and *isolated execution of modules of the program* ideas [40]. The computing platform and its network are considered as the combination of the modules then secure execution environments are tried to be produced for the protected modules.

Different methods are proposed from purely hardware based to software based approaches mostly targeting high end devices to embedded devices [11]. Recently some promising lightweight approaches are proposed. Although the scope of this thesis is lightweight solutions, all methods are summarized then promising lightweight solutions other that SMART [7] is explained in this chapter.

4.1 Software Based Systems

Tamper resistant software [61] method is proposed to detect any tampering in the code by adding built-in integrity checks to the code. Tamper resistant software typically calculates a checksum on its code and checks whether the checksum corresponds with an expected value. *Additional Run Time Checks* and *Randomization* is the main methods for tampering [40].

- ***Run Time Check*** is generally based on checking source level behaviors at the run time and terminate the program in case of violation.

  Stack Guard [45]: A randomly chosen variable, called as canary, is added on the stack boundaries, before the return address, of the function and checked after the execution of the body of the program. Catching a modified canary means a buffer

overflow therefore the program is terminated. This work is important because of the introduction of *stack canary*.

There are a lot of proposals in literature that extends the stack canary concept by adding new features to gain increased security for the stack, local variables and heap. Different versions are still in use for operating systems [40, 62].

Another approach is to make the data memory non executable with additional run time checks. This is generally provided by today's operating systems.

- ***Obfuscation (techniques based on randomization)*** [63–65] is the process of randomizing the binary executable that makes difficult to understand and change the code even at the machine code level. Full abstractions can be provided at the low level machine code. The data layout or the instruction set can be randomized in order to make the attacker's code meaningless.

  Address Space Layout Randomization (ASLR) [46]: The stack, heap and libraries of a process are loaded to different memory spaces. Therefore even the program is corrupted from an attacker, he is prevented to use the variables of the program and jump to malicious code since he doesn't know the application addresses. This is implemented in almost all operating systems.

  Abadi and Plotkin [66] proposed an approach to keep abstraction for the low level by storing a random number to each location. While the confidentiality is provided in high level code by simply protecting the location of variables, it is preserved at the low level by mapping private locations to random level addresses.

  Instruction Set Randomization (ISR) [47]: Each process is attached with different instruction set in order to make injected code meaningless. Point Guard uses encryption for the random function. Therefore even an attacker changes a pointer, the decryption of the changed data would be something absurd for his aim [67].

  There are applications using the same principle by randomizing the address space of memory [68] or the interface of the operating system [69]. However, these methods seems to be rough models for the real execution platforms [40]. Moreover, obfuscation makes the reverse engineering process more time consuming but not impossible [70, 71, 71, 72].

- **White-box cryptography**, in which the keys are hide into applications, is another method for tamper resistant software. Chow et al. [73] proposed to hide the key in executable code while Michiels and Gorissen hide in a large collection of lookup tables [74]. The key will be changed if the code is altered.

  However, when these software techniques are used to protect standalone, non-networked applications, their security is limited. Most proposals for a white-box block cipher have been broken [70, 71, 71, 72].

  Networked schemes for the design of tamper resistant software are more powerful since the comparison of integrity checksum doesn't have to be inside the software and it can be performed remotely. Moreover, runtime can be controlled to check if additional processes are executed.

- **Code Replacement** [75] is proposed by Ceccato et al as a modified approach suggesting to change the client application periodically with a new version using a new key or obfuscated.

  Although the comparison of the checksum in the remote attestation platform can be performed in a trusted manner by a remote service provider, the integrity of the platform becomes the main problem. For example, an attacker can send the result of the uncorrupted platform instead of the targeted one in the scenario that the adversary has complete control. Moreover, the platform can be corrupted just after it is attested.

  Kennell and Jamieson [76] proposed **genuinity tests** for the verifier to check whether the program is running on the targeted system e.g. specific hardware, operating system or virtual machine. In addition to being slow, they are also cracked by Shankar et al. [77].

- First **SWATT** [20] then **Pioneer** [21, 78] are proposed by Seshadri et al. for the software based remote attestation for embedded devices and high-end devices relatively. In these schemes, run time of the attestation is checked by the verifier. The idea is that modifying, executing an arbitrary code or redirection of the network flow gets extra overhead to the system and this can be detected by checking the run time of the attestation code when the checksum is requested.

- **Software Splitting** is an alternative solution proposed by Zhang and Gupta [79] in order to protect software security. Small and essential parts of an application are removed from untrusted platform and they are stored to a secure server or a coprocessor.

In the past few years, several methods have been proposed based on ***isolated execution of modules*** in order to protect software partitions and host from each other. The aim is to provide a secure execution for the modules even the host is compromised. These methods ranging from hardware-only to software-only.

- The ***Multics operating System*** [80] was a long term system modern operating system project introducing the security concepts. Protection rings idea is introduced for hierarchical layering in order to isolate less trusted programs. However, it is still being attack successfully despite many improvements and research.
- ***On-board Credentials Project*** [81] is another example of isolated execution that doesn't need additional hardware. A specific key is produced from a master key for each module to provide secure communication between different modules. Moreover, only one program is loaded effectively for any single moment.

4.2 Hardware Based Systems

An early example of a hardware-based mechanism is Secure Boot [2], which verifies system integrity at boot time. The root of trust is an immutable bootloader stored in ROM along with a public key. It verifies code signature and executes the code (i.e., boot) only if the signature is valid, thus authenticating code origin and integrity.

4.2.1 Secure co-processor

A Secure coprocessor is an hardware module, which includes CPU, bootstrap ROM, and secure non-volatile memory, that can be trusted to execute their software correctly despite attacks [82]. Sensitive information, e.g. cryptographic keys, can be stored in a secure way and the critical memory is erased in case of penetration, even for hardware attacks . Similarly, their encryption systems are assumed to be resistant

to crptoanalysis. Providing guarantees about the privacy and integrity of the secure non-volatile memory of secure coprocessors support basis for distributed security systems [83].

A secure communication channel to untrusted platform can be established over a network using secure coprocessors with proper cryptographic techniques [11]. Moreover this channel can be used for a remote attestation scheme where integrity can be verified or a software update mechanism.

The FIPS 140-2 standard [84] for cryptographic modules includes specifications for secure coprocessors and four security levels are defined according their strengths. There are highest graded coprocessors in the market, e.g. IBM 4764.

Secure coprocessors mostly used in applications that processes sensitive information like electronic commerce applications and banking applications. Electronic contracts, electronic currency copy protection and secure postage meters can be given as examples [82]. Latest smart card designs can also be shown another example. They includes coprocessors that are constrained with power and storage capacity.

4.2.2 Trusted computing and trusted platform module (TPM)

Trusted Platform Module (TPM) is a dedicated hardware [85], like coprocessor, that is optimized to provide security and safeguard private and secret data. It can securely store cryptographic primitives, including passwords, certificates and encryption keys, that is used to authenticate the platform. In addition platform measurements can be stored for the measurement of assurance of the computing platform.

The first TPM were developed by the Trusted Computing Platform Alliance (TCPA), including HP, IBM, Intel, Microsoft etc. in 1999. Then TCPA was suspended and a new group called the Trusted Computing Group (TCG) is announced with membership of 14 companies including AMD, Hewlett-Packard, IBM, Intel Corporation, Microsoft, Sony Corporation and Sun Microsystems in 2003.

They define themselves as a non-for-profit organization formed to develop, define and promote open, vendor-neutral, global industry standards, supportive of a hardware-based root of trust, for interoperable trusted computing platforms [85]. TCG

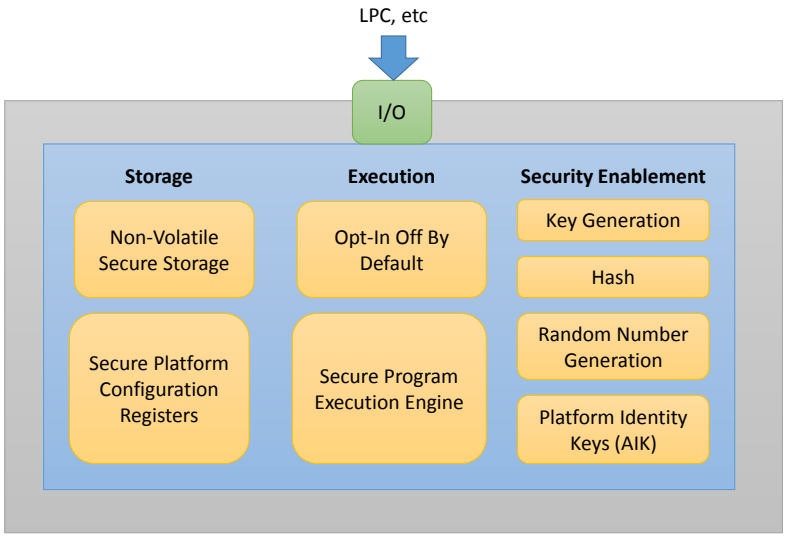

Figure 4.1: Components of Trusted Platform Module.

adopted the TCPA specifications and shifted their focus to expansion of the use of the specification and maintaining its further development. Open industry specifications are published for the Trusted Platform Module (TPM) security chip by TCG.

Authentication and attestation are necessary for safer computing in all environments and TPM chips helps to provide these properties [86]. It is guaranteed that the platform can prove that it works as it is claimed via authentication while the trustworthiness of the platform is checked using attestation. The components of the TPM specification is shown in the Figure 4.1. The components can be classified as three parts that are for secure storage, execution and security artifacts. TPM is communicates with the central microprocessor using the Low Pin Count (LPC) bus, which is standardized interface. Also there are some manufacturers communicating with embedded systems using Inter-Integrated Circuit (I2C) or System Management Bus (SMBus) interface.

One of the main structure of the TPMs are configuration registers (PCRs). PCRs keep the state of the system. They are only accessible only via a fixed API and they are only reset on boot. PCRs are updated after a module is called with the concatenation of the old values and the cryptographic hash value of the software. A comparison is made between current measurement and corresponding expected value. System is halted in case of any violation. As a result integrity of the system can be controlled using PCRs and a root of trust for measurement (RTM) can be established. The integrity of the system, including TPM itself, BIOS, OS etc., is checked in each boot.

$$PCRnew = H(PCRold||M) \quad (\mathbf{4.1})$$

Although TPM firstly thought a solution for all computing platforms it is seen that different platforms can have different security requirements especially embedded devices and mobile phones. As a result Mobile Trusted Module (MTM) is published by trusted computing mobile phone group (MPWG). Defining a subset of TPM commands are defined as mobile specific and distinguishing local and remote owner of the platform are main differences.

4.3 Software Systems Using Hardware

Hardware based cryptography is considered stronger than software based schemes against external software attacks. They have a lot of applications such as digital signing, mission critical applications, secure email or secure document management where greater level of security is needed. It is hard to access information in a hardware protected computing platform without proper authorization. The sensitive information and functionality can be sealed in case of violation.

Trusted Computing Base (TCB) is developed to provide isolation of software modules with the help of additional hardware. However, It is important to remember that TPM can not control the software in the device. Pre-run time configuration parameters can be stored in TPM but beyond that it is the other applications duty to determine and implement policy of using TPM. Therefore systems are developed that uses the TPM that provides isolated secure execution for modules in order to protect them from attacks. Flicker [87], TrustVisor [88] and SICE [89] are important designs in this content.

Flicker [87] uses a chip called *Trusted Platform Module (TPM)* for high end devices. Operating system and Basic Input/Output System (BIOS) are excluded and The central processing unit (CPU) is set into a known safe state before a module is executed. This is called late lunch. The allocated memory for the module is cleared after it is terminated and the execution of the application is resumed. PCRs are updated after a module is called. Moreover, sensitive information is stored or shared between modules by specifying PCRs for them.

TrustVisor changed late lunch sequence with virtual machine extension to check the unviolated execution of the module. Although Flicker only uses 250 lines of code, it has significant performance overhead to the system. The performance is improved in TrustVisor and SICE.

SICE [89] is a novel framework to provide hardware-level isolation and protection for sensitive workloads running on x86 platforms in compute clouds. The security of the isolated environments is guaranteed by a trusted computing base that only includes the hardware, the BIOS, and the System Management Mode (SMM). System management module (SMM) ensures that the software enters a known state instead of late lunch of Flicker.A module can be started after a management interrupt is requested (SMI) and an isolated execution environment is prepared before the module is executed in this scheme.

4.4 Lightweight Solutions

Recently, more lightweight solutions for isolated execution are proposed with ***tiny operating systems*** [55–60] or small hardware additions [2, 5, 7] to ***control memory access***.

Software based attestation algorithms uses a checksum function that is placed in the prover. The prover calculates the state of the software in the platform and sends it to the verifier. The software state is generally calculated by using a cryptographic hash function on the execution elements like the program code. Then verifier checks the checksum. In addition, checking the *execution time* is proposed by the SWATT [20] team. Validating the checksum means that the challenger has the expected software. Checking that the execution is finished in a specific time frame proves that no extra overhead has been created, which means that the checksum is calculated honestly by the platform. Note that the term prover, which can also be used as challanger, is used for the platform that need to be attested while the verifier represents the platform that verifies the prover.

However, strong assumptions must be made in order to make the software based attestations secure. Firstly, they are vulnerable to network attacks such as

impersonation or collusion with other devices [5]. This means that the prover should be linked directly to the verifier without using a network. In addition, they are vulnerable to hardware attacks like overclocking or increasing the memory of the processor [5].

4.4.1 PUF based system of Schulz et al.

Schulz et al. [5] constructed a lightweight solution that combines software-based attestation with a PUF. Schulz et al. [5] discusses that a software based system can be secure if the checksum calculation is linked to challenger's platform. One approach is to include hardware specific side effects, e.g. CPU states, caching effects to the checksum function [76] in order to prevent an adversary to simulate the attestation process. Unfortunately, it is shown that these properties are not strong enough to define the individual hardware platform instead only groups of devices can be distinct [77,90].

Schulz et al. [5] proposes using Physically Unclonable Functions (PUFs) instead of hardware specific side effects. Software based attestation scheme is modified by integrating device specific hardware properties in this way. The scheme is claimed to be secure against collusion of challengers meaning that the attestation can't be simulated by a fake challenger. Therefore authentication and attestation of remote challengers can be securely supported. In addition, the hardware attacks can be detected by the challenger.

The scheme needs PUFs that are designed to behave deterministic and have the following properties [5], more detail can be found in [91]:

- *Robustness:* PUF always returns the same output y for the same input x.
- *Independence:* There is no collusion between the outputs for the same input of any two different PUFs. The output y of PUF must be different with the output y' of PUF' for the same input x.
- *Pseudo-randomness:* PUF should show the characteristic of a pseudo-random function and it is unfeasible to break.
- *Tamper evidence:* PUF irreversibly changes in case of any attempt of physical access.

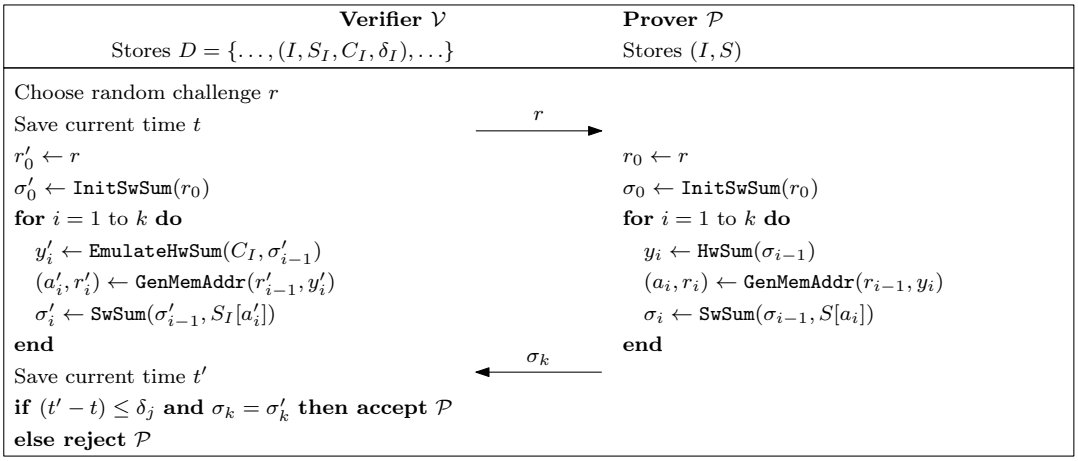

Figure 4.2: PUF Based Attestation Scheme in the Paper [5].

The scheme of the PUF based software attestation is given in the figure 4.2. The cheksum and the execution time is verified as in the software based attestation schemes. The only difference from software based attestation schemes is HwSum() function, which is generally a cryptographic HMAC function. HwSum() function calculates a hardware checksum using the PUF that is integrated to the platform. As a result the software based attestation scheme is bind to the respective hardware platform in order to provide secure remote attestation.

Verifying the execution time guarantees that no extra operation like malicious execution has been made by the remote computing platform. Hardware attacks like accessing a faster interface between the CPU and HwSum() or performance increase of HwSum() function will be detected by the tamper evident PUF. In addition PUF is unclonable and platform specific. For these reasons the integrity is assured.

Although everything seems very clear, PUF based software attestation has some design complexities. The verifier should know the responses of the platform specific PUF before any challenge is given while this must be unfeasible for the adversary to be secure. Schulz et al. [5] discusses two solutions for this challenge: (1) Using *emulatable PUF*, whose responses can be set up a mathematical model for the unknown challenges [92, 93]. (2) Creating a database *D* of the Challange Response Pairs (CRP) sets before challenger's platform is deployed. However, simulating all CRP elements must be unfeasible for an adversary therefore the database needs to be very large. Since producing a big database and searching for the elements of each

iteration step, shown in the figure 4.2, is cost expensive. Two approaches are suggested to reduce the database. (a) Instead of saving intermediate results of the iteration, shown as σ_i in the challenger part of the figure 4.2, only the responses derived from a random offset q can be produced. Then a subsets of the database D can be created depending on offset q, challenge r. (b) Creating a mutually authenticated and confidential channel between the challenger and the verifier. The verifier can send two random offset q instead of one at the start of attestation. Then challenger can produce an additional subset depending on the second random offset q. As a result the subset database can be updated after each successful attestation. However this will create an additional load for the communication channel

There is no published implementation for the PUF based software attestation system proposed by Schulz et al. [5]. Implementing a PUF hardware that gives same responses for each inputs is not an easy task. Moreover, producing a database of challenge response database or setting a PUF to to a mathematical model needs extra effort. In addition, security of the scheme is based on the challenge response sequences, which are only known by the verifier. Although this system is shown as promising [11], its implementation is hard and it has certain shortcomings.

4.4.2 Self protecting modules of Strackx et. al.

Strackx et al. [2] proposed a lightweight method called self-protecting modules (SPM) in order to gain isolation for subsystems of same processor and memory space. The SPM architecture needs simple hardware changes to the system while these changes provides strong assurance for the secrecy of the keys. Software is used to gain cryptographic capabilities.

SPM is a structured area of memory having three memory sections, which are SSecret, SPublic and SEntry as seen in the Figure 4.3. SSecret section keeps the sensitive information such as cryptographic keys, which should be prevented from access of an attacker. SPublic section is non-executable and includes non-secret memory area such as regular program code or data that can leak. All of the memory sections can only be entered from specified entry points that are stored in the SEntry section. In addition, rights of accessibility of memory sections are restricted depending on the

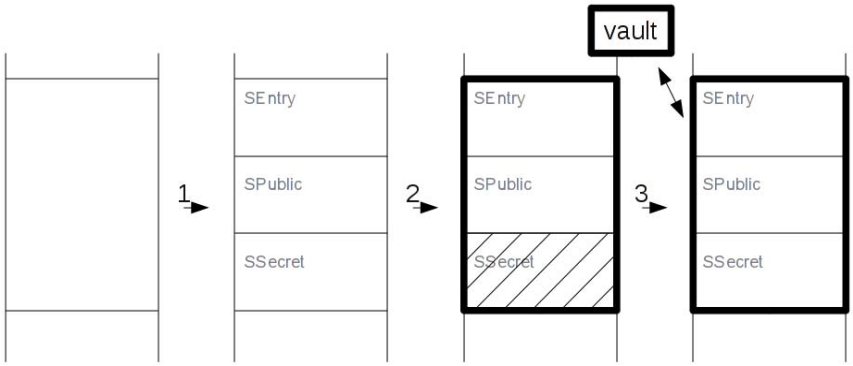

Figure 4.3: Initialization of Self Protecting Module (SPM) [2].

Table 4.1: Memory access control matrix of Self Protecting Module (SPM) [2].

from\to	SEntry	SPublic	SSecret	unprotected
SEntry		x		
SPublic	rx	rx	rw	rwx
SSecret				
Unprotected/other SPM	rx	r		rwx

current address of PC. The memory access control matrix of SPM is given in the Table 4.1. For instance, SPublic section can't be executed from outside of the SPM memory region and can only be executed from SEntry or SPublic sections.

One of the important contribution of this work is the program counter (PC) based memory access control model. An additional hardware reads the PC and its previous value in order to restrict the memory accesses and rights to read, write or execute operations.

The creation of an SPM has 3 steps as shown in the Figure 4.3. First, SPublic and SEntry sections are loaded to memory. Second, boundaries of the SPM are defined, memory access control protection is enabled and secret section is initialized with zeros. The created SPM can be authenticated and any corruption in the step 1 can be detected after the second step. SSecret section can be loaded if the created SPM is correctly authenticated by another trusted SPM called *vault*. Likewise destruction of an SPM has a similar protocol. Extensibility is supported using a trust chain like procedure after first SPM is loaded at boot time.

The isolation of modules and secure interaction is protected and provided with similar protocols by taking advantage of SPM structure. It is forced that an SPM can be called

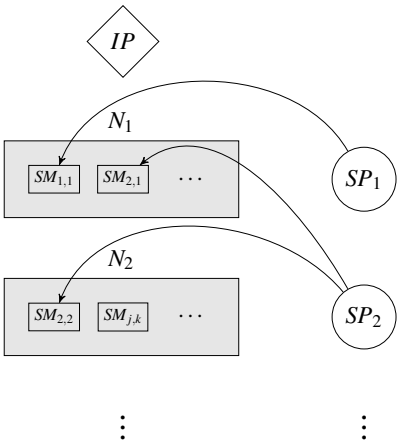

Figure 4.4: The IP infrastructure of Sancus [6]. N_i: sef of microprocessors administrated by IP, SP_j: set of software providers, SM_j,k:software modules

from another only from an entry point if the connected SPM has a correct identity. Moreover, integrity and confidentiality of parameters and return values of SPM is protected.

More detailed information of the scheme can be found in the paper [2]. No implementation example of the scheme is presented in the literature. However, the idea of creating isolated modules using a lightweight program counter based hardware have been considered as very powerful and promising [6,7,10,11]. Moreover, different architectures using the same idea have been proposed, e.g. El Defrawy et al. SMART [7], Noorman et al. Sancus [6], Avonds et al. Salus [94].

4.4.3 Sancus of Noorman et al.

Sancus [6] is proposed by Noorman et al. as an architecture that provides isolation and extensiblity without need of any trusted computing base TCB software. The scheme uses the idea of PC based memory access control and only needs very minimal hardware additions. Sancus has an infrastructure called *IP*, which contains set of microprocessors that are represented as *nodes N_i*. *SP_j* represents third-party *software providers*, which deploys *software modules SM_j,k* to nodes administrated by IP.

As can be seen in the Figure 4.4, the scheme provides execution of software modules of different stakeholders on the same node of the network. In addition, strong assurance of untampered execution is provided for each party so that their modules can be run

untampered. These properties of the scheme bring secure extensibility for third-party software over a network of low-end processors.

Preventing different modules to interfere in undesired way is crucial for a system to support extensibility. Many solutions are proposed to this problem for high end devices in the literature. They are classified in two topics by Noorman et. al. [6]: (1) Using virtual memory to get a specific memory space for each module and to control memory usage by the help of the operating system or an hypervisor. (2) Using memory safe virtual machine to store program modules in a memory safe space and guard the interactions of modules. However, these solutions are expensive for low cost embedded devices. It is discussed by Noorman et. al. [6] that the presence of a sizable trusted software layer is a common point for all of the proposed solutions.

Software modules in the Sancus has different sections, text section and protected sections are essential parts, as the isolated memory model shown in the Figure 4.3. Moreover, each software modules have their own identity. Each module uses three different key mechanisms: (1) Node N and an IP shares a *master key* (2) A software provider SP and a node N shares *software providers key* (3) A node N and a software module SM shares *software module key* and hardware forces this key to be only used by SM. Finally, a PC based hardware circuit checks accesses to memory sections, which is similar to the system proposed by Strackx et. al. [2].

As a result software module isolation, remote attestation, secure communication and secure linking is achieved in the Sancus. A TCB mechanism is supported with a minimal hardware without need of software. An implementation on MSP430 and a compiler that produces appropriate code is provided. Additional hardware doesn't need too many changes in the architecture of the processor and doesn't cost too much therefore the scheme can be applied to different platforms easily.

4.5 Conclusion

Countermeasures against software vulnerabilities are summarized in this chapter. Then lightweight solutions are presented as related work. Smart scheme will be explained in detail in the chapter 5. It can be seen partitioning the computing device and its

network as modules then providing isolated execution environment is a powerful idea to build remote attestation and safe execution. Moreover, using an additional PC based memory accesses control circuit is a powerful and lightweight idea to support isolated modules.

5. SMART: SECURE AND MINIMAL ARCHITECTURE FOR ROOT OF TRUST

In this chapter, the SMART architecture [7], which is proposed to provide a dynamic root of trust for low-end embedded devices, and its security considerations are investigated theoretically. First security treatment of the system and then generic implementation models are discussed.

5.1 Structure of the SMART

Design philosophy and security considerations of the SMART is discussed in this topic. There is no formal security proof or a standard that is published for these hardware software based approaches targeting the low power embedded devices at the time of this work is written. Therefore these lightweight attestation architectures can only be claimed to be secure with certain design considerations against known attack schemes. A systematic security treatment of remote attestation protocol to produce a minimum set for lightweight root of trust is presented by Francillon et. al. [95].

5.1.1 Security considerations

The Remote attestation protocol that is used in the SMART scheme is shown in Figure 5.1. Prover can be an untrusted embedded system, Verifier is the authority that gives assurance and they are connected over a network. It is assumed that a secure algorithm, e.g. HMAC, calculates a cryptographic checksum of prover's state using a key and random number nonce when the challenger requests. The key is necessary to prevent fake replies and nonce is necessary for freshness. The challenger also knows the expected checksum and verifies the prover after the checksum is controlled.

Certain assumptions have been made about the adversary in the SMART scheme [7]. The adversary can have a complete control over the system before and after attestation execution. He can control and change the software, code and data of the prover and

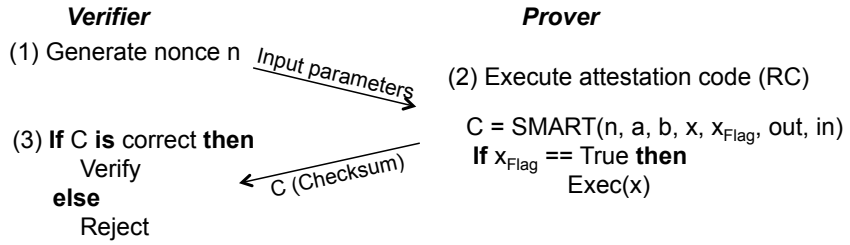

Figure 5.1: The attestation scheme at the SMART [7].

he can control the communication channel any time. However, the adversary can not perform hardware attacks on the prover, e.g. changing SMART code in ROM, learning the key using side channel attacks [7]. It is also assumed that the Prover and the Verifier shared a secret key, which can be loaded at production time or later in a secure way.

The Prover is a low-end embedded device that has been modified to have the following characteristics in the SMART scheme [95]:

- It has single memory space. There should be no separation between "kernel" and "user" memory.
- It has single thread of execution, with the exception of interrupts. It means that the device should not offer Direct Memory Access (DMA) or that DMA can be securely disabled during attestation.
- It has the ability to disable interrupts and force a region of code to execute atomically.
- It has a read-only memory (ROM).
- It has the ability to clean up memory upon device reset.
- It has a hardware-based control mechanism to prevent unauthorized access to certain memory locations.

From the definition, remote attestation must satisfy the following two properties to protect following two attack scenarios respectively [95]:

- *Attack scenario 1:* The adversary simulates the attestation.
- *Attack scenario 2:* The adversary can corrupt attestation of the state for an incorrect calculation of checksum.
- *Property of Attestation 1:* Only the attest can compute a valid checksum. This can be done by giving the only access of the correct key to the attest.
- *Property of Attestation 2:* Attestation checksum results must be different for different states s1,s2:

$$Attest(s1,k) \neq Attest(s2,k), \quad s1 \neq s2. \tag{5.1}$$

Three security objectives are summarized by the SMART team in order to protect the system from the attestation properties against these attack scenarios:

Security Objectives:

- *Prover Authentication:* Authentication of the Prover by the Verifier.
- *External verification:* Guaranteeing the Verifier that the Prover has the expected content for the questioned segment [a, b].
- *Guaranteed execution:* Assuring the Verifier that the Prover executed the expected code location x.

5.1.2 Design elements

The main goal of SMART is to realize the found necessary security objectives with minimal hardware changes. This is done with a piece of code and architectural changes to the platform that provides a set of features, which are discussed as necessary and sufficient by Francillon et. al. [95].

- *Exclusive access to key:* Attestation must have exclusive access to key. A privileged mode can be used if the processor supports multiple privileged modes and separation of memory for each process. Since low-end embedded devices generally do not have this feature, adding a piece of hardware enforces the key to be accessible only from the attestation region by checking the PC and the address bus [7].

- *No Leaks:* Attest must not leak any information related to the key except the result. Any intermediate values should be erased after the attestation to prevent leaking [7].
- *Immutability:* The attestation code must be immutable to prevent the adversary from changing the code to learn the key. In order to protect the attestation code, it is placed into a ROM [7].
- *Uninterruptibility:* The adversary can change the place of malware during attestation to hide it from detection. Attestation must run from the start to end without interrupted. This can be done by disabling the interrupts at the beginning of the code for single threaded execution [7].
- *Invocation from start:* The adversary can make attestation interruptable by invoking the code from anywhere or he can check the sanity checks on input parameters. Therefore the attestation code must be forced to run from the start [7].

Both uninterruptibility and invocation from start properties together provide atomic execution that mostly protects from scenario 2 attacks. Exclusive access, no leaks and immutability are necessary properties to prevent scenario 1 attacks. Using the following four main building blocks for the Prover, SMART architecture can be realized on the low cost embedded devices [7].

Building blocks of the SMART:

- *Attestation Read-Only Memory:* Attestation code in ROM. Key only accessible from here.
- *Secure Key Storage:* Memory region inside CPU to store key.
- *MCU Access Controls:* Custom hardware to control exclusive access to key and atomic execution.
- *Secure Reset And Memory Erasure:* Secure reset mechanism with reliable and secure memory erasure

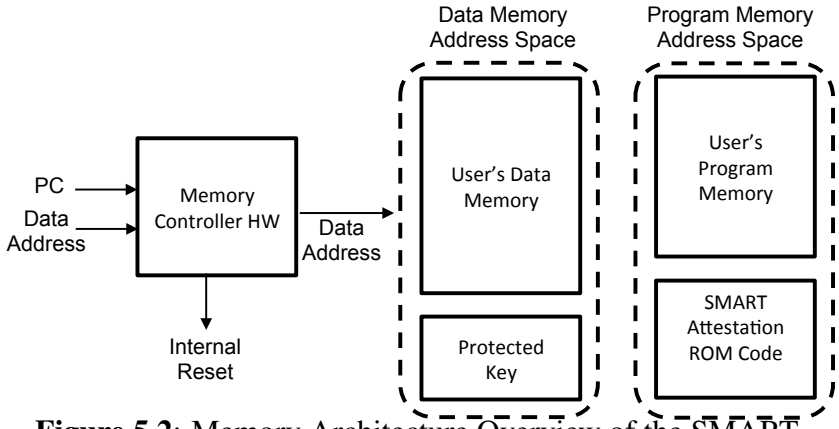

Figure 5.2: Memory Architecture Overview of the SMART.

5.2 Architecture Design of the SMART

Special hardware features of the Prover are used in the SMART architecture. An example architecture for SMART scheme is shown in the Figure 5.2. The attestation code, which is called as SMART ROM code (RC), is put in an additional ROM that is placed in the program address space. The key is put in an additional memory that is placed in the data address space. The access to key memory is controlled by a special hardware and the key is protected from software to be changed. A secure load mechanism can be placed for key memory or a static key can be hard coded to chip.

The additional memory controller hardware has two takes Program Counter (PC) and data address as inputs and controls memory accesses. Its main job can be summarized to two objectives: (1) Forcing key accesses to be exclusive for the RC code (2) Forcing that RC code can only be invoke from the start and can only be executed atomically.

In order to gain atomic execution capability for the RC code by checking the PC, hardware should make the memory access controls as shown in the Algorithm 5. Additional hardware trigger the internal reset flag in case of any violation.

Algorithm 5 The Atomic Execution Controls

- If the PC is in the RC but differs from the start of the RC, then the previous instruction must also be in the RC.
- If the PC is outside of the RC, then the previous instruction must be outside of the RC, except for the last instruction of the RC.

The attestation program (RC algorithm) pseudo code is presented in the Algorithm 6. A Keyed-Hash Message Authentication Code (HMAC) [38] or public key digital signature algorithm can be used for the checksum calculation. Attestation starts after Verifier sends input parameters to the prover. Then, RC computes a checksum using random number nonce over the memory region [a, b] of prover's memory. The random number must have sufficient bit length to keep the result unique enough for each request. Fresh and secure checksum calculation yields *prover authentication*. Next, RC passes control to the function placed at the address x if the xflag is set. Otherwise user program is returned after interrupts are restored. This yields *guaranteed execution* [7].

Algorithm 6 SMART Attestation Algorithm in ROM (RC Code)

input: a, b: start/end addresses for attestation
x: address to jump to after attestation
xflag: decision flag to jump x address after attestation
n: nonce number sent by the verifier
out: output address to store checksum
in: (optional) input parameter for the function at the x address

output: C: checksum

body:
1) Disable interrupts
2) Check validity of the out address and stack pointer
3) Check validity of the x address
4) Compute the checksum C, using HMAC of range [a, b] and nonce
5) Clean all temporary variables from memory
6) Write the checksum to the output address
7) if *xflag== 1* then // Check if x flag is set
// Set Jump address to the function at the x with its input in
Prepare Jump (x,in);
else
Restore interrupts;
end
8) Return to targeted address // Jump to x or upper function.

There are some certain rules that a designer have to be careful about in the architecture design, which are mainly discussed by Francillon et. al. [95]:

A denial of service (DOS) attack can be realized by setting the termination point of the attestation to the RC code. An infinite loop can be targeted by an adversary if the

function x is set to the last instruction of attestation code. Since PC never leave RC, no violation will be detected. Although verifier is assured, the program of prover never executes. In addition, SMART can be invoked partially by addressing x to somewhere in the RC code [95]. Since the program counter never leaves from the RC region the attestation can be compromised without any violation. Therefore, termination point x must be controlled at the start of attestation code. This control is added as 3rd step, which is not shown in the SMART paper [7].

Output address, stack pointer must be controlled at the start of the algorithm to avoid vulnerabilities [7, 95]. Stack pointer can be addressed to an IO or an adversary controlled memory region that will not be cleaned after RC execution so that information about attestation and key can be leaked. Output address can be addressed to stack in order to corrupt that region, which is used by RC at the time of attestation, when output is written [7].

Interrupts must be disabled for the architectures that uses a processor whose interrupt routines are not fixed addresses can be set manually. The reason is that RC can be partially called if an interrupt routine is addressed to RC code area and invoked at the time of RC execution. On the contrary a violation will be detected in case of a triggered interrupt if interrupt routines have fixed addresses outside of RC code and a secure reset mechanism will be set. Although disabling interrupts are discussed to be optional by the SMART paper [7], this design rule is added by Francillon et. al. [95]. Therefore disabling interrupts are given as the 1st step of the Algorithm 6.

Another design rule is that SMART can not reserve data memory, it can not use global variables and the heap to avoid relying on trusted data [7]. So, RC code can only work on the stack using local variables and registers.

Finally, exit protocol of attestation that is given by the SMART paper [7] is modified. A jump protocol is added instead of a call procedure at the step 7 of the Algorithm 6. Although a call procedure is given, a jump procedure is described in the text of SMART 5. According to the architecture algorithm can only exit from the one last instruction. In case of a function call in the 7th step, the RC code would have two exit points. The hardware can be modified in this way however complexity of the atomic

execution hardware increases. Therefore 7th step is optimized for a better design by adding a prepare jump code. In this way, algorithm changes the return address of RC code with jump address x if necessary conditions are satisfied. Then algorithm exits from last instruction.

5.3 Conclusion

In this chapter, formal description and principles of SMART architecture design is explained by using papers [7, 95]. Weaknesses and implementation details of the scheme is discussed and design rules of the architecture are explained. The attestation code procedure is updated according to discussed design rules and presented as a final pseudo code in the Algorithm 6. Abstract design of the SMART architecture is discussed and meaning of additional hardware is summarized.

6. SOFTWARE DESIGN

Software implementation of the SMART scheme is discussed in this chapter. First a hash function is chosen for the design of checksum function. The chosen hash function should be lightweight since the code will be executed in each attestation as an additional burden to real purpose of the processor.

A comparison of hash functions is published by Balasch et. al. on AVR AtTiny85 microprocessor is given by Balasch et. al. [8,96]. Comparision results of SHA3 finalist are shown in the Figure 6.1 and and lightweight hash algorithms are given in the Figure 6.2.

Keccak algorithm is chosen as SHA3 standard [36]. Moreover, the code size of Keccak[1600] is minimum and its hashing speed is third best when it is compared with other SHA3 finalists. It can observed from the Figure 6.1 that the marginal difference between keccak and the fastest algorithm very smaller than the fourth algorithm. In addition, a lightweight version as Keccak[400] has second best speed and third minimum code size comparing to other lightweight hash algorithms as can be observed from the Figure 6.2. In fact, marginal differences of Keccak[400] with the best of lightweight algorithms are very small.

A keyed hash function is necessary for the SMART scheme. Keccak as SHA3 standard is resistant against length extension attacks on the contrary of SHA1 and SHA2 standards. Therefore, a padding scheme like HMAC is not necessary for SHA3 in order to build a keyed hash construction (MAC). Only prepending the key with the message is enough MAC constructed SHA3 [36].

For these reasons a lightweight version of Keccak function is chosen for the design of SMART code. In Keccak algorithm, mathematical operations are applied to a state array that has 25 elements after the message is initialized to a part of it as will be explained in this chapter. Most efficient way to use this state array is to choose

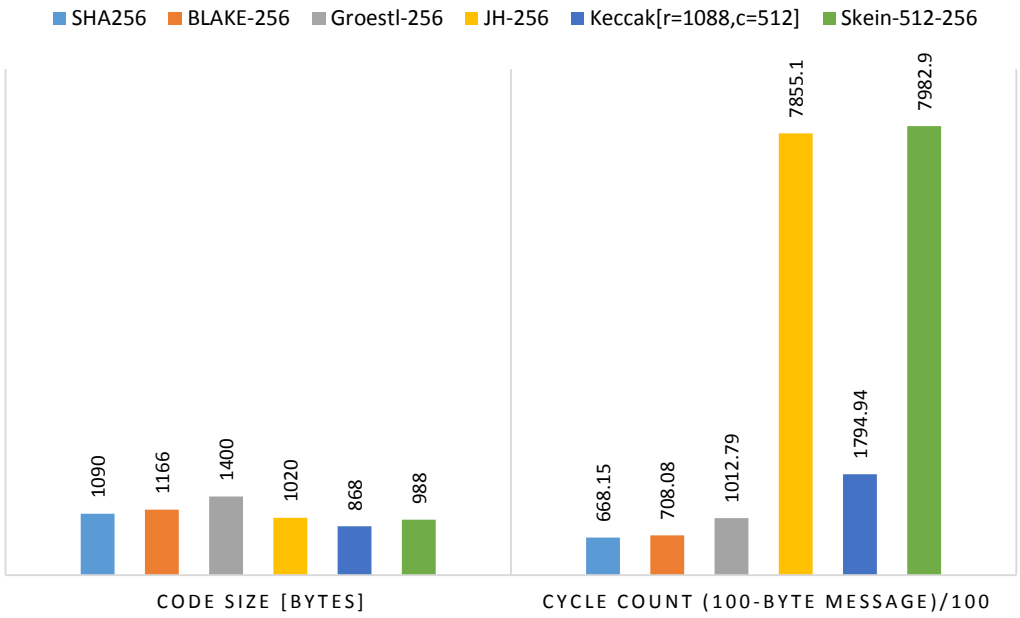

Figure 6.1: Comparision of code size and cycle count/bytes (100 bytes length messages) of implementation of SHA3 finalists on AVR AtTiny microprocessor [8].

its element size equal to word length of the microprocessor. Since 8051 is an 8 bit microprocessor, the size of the elements of state array is chosen as 8 bit. As a result Keccak[200] (Keccak[r=40, c=160]) is chosen for the keyed hash function of SMART code.

This chapter starts with the explanation of hash functions and the Keccak algorithm. Then, the implementation of Keccak on 8051 microprocessor is discussed and the final design of the attestation algorithm is explained. Implementation details for the given platform and chosen compiler is explained. Trade of between code size and speed is given. Moreover effects of secure design rules given in Chapter 5 on the performance of code is discussed.

6.1 Keccak Algorithm (SHA3)

Keccak is a family of sponge functions, which is a function that uses sponge construction, that rely on permutation. The sponge construction and the Keccak algorithm are explained in the next subchapters.

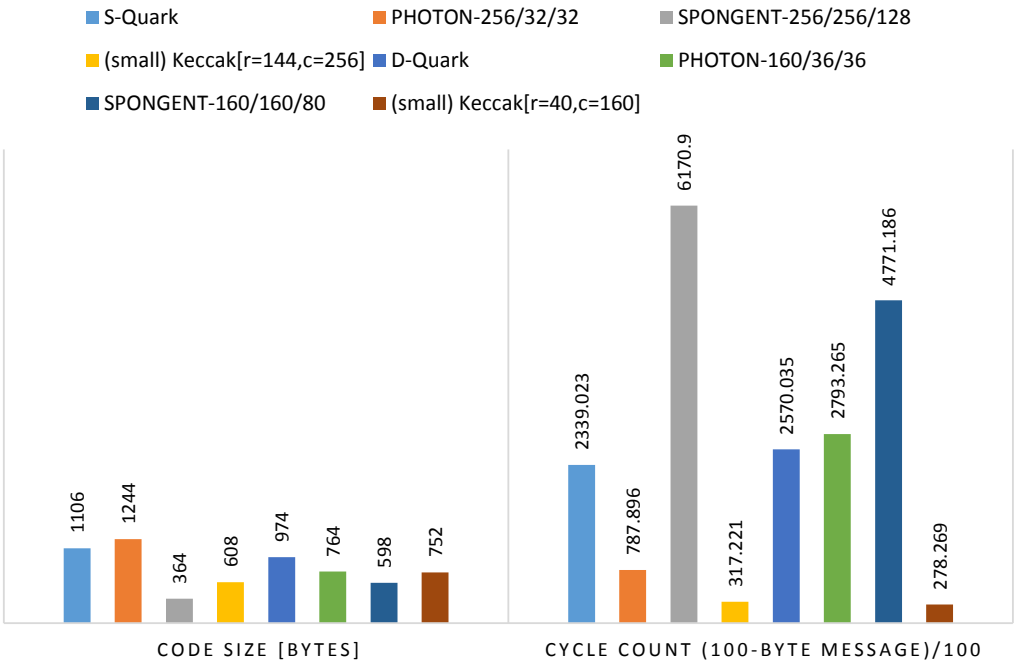

Figure 6.2: Comparision of code size and cycle count/bytes (100 bytes length messages) of implementation of lightweight hash functions on AVR AtTiny microprocessor [8].

6.1.1 The sponge construction

In cryptography, the sponge construction is an iterated construction based on a fixed length permutation f operating on a padding rule to build a function F mapping variable length input to variable length output [97]. It operates on fixed number of b bits using a state of b=r+c bits. The value b, r and c are called the width, the bitrate and the capacity respectively.

As seen in the Figure 6.3 the construction is made with padding and two phases which are absorbing and squeezing. In the initialization and padding phase, the input message is padded with a reversible padding rule and split into bitrate width (r bits) blocks. The b bits width state is initialized to zero. During the absorbing phase, the partitioned r bits input message blocks are XORed with the first r bits of the state and processed through the function f. After all the blocks are processed similarly, the squeezing phase starts. In the squeezing phase, the first r bits of the state is returned and concatenated

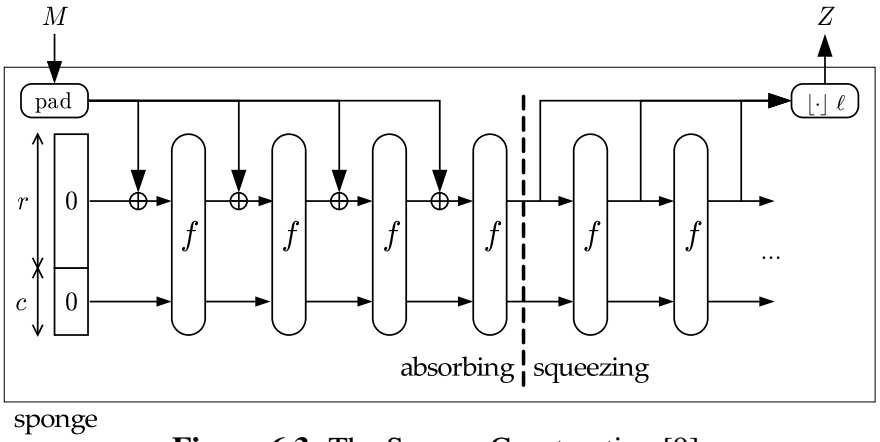

Figure 6.3: The Sponge Construction [9].

with the previous one to generate output and the state is processed through the function f repeatedly until the desired output length is reached.

The sponge constructions can be used to build hash functions, stream ciphers, pseudo-random bit generators and they have many advantages [97]:

- The flexibility to choose the adequate bit rate/capacity values while using the same permutation
- Simple security claims: The expected complexity resistance level of the sponge construction is $2^{c/2}$ [98].
- It has a variable length output meaning that the output bit length can be chosen freely.
- Disentangled from the output length: the output length doesn't determine the security level.
- It is secure against generic state recovery attacks.

6.1.2 Keccak overview

Keccak is a family of hash functions based on a sponge construction that uses Keccak-f permutation function as fundamental. The pseudo-code for the Keccak[r,c] hash function is shown in the Algorithm 7 where the parameters c is the capacity, r is the bitrate and M is the message whose length is multiple of the 8 bit. The algorithm works same as sponge construction with its padding, absorbing and squeezing parts. The S

is denoted for the state array and P_i is used for the r bits width blocks of the padded message while the || symbol represents the concatenation.

Algorithm 7 The pseudo-code of the Keccak[r,c] [9].

KECCAK$[r,c](M)$
  PADDING
  $P = M||$0x01$||$0x00$||\ldots||$0x00
  $P = P \oplus$ 0x00$||\ldots||$0x00$||$0x80

  INITIALIZATION
  $S[x,y] = 0$, $\forall(x,y)$ in $(0\ldots4, 0\ldots4)$

  ABSORBING PHASE
  for every block P_i in P
    $S[x,y] = S[x,y] \oplus P_i[x+5y]$, $\forall(x,y)$ such that $x+5y < r/w$
    $S =$ KECCAK-$f[r+c](S)$

  SQUEEZING PHASE
  $Z =$ empty string
  while output is requested
    $Z = Z||S[x,y]$, $\forall(x,y)$ such that $x+5y < r/w$
    $S =$ KECCAK-$f[r+c](S)$

  return Z

There are seven Keccak-f permutations, indicated by Keccak-f[b] where the permutation width b=r+c defines the state of the sponge function and $b \in \{$*25, 50, 100, 200, 400, 800, 1600*$\}$. The state is an array of 5 $\times$ 5 lanes whose bit length w is *b/25 ($w \in \{$1,2,4,8,16,32,64$\}$)*. The iteration number n_r of the Keccak-f[b] function is calculated by n_r = *12+2*$\times l$ where l=*w/2* and this gives a set of round number $n_r \in \{$*12, 14, 16, 18, 20, 22, 24*$\}$ for the set of permutation width b.

The pseudo-code of the Keccak-f[b] permutation algorithm is shown in the Algorithm 8. The state array is denoted by A and $A[x,y]$ shows a particular lane in that state. The rotation constants $r[x,y]$ and the round constants $RC[i]$ of the algorithm are shown in the Figure 6.4. The rotation constants are given in 64 bit hexadecimal format and it can be truncated for a smaller w. $B[x,y]$, $C[x]$, $D[x]$ are intermediate variables. All the operations of the indices are done in modulo 5. Bitwise cyclic shift operation is shown with $ROT(w,r)$, which moves a bit from position i into position $i+r$ in the modulo lane size. The Keccak-f algorithm inputs and outputs state array

Algorithm 8 The pseudo-code of the Keccak-f[b] [9].

KECCAK-$f[b](A)$
  for i in $0 \dots n_r - 1$
    $A = \text{Round}[b](A, \text{RC}[i])$
  return A

Round$[b](A, \text{RC})$
  θ STEP
    $C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4]$, $\forall x$ in $0 \dots 4$
    $D[x] = C[x-1] \oplus \text{ROT}(C[x+1], 1)$, $\forall x$ in $0 \dots 4$
    $A[x,y] = A[x,y] \oplus D[x]$, $\forall (x,y)$ in $(0 \dots 4, 0 \dots 4)$

  ρ AND π STEPS
    $B[y, 2x+3y] = \text{ROT}(A[x,y], r[x,y])$, $\forall (x,y)$ in $(0 \dots 4, 0 \dots 4)$

  χ STEP
    $A[x,y] = B[x,y] \oplus ((\text{NOT } B[x+1,y]) \text{ AND } B[x+2,y])$, $\forall (x,y)$ in $(0 \dots 4, 0 \dots 4)$

  ι STEP
    $A[0,0] = A[0,0] \oplus \text{RC}$

  return A

	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	25	39	3	10	43
$y=1$	55	20	36	44	6
$y=0$	28	27	0	1	62
$y=4$	56	14	18	2	61
$y=3$	21	8	41	45	15

RC[0]	0x0000000000000001	RC[12]	0x000000008000808B
RC[1]	0x0000000000008082	RC[13]	0x800000000000008B
RC[2]	0x800000000000808A	RC[14]	0x8000000000008089
RC[3]	0x8000000080008000	RC[15]	0x8000000000008003
RC[4]	0x000000000000808B	RC[16]	0x8000000000008002
RC[5]	0x0000000080000001	RC[17]	0x8000000000000080
RC[6]	0x8000000080008081	RC[18]	0x000000000000800A
RC[7]	0x8000000000008009	RC[19]	0x800000008000000A
RC[8]	0x000000000000008A	RC[20]	0x8000000080008081
RC[9]	0x0000000000000088	RC[21]	0x8000000000008080
RC[10]	0x0000000080008009	RC[22]	0x0000000080000001
RC[11]	0x000000008000000A	RC[23]	0x8000000080008008

Figure 6.4: The r[x,y] cyclic shift offset and the RC[i] round constants of Keccakf[b] algorithm [9].

and processes it with some mathematical operations in five states that are theta, rho, pi, chi and iota (θ, ρ, π, χ and ι).

6.2 The Implementation of the SMART Attestation Algorithm on the 8051

This topic covers the software design of the attestation algorithm of the SMART architecture for the 8051. It starts with memory management of the 8051 and examines the implementation of the Keccak algorithm by looking at three parameters, the memory consumption, the code size and the speed of different codes, which are written in C and assembly (ASM) programming languages.

Table 6.1: Some of the memory type specifiers of the SDCC and Keil for the 8051 [3].

Memory Type	Description
code	Program memory (64 KBytes); accessed by opcode MOVC @A+DPTR.
data	Directly addressable to first 128 bytes of internal data memory (fastest access).
idata	Indirectly addressable internal data memory (Using R0, R1 registers); accessed across the full internal address space (256 bytes).
xdata	External data memory (64 KBytes); accessed by opcode MOVX @DPTR.
pdata	Paged (256 bytes) external data memory; accessed by opcode MOVX @Rn.

6.2.1 Memory management

Looking at the memory management of the Intel 8051 is important to understand the time consumption of the algorithms and design choices. As discussed before, the Intel 8051 has two types of memory that are program memory (ROM) and data memory (RAM) with an address bus of 16 bit and 8 bit respectively. In addition to its 256 bytes of internal RAM as data memory, it can use an 256 KBytes of external RAM. Since the 128 bytes of its internal RAM is used for the Special function registers, only its first 128 bytes can be used for program data memory. Moreover, the first 32 bytes (4 banks with 8 registers) of it are the registers of the processor.

SDCC and Keil compilers use some memory specifiers to assign a variable to a specific or an implicitly assigned area. Relevant specifiers are shown in the Table 6.1.

The access speeds to these regions are different. The cyle refers to instruction cycle for the following of the thesis and 1 instruction cycle is 12 clock cycle for the 8051 architecture. The fastest access is possible to internal RAM, while the slowest one is to ROM. The MOVX, MOVC instruction takes 2 instruction cycles and regular MOV instructions take 1 cycle. The DPTR register can be loaded in 2 cycles by using DPH and DPL registers (1+1=2 cycle) or direct initializing. So, regular external RAM accesses takes 4 cycles using the xdata specifier and 3 cycles using the pdata specifier (only the first 256 bytes), while direct and indirect internal RAM accesses take 1 and 2 cycles respectively. The compilers use these features and give three compiling options as seen in the Table 6.2.

Table 6.2: Default memory usage for the compiling memory models [4].

Memory Model	Parameters and Automatic Variables	Default Global Variables	Default Constant Variables	Default Pointer Size
Small	data	data	data	3 bytes
Compact	pdata	pdata	pdata	3 bytes
Large	xdata	xdata	xdata	3 bytes

In default mode the SDCC and the Keil compilers assign every variable as global and address them to the specific locations in RAM. The assigned variables are used from the constant RAM addresses instead of referencing from the stack pointer. Since access of any variable in the internal RAM becomes as fast as registers, better performance is gained by this way. Stack is only used for parameter passing between functions. In fact, registers are the globally addressed first 32 bytes of the internal RAM in 8051.

6.2.2 Design choices

Because of the security concerns of the SMART architecture, only the internal RAM can be used from the attestation code as discussed before. The processor must have the necessary area for the attestation code before its execution. The available internal RAM area is the total of the 32 bytes of registers and 96 bytes of stack. The user application stops while the execution of the attestation code. Considering the constraints of the platform and the SMART architecture, the speed and the memory consumption are very critical for a compact SMART design and a lightweight algorithm should be chosen for the attestation code.

In addition to fact that the Keccak (SHA3) is the last standard of the secure hash algorithms, the speed and the memory usage of the algorithm is one of the bests between the finalists. Therefore, a lightweight version of the Keccak algorithm is decided to be used for the attestation code. The bit rate parameter r and the capacity parameter c are set to 40 and 160 respectively. This means that the security level is 2^{80}, the algorithm will process 40 bits at a time and the lane size of the inner state is 8 bit, which gives the usage of 25 bytes internal state. The width of the output hash value is chosen as 224 bit, which is the shortest in the standard, but changing it to a bigger

value effects the cost of the algorithm such little for the long messages that it can be ignored.

SDCC is used as the compiler since it is free. All of the necessary operations for this design can be done and optimized efficiently when assembly is also used.

Some design tricks are made for an efficient design. Registers can't be used directly in SDCC and the program grows enormously if local variables are used. Moreover, the code size grows and the speed decreases a lot when the stack is used to place local variables. Because of the design rules of the SMART architecture, using a global variable isn't allowed. Therefore the ability to use registers is very important. In order to use registers an unused array is defined to tell the compiler that all of the registers will be used. Then, the necessary registers are addressed and used from the SMART code. Another benefit of this method is that the attestation algorithm can use all of the register banks instead of one. In the working philosophy of the SDCC compiler, each program is responsible for keeping its own registers. So, all of the necessary variables kept in registers are put and taken from the stack by the functions before and after any function call.

6.2.3 Implementation of Keccak (SHA3) on the 8051

Implementation of the Keccak[r=40, c=160] algorithm using SDCC for the 8051 is discussed in this section. The Keccak[40,160] hash algorithm is used that uses the Keccak-f[200] permutation function.

The Algorithm 7 is the top module of the Keccak and it produces 40 bit hash result for a 40 bit message after absorbing phase. In order to produce 224 bit output Keccak-f function is called 5 times more in the squeezing phase.

As seen in the Algorithm 8, most of the mathematical operations of the sponge function is in the Keccak-f part. The Keccak-f function makes at least 12 iterations to find the hash result. The input and output of the Keccak-f algorithm is the state array. Design of Keccak-f effects the code speed and the code size significantly since it is the most time consuming part and it is called many times.

Table 6.3: The cost and the speed comparison of the implementations of Keccak-f[200] that reserves memory on 8051 using SDCC. SM: Small model design

	Keccak-f Permutation Sofware Designs on 8051				
	1st	2nd	3rd	4th (ASM)	5th (ASM)
RAM Data [bytes]	25	25	25	25	25
Registers [8-bit]	9	13	13	12	12
Total LUT Size [bytes]	76	66	18	18	18
Code Size [bytes] (SM)	466	605	904	443	547
Inst. Cycle count (SM)	76551	61683	15311	7305	5325

Table 6.4: The cost and the speed comparison of the Keccak-f[200] implementations that only uses stack and registers on 8051 using SDCC. SU: stack used model design.

	Keccak-f Permutation Sofware Designs on 8051				
	1st	2nd	3rd	4th (ASM)	5th (ASM)
RAM Data [bytes]	25	25	25	25	25
Registers [8-bit]	9	13	13	12	12
Total LUT Size [bytes]	76	66	18	18	18
Code Size [bytes] (SU)	504	697	1749	637	772
Inst. Cycle count (SU)	83277	69381	28814	9052	8188

While the Keccak top function is the controller part, the cost of the SHA3 comes from the Keccak-f function. Therefore, Keccak-f function is designed efficiently by using ASM while the top module is written in C for flexibility.

As discussed before, SDCC assigns every variable as global to the internal RAM in the small model. However, SMART attestation code can not reserve memory for its own use [7]. Indirect accessing is used in order to use stack as memory. In this method, a frame pointer is addressed to the functions and other variables are called using their references to that pointer. This method decreases the speed and increases the size of the algorithm by around of the factor two. SMART architecture can be adapted to use its own memory by making additional hardware optimization.

As a result, two versions of software implementations, which are the small model (SM) and stack using model (SU), are designed to investigate the cost of design choices. The state array is either addressed globally or written to the stack depending on the small model or not. Other variables are assigned to registers in both designs.

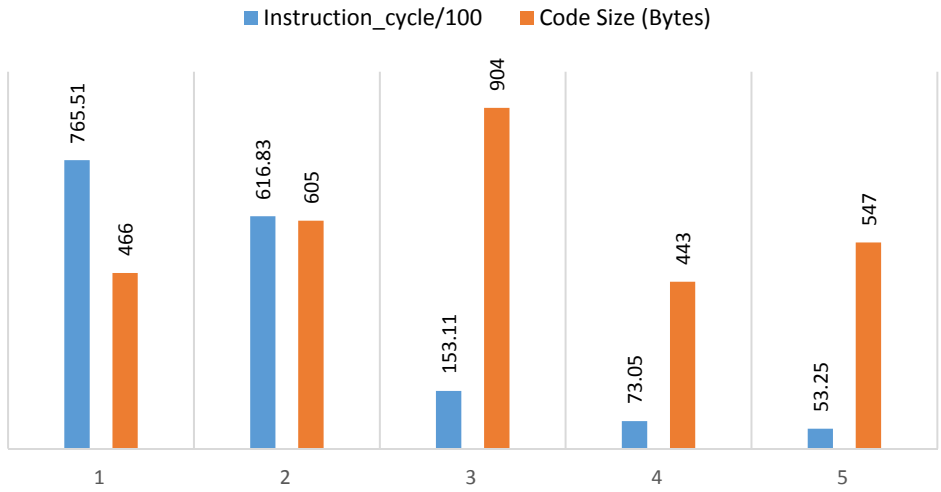

Figure 6.5: Performance comparison of Keccak-f[200] function for Small Model.

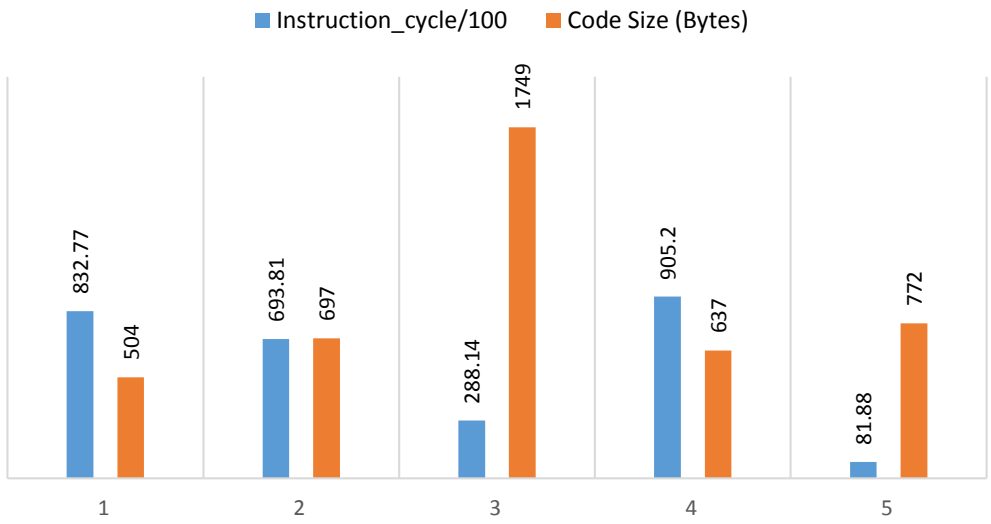

Figure 6.6: Performance comparison of Keccak-f[200] function for Stack Used Model.

As seen in the Algorithm 7 Keccak-f inputs and outputs the state array. One coding style for a fast and small program is to design one function for Keccak and Keccakf in a way that program flow is controlled by using *goto* instruction. However, it is known that using *goto* is not a good software programming way. The same performance and area properties can be reached if no time is consumed for the parameter passing. This is done with a few tricks and using ASM. The Keccak function reserves the necessary memory for all functions. Keccak-f function does not pass any parameter and it knows the address of the state array in the memory. While the small model designs uses the fix address of the state array, stack used model designs know the reference distance to the frame pointer.

10 software implementations of Keccak-f function are shown in the Table 6.3 - 6.4. Table 6.3 shows small model design (SM) implementations while the Table 6.4 shows the stack used model design implementations. First three implementations are written in C and the last two are designed in ASM. Same optimization steps are applied to program codes of the stack using model and the small model in the Table 6.3 and Table 6.4 and they follow the same program flow. However, they are different codes especially for the forth and fifth implementations, which are written in ASM.

The first code minimizes the area and it is adapted from the one named avr8.c in the reference source codes of Keccak [99]. The constants are placed to the look-up (LUT) tables and loops are used to apply same mathematical operations to the elements of LUTs. It uses 4 LUTs that are RC[i] round constants, the r[x,y] cyclic shift offsets and two tables for the indices. One of the indices table is to keep modular-5 numbers and the other one is for the state indices of the Rho-Pi phase. The LUT size in the Table 6.3 references to these tables.

In the third code, 3 LUTS (ones except RC[i]) and their loops are unrolled and optimized. These loop unrolling operations can only be done with using extra register. The loop of RC[i] constants isn't unrolled in none of implementations since it is the main iteration loop of the Algorithm 8.

Table 6.5: The Keccak-224 implementations on 8051 where r=40 and c=160. (cycle refers to instruction cycle)

	RAM data [bytes]	Registers [8-bit]	Code size [bytes]	Rate of short messages (30 byte) [cycle/byte]	Rate of long messages (1000 byte) [cycle/byte]
Small Model	31 (25+6)	15	709	1953	1532
Stack Used M.	31 (25+6)	15	1084	2740	1713

The second code is given to show a middle trade. The Rho-Pi phase isn't unrolled in this implementation. There is one more loop in addition to the iteration loop in all of these three designs and Chi, Iota and a part of Theta phases are combined in this loop.

The third code is taken as reference implementation because of its speed and ***the fourth code*** is designed with ASM optimization. Finally, more optimization is done after theta phase is combined completely and the last loop of the round loop is unrolled thanks to efficient coding capabilities of ASM. ***The fifth code*** as the final implementation in the Table 6.3 only has one main round iteration using different RC[i] constants.

The ratio of the speed and the code size between small model and the stack used model is smaller in the fourth and the fifth implementations when they are compared with the first three implementations thanks to ASM coding. R0 and R1 registers are efficiently used to access internal RAM in ASM designs. Having two more pointers give good opportunities to track the variables and access them without changing any other register for addressing. This is very beneficial especially for the stack used model implementations since a variable can only be accessed after its address is calculated using its distance to the frame pointer of the function every time.

Finally, fifth Keccak-f implementations, which are coded in ASM, are used and the rest of the Algorithm 7 is coded in C language. Two final designs are produced as given in the Table 6.5.

Implemented Keccak codes has two inputs, an input pointer and an output pointer. They are stored in the stack for both cases with a total cost of 6 bytes (2×3 bytes). In addition, three more registers are used for the indexes.

A generic pointer can address any kind of memory specifier. However, they are slower than the specified pointers. For this reason none of the pointers are defined as generic. The aim of the attestation code is to verify the code space so pointers are specified to the ROM memory. The hashing rate of long messages is faster than short messages. The difference mainly comes from the squeezing phase of the Keccak algorithm where the hash output is generated. In this phase Keccak-f permutation is called 5 times more to produce 224 bit output although the algorithm processes 40 bit.

6.3 Design of the SMART Attestation Code

The explanation of the SMART attestation code is given in Algorithm 6. The algorithm inputs pointers to the attestation region start and end addresses, pointer to the nonce, the xflag, function pointer for the jump address and input of the targeted function. The performance results of the SMART software implementations are given in the Table 6.6.

SMART attestation code is designed by obeying certain design rules as discussed in Section 5.

Designed SMART code is placed in a different ROM in the program address space. This is achieved by a regular software design that places the SMART code to a specified area. Placing a piece of code a specified area can be achieved in a few ways. Since mapping the addresses of functions is the duty of the linker, certain directives can be given to force a custom design. One of them is to use *codeseg* directive to target a function to a certain address by giving it as a *pragma* in the code or as a compiler directive during the compilation [100]. SDCC only allows two partitions of the code to be targeted in this method. The second way is to write the complete program in assembly and specify the address in the code. Inline assembly can only be used for the body of a function so it cannot be used for this aim. Another way is to compile a library file by using *libsdcc* tool.

Although generic pointers give the flexibility to address any type of memory, they call some compiler subroutines to work in the way of SDCC [100]. These subroutines are the assembly codes that check the type of the generic pointers and they are placed with

Table 6.6: Designed smart attestation codes. (cycle refers to instruction cycle)

	RAM data [bytes]	Registers [8-bit]	Code size [bytes]	Rate of short messages (30 byte) [cycle/byte]	Rate of long messages (1000 byte) [cycle/byte]
Small Model	37 (25+12)	15	1082	2768	1558
Stack Used Model	37 (25+12)	15	1472	3650	1742

the compiled program in default. However the SMART code mustn't leave its address space. Therefore, they must be copied to the area of the SMART code if they are used. This can only be done by designing an independent library [100]. Otherwise, it causes a design violation that the SMART code is not completely run in its address space. Generic pointers not only make the program slower but also make it bigger. Considering these reasons no pointer is used as generic in this implementation of the SMART code.

As discussed before the SMART can not reserve memory for its own use however a special RAM can be added to processor. In order to show the cost of this change two SMART implementations are designed using the Keccak shown in the Algorithm 7. Both designs uses registers for the variables except the state array.

Stack used model implementation is only become possible if the function is defined as *reentrant* in SDCC. Normally defining a function as *reentrant* is used to design functions that can be called safely before its previous invocation completed. However it is used in this concept since it assigns a stack area to the function.

Another requirement of the SMART is to disable interrupts if interrupt service routines can be addressed manually. Since this is the case for the 8051, interrupts are disabled at the start of the algorithm by setting the EA flag of the interrupt register. This can be done by defining a function as *critical* in SDCC [100].

The output address, the stack address and the address to jump to after the attestation must be checked as discussed in the chapter 5. Since the pointers are not generic, they can only address their specified address space. The output pointer is specified to the RAM therefore checking it becomes unnecessary. The stack is addressed to the internal RAM since the code is compiled as small model whose memory usage is given in the

Table 6.2. The pointer of the address that will be jumped after the execution is checked to prevent program to go to program space of the SMART code.

Another design rule is that any intermediate values must be cleaned from memory. Therefore all registers and RAM locations between the stack pointer and frame pointer are set to zero. At the end of SMART code execution, program flow jump to another function if the xflag is set as can be seen from the Algorithm 7, this functionality can not be gained without using ASM programming language because of the reasons that are discussed in the following.

Calling a function using a function pointer or jumping to an address using *goto* are not the solution for the necessary need. The SMART function can return the given function pointer and direct program flow to that program. However the program flow comes back to the last address of the SMART function instead of the main function after the execution of the addressed function is finished. This is a violation of the atomic execution given in the Algorithm 5.

The functionality of jumping to a given function and passing a parameter as an input to that function can be gained with ASM coding for the SMART code. An ASM code is added to change the return address that is stored in the stack at the start of the SMART code. The explanation of the program flow is given: The SMART code starts execution when it is called. The return address, showing the address that the SMART is called, is pushed to the stack as regular. In the end, the added ASM code changes the return address in the stack with the given function pointer if the xflag input is set. So program goes to the given function instead of the regular a return. The first variable of a function is passed using DPTR register in 8051. This parameter passing mechanism is also applied for the SMART manually. ASM code changes the DPTR with the given input before the SMART returns to the given function.

Prepending the key and the nonce to the message is enough for a keyed hash version of Keccak. Therefore, the result of the key of the absorbing phase, shown in the Algorithm 7, is initialized to the state array before the message is hashed. The nonce is concatenated with the message.

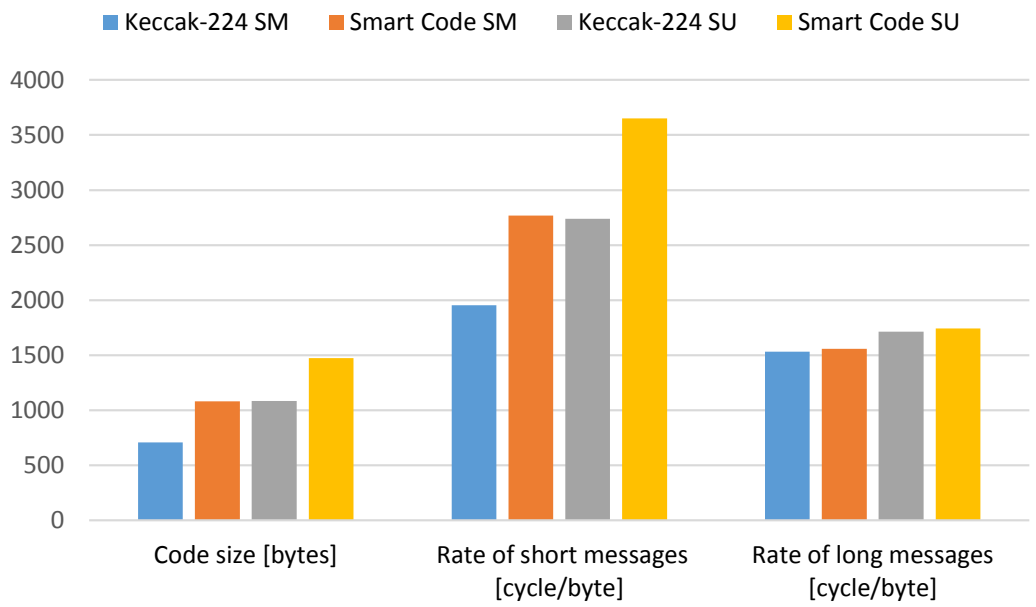

Figure 6.7: Performance comparison of SMART attestation code and SHA3 algorithm for both long messages (1000 Bytes) and short messages (30 Bytes).

The final results of SMART code design is given in the Table 6.6. The SMART implementation and the SHA3 implementation can be compared by looking the Table 6.6 and the Table 6.5 as shown in the Figure 6.7. It can be seen that the speed of the code is not changed too much while the code size is increased a little. The reason is that additional loops are used to hash the key and the nonce in the SMART design. The increase of the RAM data consumption comes from the input parameters of the SMART function since they are placed to the stack.

6.4 Conclusion

In this chapter, the software design of the SMART remote attestation architecture for the 8051 microprocessor is explained. A lightweight version of Keccak hash algorithm (SHA3) is used for the checksum function of the attestation. SHA3 algorithm is explained and trade off between the speed and code size of the algorithm is presented. 10 different Keccak-f function, which is the sponge construction of SHA3, is implemented. After the code is written using C programming language, loop unrolling and ASM inlining is used for optimization. It is seen that both code size and speed has become better after these optimizations as can be seen in the Figure 6.5 and Figure 6.5.

All design rules of the SMART scheme, which is also explained in chapter 5, are applied using the hash function. Design problems and choices are explained. At the end two different program code is produced. Although one of the design rule is that the RC code (SMART program code) can not reserve memory for itself, a program called small model is designed in addition to stack using model. It is a common compiler optimization for lightweight embedded systems to globally address all the variables to memory, the performance loss to force the program to use stack is presented.

The code size and program speed of Keccak (SHA3) function and SMART code can be investigated by looking Figure 6.7. The code size and the time that is consumed to calculate hash increase more dramatically for the stack used model than the small model. In addition, hashing speed decreases a lot more for short messages comparing to long messages. The reason is that a random number nonce has to be sent to provider by challenger as explained in Chapter 5. This random number has to be big enough to keep freshness of result and 160 bit is considered as a satisfied number. Moreover, 80 bit key, which is enough since security level of Keccak[200] is 80, is prepended to the message. As a result, hash result of 60 Bytes, whose 20 bytes are nonce and 10 Bytes are key, are calculated in order to take hash of 30 bytes of sensitive data.

No other Keccak[r=40, c=160] implementation can be found for the 8051 at the time of this thesis is written. However there are some implementation or estimation results for Keccak[1600] function in the paper [101, 102]. In addition, ASM software implementations of SHA3 finalists on AVR AtTiny85 microprocessor are given by Balasch et. al. [8, 96]. The best numbers for the 224 bits of output of Keccak[1600] are 1648 bytes/cycle for 8051 and 1794 bytes/cycle for AVR AtTiny85. Moreover, a lightweight implementation for Keccak[r=144,c=256] is given as 3629 bytes/cycle for 100 bytes length message and 2627 bytes/cycle for 500 bytes length message by Balasch et. al. [8]. Our implementation results are 1532 and 1953 bytes/instruction cycle respectively for small messages and long messages as given in the Table 6.5. Although it is not fair to compare Keccak[200] with Keccak[1600] or Keccak[400], our implementation seems efficient.

Since each instruction of Keccak-f function is optimized using ASM and top level control block is coded in C efficiently, final design of the program is considered as compact and efficient implementation.

7. HARDWARE DESIGN

Smart scheme is implemented FPGA for 8051 architecture. Dalton core [103] is modified and used for the implementation of 8051 microprocessor on a Xilinx Spartan 6 CSG324 FPGA. Hardware implementation details of the SMART scheme is discussed in this chapter. Cost, compactness and design effort of the SMART scheme are examined for this restrictive microprocessor.

7.1 The Hardware of 8051

The 8051 implementation of the California's Dalton Project, which is synchronous as same as the standard, is used in this project. The architecture of the 8051 dalton core hardware is given in the Figure 7.1. While the execution of the processor, instructions are read from the ROM by the controller hardware and op-code of the instruction is learned from the decoder. Then necessary control signals of ALU and internal RAM are set by the controller hardware in order to execute the read instruction. Execution state generally starts with a read from RAM and finishes by writing the result of ALU to RAM. In addition, external RAM is used by the controller hardware if necessary.

Implementation of a smaller version of microprocessor is presented in the following sections. Implemented core has 4 KB ROM and 2 KB external memory although they are 64 KB in the standard. In addition, the external memory is embedded into the core. Implementation details of 8051 core, which is not modified, for the targeted FPGA is shown in the Table 7.1. Implemented hardware is smaller and faster than the standard. Therefore, the cost of additional hardware shows a lower boundary for the standard implementation.

Table 7.1: Implementation Results of the Embedded 8051 Core

Number of Slice Registers	1.363
Number of Slice LUTs	2.759
Number of RAMB16BWERs	4
Maximum Clock Frequency	58.072 MHz, (min period: 17.22 ns)

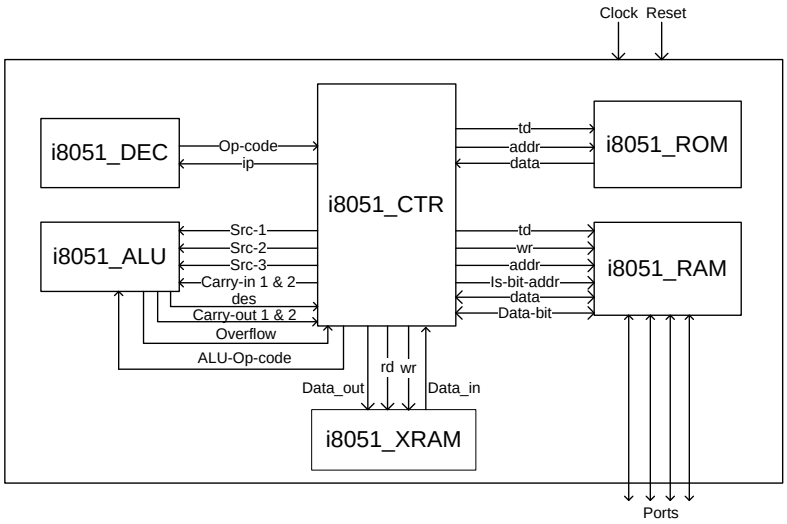

Figure 7.1: Architecture of the Dalton implemantation of 8051.

7.2 Hardware Design of the SMART

Hardware modifications to 8051 architecture are necessary as discussed in the Chapter 5. Exclusive key accesses to attestation code and atomic execution of the attestation code must be forced by an additional hardware that reads program counter (PC).

An implementation of the SMART scheme, which is very same in all details with the proposed system by El Defrawy et. al. [7] and Francillon et. al. [95], is explained in this topic. In the end implementation details for the explained hardware is presented and compared with the non modified 8051 Dalton hardware core.

7.2.1 Standard implementation of SMART architecture

An additional hardware is deployed between the controller unit and the ROM-RAM memories as shown in 7.2. The additional hardware takes data address and PC as input and outputs whether the Key or RAM data. An internal reset occurs and resets the memory in case of a violation of atomic execution and key access rules.

Top level schematic of the additional hardware is shown in the Figure 7.3. As can be seen in the figure, the data address and PC is compared with low-high values of the key and the attestation code address to learn the state of the memory accesses. It should

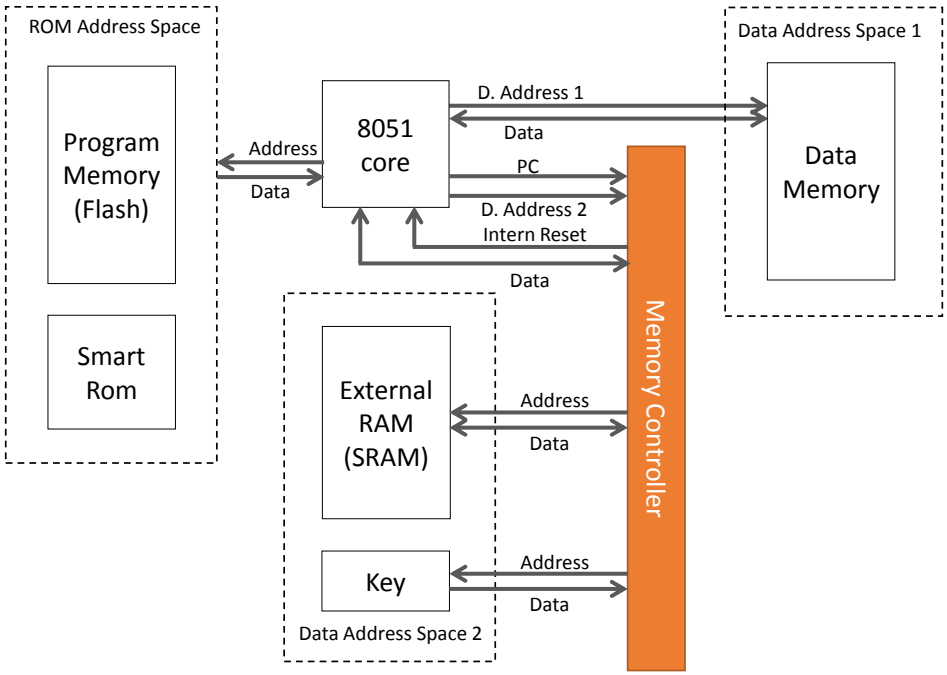

Figure 7.2: Standard Implementation of SMART Architecture for 8051

be observed that these addresses are constant. Size of the comparison hardware blocks depend on these constant addresses.

Regarding to *atomic execution* rule, smart attestation code (RC code) can only be invoked from the start and can be exit from the last instruction as explained in Chapter 5. Two checks should be realized for a PC based implementation of atomic execution control hardware as explained in Algorithm 5: (1) If the PC is in the RC but differs from the start of the RC, then the previous instruction must also be in the RC. (2) If the PC is outside of the RC, then the previous instruction must be outside of the RC, except for the last instruction of the RC.

Comparison circuits of the designed hardware can be done efficiently with a small trick. Reserving 2^k bits memory space for the additional RAM after 2^k bits data RAM of 2^n bit data memory space provides a smaller circuit that only compares high significant $(n-k)$ bits of n bits address. For instance, if 4 KB memory area is reserved for the attestation code (RC) after 4 KB program code in a 16 bit address space, looking high significant 4 bits of PC is enough to understand whether the PC is in the RC or not. However, knowing that the program flow is in the last instruction of the RC is necessary to control atomic execution. Therefore, an address called *RC_exit* is

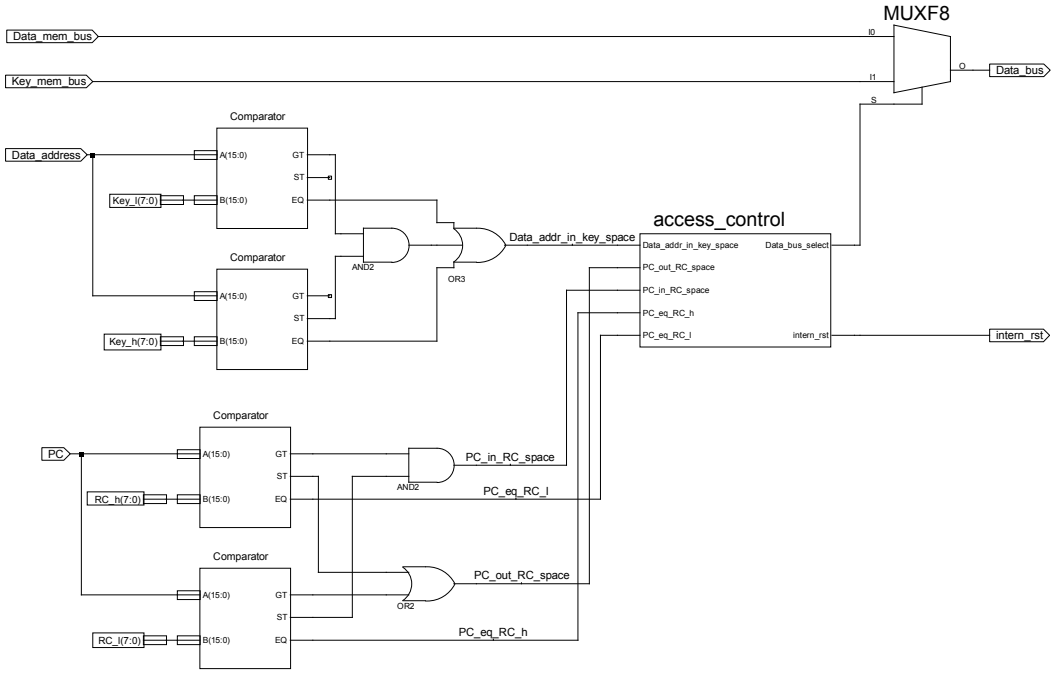

Figure 7.3: Hardware of the Memory Controller

defined for the termination address of RC and this is used instead of *RC_high* in atomic execution hardware. This is also a more realistic approach since many compilers put some constant tables or sub-functions before or after attestation code as discussed in Section 6. It should be mentioned that this is dangerous for the situations where hardware attacks are considered since an additional IO or a memory block can be integrated. Note that this optimization is not shown in Figure 7.6 in order to keep the coherency with the proposed scheme.

While the memory access_control hardware, which is given in the Figure 7.3, is mostly combinational, a memory element is necessary for the atomic execution hardware in order to keep previous value of the PC. Instead of reserving 16 Flip Flops (FF) for the previous PC and deploying relatively big comparison hardware for it, as given by the Sancus team and shown in the Figure 7.10, an efficient implementation is presented in this thesis that only uses 1 FF with a smaller logic.

Information of being in RC region in the previous instruction is stored to one Flip Flop, which is named as PCinRC in the following, for the atomic execution. In order to visualize hardware designs truth tables are given in this chapter. The truth table for the behavior of the PCinRC FF is given in the Table 7.2. PCinRC FF is set to logical

Table 7.2: Truth Table for the input of the PCinRC Flip flop

PC_eq RC_h	PC_eq RC_l	PC_in_RC space	PCinRC FF_q	PCinRC FF_d
(0)	1	(0)	1	1
(0)	(0)	1	1	1
(0)	1	(0)	0	1
All other Combinations				0

high when PC equals to the low address of RC and its output is logical zero. Then it is kept as logical high while the execution of RC. In the last instruction of RC, the FF is set to logical zero. Final design of the atomic execution hardware is shown in the Figure 7.4.

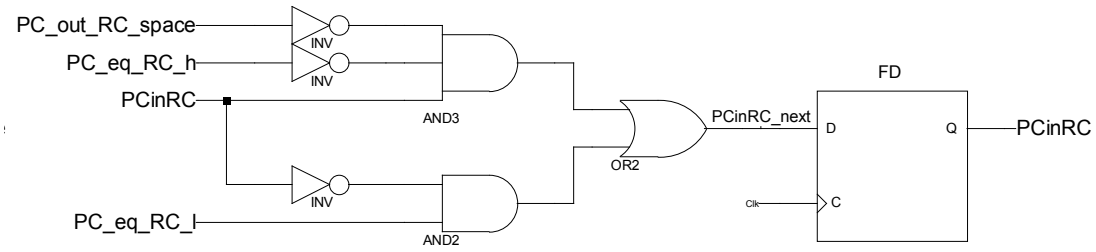

Figure 7.4: Flip flop hardware for Atomic Execution

Internal Reset signal is designed as the output of the OR logic of two reset circuits as presented in the Figure 7.5. The First reset hardware checks if the key is accessed from the RC code and the second one is the violation output the atomic execution.

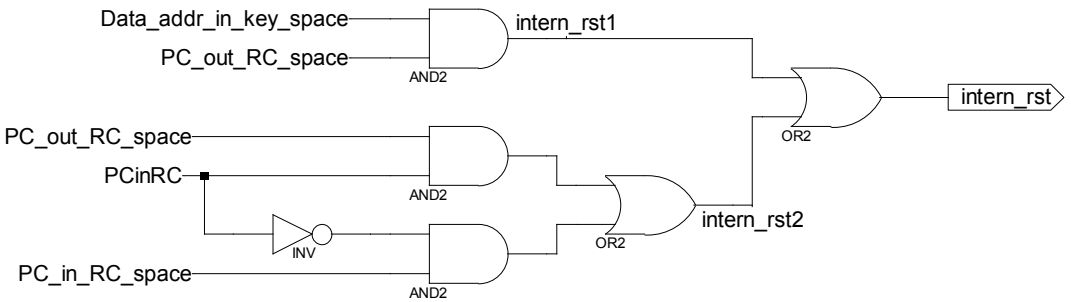

Figure 7.5: Internal Reset Hardware

The truth tables for the reset mechanism are given in the Table 7.3 and the Table 7.4. Intern_rst2 signal rises in two conditions: (1) The output of PCinRC FF is logical 1 and PC is not in the RC address space (2) The output of PCinRC FF is logical 0 and PC is in the RC address space while it is not equal to RC low or RC high addresses.

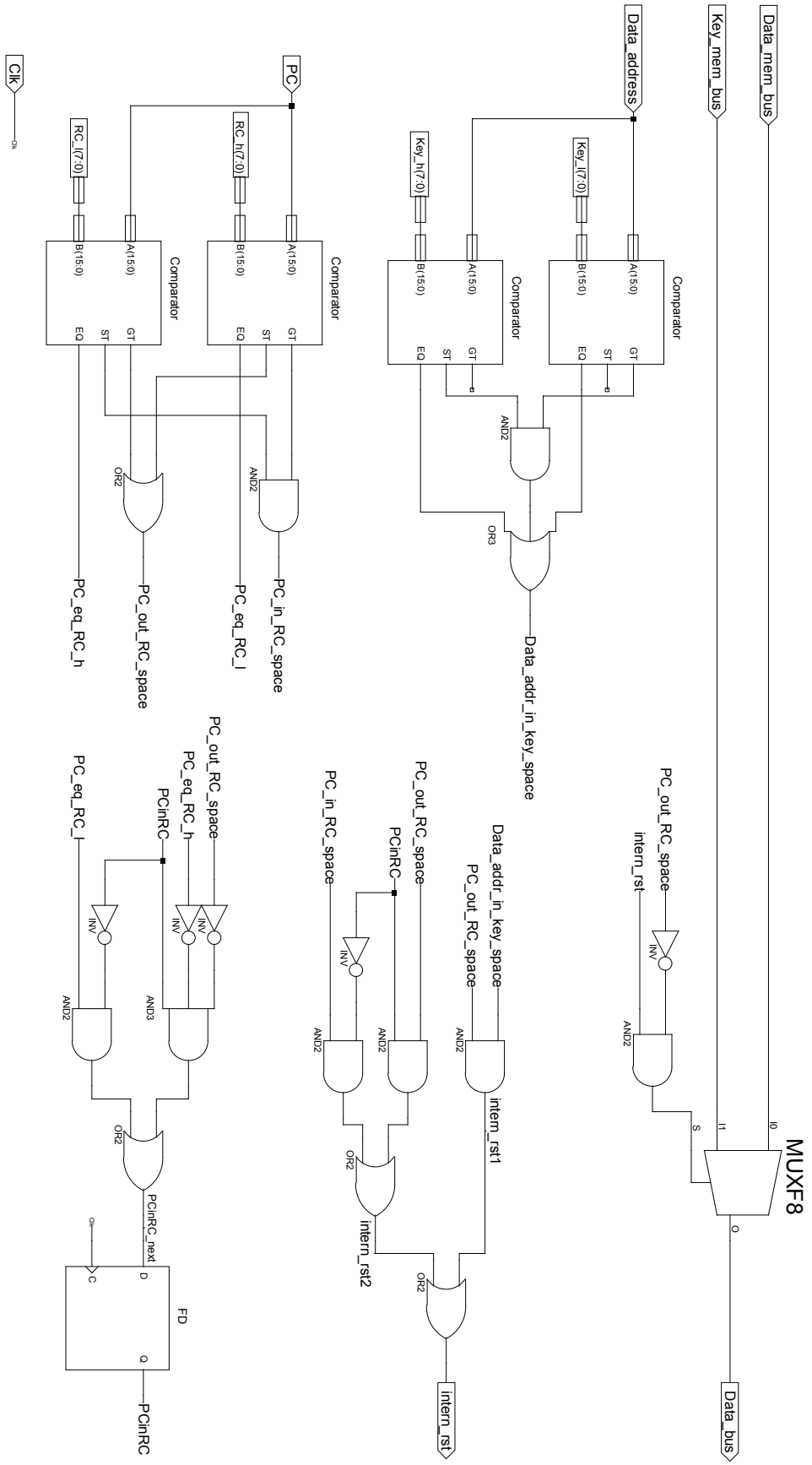

Figure 7.6: Hardware of the Memory Controller for Standard Implementation

Table 7.3: Truth Table for the Internal Reset 1 (Access of Key)

PC_out_RC_space	Data_addr_in_key_space	intern_rst1
1	1	1
All other Combinations		0

Table 7.4: Truth Table for the Internal Reset 2 (Atomic Execution)

PC_eq RC_h	PC_eq RC_l	PC_in_RC space	PCinRC FF_q	intern_rst2
0	0	0	1	1
(0)	(0)	1	0	1
All other Combinations				0

The schematic view for the additional circuit that showing all parts is given in the Figure 7.6.

The key is placed to the next address after the 2KB XRAM data memory and the key space is defined as 25 bytes. Although internal ram memory can be used faster and external RAM is convenient for the products where external ram is controlled with the input and output pots (IOs) of the processor, XRAM address space is chosen since internal memory is very limited. However, modified versions for this architecture that shows efficient ways will be presented in the following topics.

The explained system is implemented for the targeted FPGA. The implementation results are shown in the Table 7.5. It can be observed that hardware changes are very minor when the Table 7.5 and the Table 7.1 are compared. Extra ROM memories are added to store the attestation code and the key. The critical path of the microprocessor is increased because of secure reset mechanism implemented. Final critical path is given between reset input signal and internal ram. The maximum operating frequency of 58.072 MHz is reduced by negligible amount (0,008%) and the increase in the area is minor, which is 0,014% for registers and 0,047% for LUTs.

Table 7.5: Implementation results of standard SMART architecture

Number of Slice Registers	1.383
Number of Slice LUTs	2.891
Number of RAMB16BWERs	4
Number of RAMB8BWERs	0
Maximum Clock Frequency	57.577 MHz, (min period: 17.368 ns)

7.2.2 Modified SMART architecture that uses exclusive memory

According to design rules of SMART, attestation program (RC) cannot reserve memory in order to avoid untrusted data. However, it can be observed from Table 6.6 and Figure 6.7 that code size increases 36% and speed decreases 12% when stack used model (SU) is used instead of small model (SM). It should be mentioned that the SU program is designed very efficiently using ASM and an efficient variable addressing method as explained in Section 6. Therefore the difference between SM and SU would be even bigger for most of the designs. For this reason, an approach is developed for a better architecture in terms of area and speed.

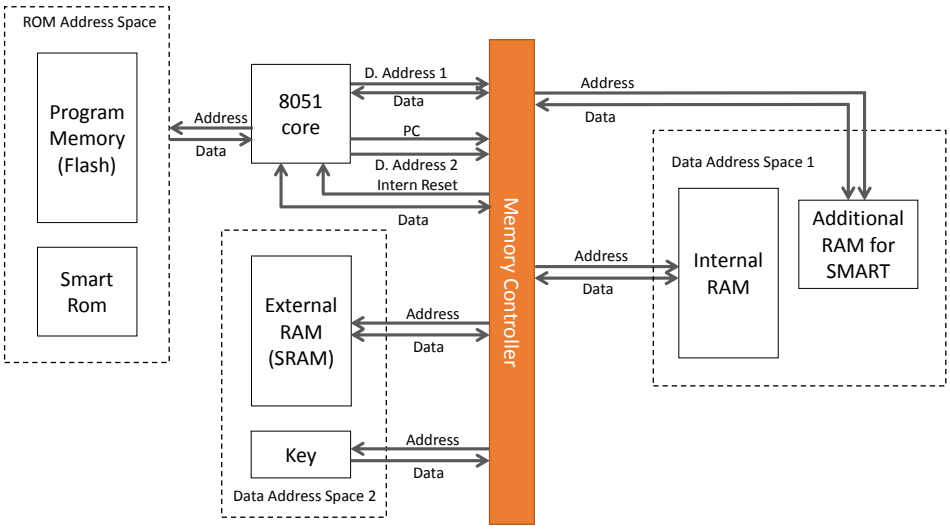

Figure 7.7: 8051 SMART architecture that uses exclusive memory

An exclusive additional RAM, which is 25 bytes for this design, is deployed into the internal RAM address space as an alternative memory for the use of RC of SMART as shown in Figure 7.9. A data selection circuit that switches the memories internal RAM and additional exclusive RAM depending on the state of PC to be in RC is deployed to input of internal RAM data bus. Exclusive memory is placed to address out of the memory spaces of user registers and special function registers in order not to crash processor settings.

In order to implement SMART to any architecture, processor's RAM must be changed with a resetable one in order to implement secure reset mechanism that resets

everything in one clock cycle. However, this will increase the power and area. On the other, hand our approach also optimizes this requirement by only deploying a small resetable memory for the use of RC.

The data selection circuit, which is shown in the Figure 7.8, only consists MUXes that is controlled with the output of the FF of additional memory controller hardware, which is shown in Figure 7.6. Therefore minor changes are applied to modify implemented SMART architecture. The MUX selects the data to send the internal memory data bus. If the PC is in the RC code and data address is in the exclusive RC RAM, exclusive RAM is Read and its output is selected. Microprocessor continue to work in other cases.

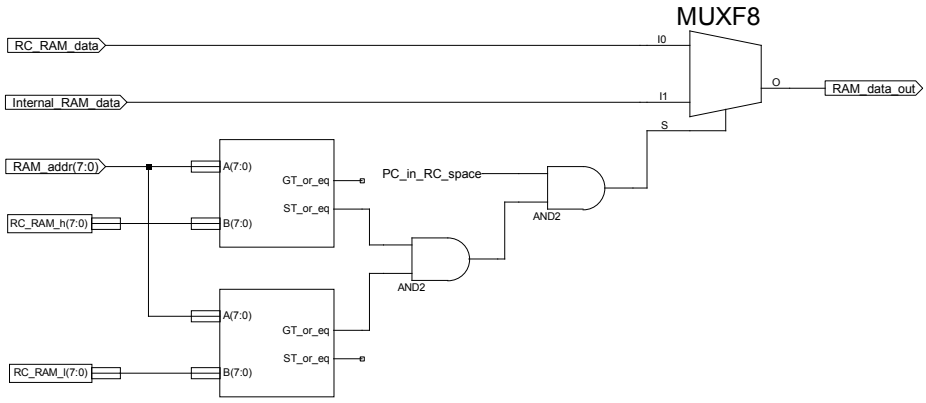

Figure 7.8: Data selection hardware of the Exclusive memory of Modified SMART Architecture

SMART team [7] conditions stack used model in order to provide minimality and avoid leakage. However, deploying an additional RAM as proposed in the Figure 7.7 for the use of the attestation code decreases total cost of additional hardware although total change in architecture seems to be increasing. However total cost of additional memory since code size of RC decreases

Table 7.6: Implementation of the modified SMART architecture that uses exclusive internal memory

Number of Slice Registers	1.386
Number of Slice LUTs	2.979
Number of RAMB16BWERs	3
Number of RAMB8BWERs	1
Maximum Clock Frequency	56.561 MHz, (min period: 17.680 ns)

The modified system is implemented for the targeted FPGA. The implementation results of hardware design is given in the Table 7.5. It can be observed that hardware change and clock change is negligible while the total RAM usage is decreased as expected when the Table 7.6 and the Table 7.5 are compared. As a result, an architecture is proposed to support memory reserving attestation code so that the system can be implemented with faster attestation algorithms. In addition, the size of the processor becomes smaller.

7.2.3 Modified SMART architecture using key in exclusive memory address space

Placing key to an alternative address like RC exclusive RAM as shown in the Figure 7.9 is a more efficient approach. key memory is placed after RC exclusive RAM in a same way as shown in Figure 7.9. Thanks to this, the unified memory area of the key and RC exclusive RAM are treated like the key address space of SMART architecture. This optimizes the comparison circuit of exclusive RAM. In addition, first internal reset circuit, which controls the key accesses, becomes unnecessary since key can only be accessed from RC code. As a platform specific result, 15 bit address comparison circuit of key is turned to 8 bit comparison.

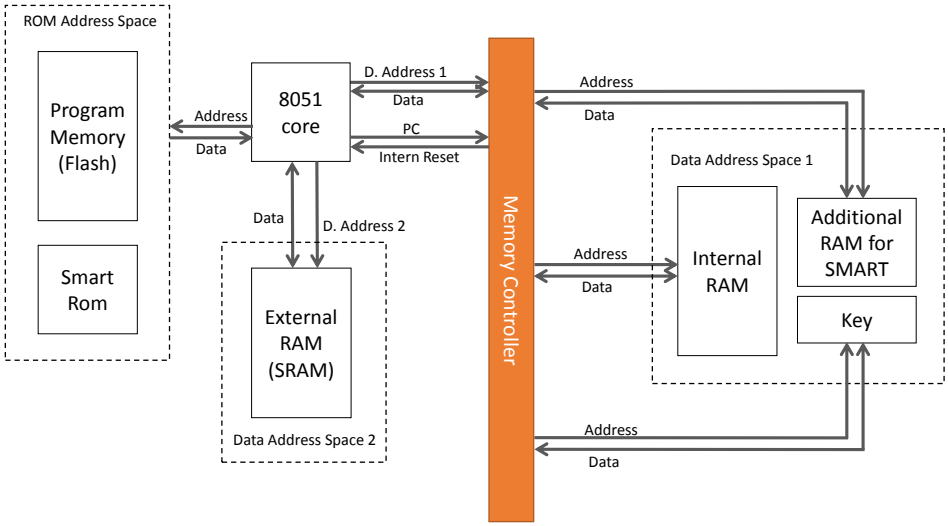

Figure 7.9: 8051 SMART architecture that uses key in exclusive memory address space

Table 7.7: Implementation of the modified SMART architecture in which key is placed to exclusive memory address space

Number of Slice Registers	1.369
Number of Slice LUTs	2.789
Number of RAMB16BWERs	3
Number of RAMB8BWERs	1
Maximum Clock Frequency	57.776 MHz, (min period: 17.509 ns)

The modified SMART architecture, which is given in the Figure 7.9, is implemented on the targeted FPGA. Implementation results are given in the Table 7.7. The area becomes smaller that the standard implementation, which is shown in the Table 7.5, of the SMART and maximum frequency is increased.

7.3 Conclusion

An efficient hardware implementation of SMART scheme for 8051 is designed on the Spartan 6 CGS324 in this chapter. Attestation code uses SHA3 for the keyed hash function. SMART scheme uses PC based memory controller idea that requires an additional hardware as discussed in previous chapters. The top level design of the additional hardware is given in the Figure 7.3 and the total hardware is given in the Figure 7.6. Although a behavioral design approach is used, truth tables and logic gates are presented to visualize the hardware.

This is the first hardware implementation of the SMART scheme that shows the cost of the changes in terms of area and speed. Therefore, there is no other work to compare the study. However, a recent work is published named Sancus by Noorman et. al. [6] that uses PC based memory controller hardware. A very similar hardware implementation is presented by Sancus team [6]. If the Sancus memory controller hardware scheme given in Figure 7.10 is compared with our design shown in the Figure 7.6, our design can be observed as a better implementation. Instead of using many Flip Flops to keep PC, 16 bit for 8051, and hardware blocks to compare previous PC with attestation code region boundaries, only one FF and a very smaller logic is used as explained in the topic 7.2.

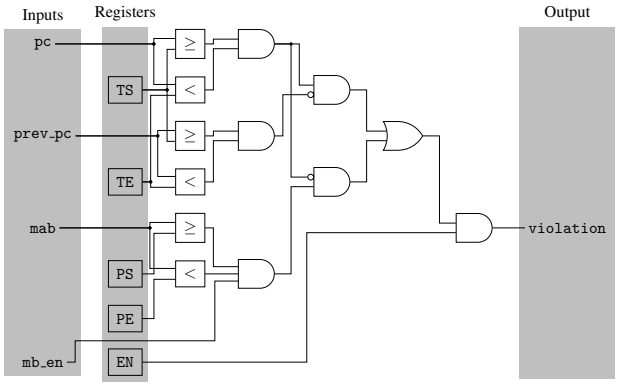

Figure 7.10: Memory control hardware given by Sancus team in the paper [6]

The effects of modifying the 8051 microprocessor hardware to implement SMART scheme can be observed by investigating the Figure 7.11. It can be seen that the burden of additional memory controller hardware is negligible (<%1) while a little cost comes from the additional ROM of attestation code memory. The critical path is increased very little that mainly comes from the secure reset mechanism.

It must be mentioned in here that the Dalton 8051 soft core is implemented using 2 KB additional RAM. That is the reason that the maximum clock frequency of the implementation seems fast (55.972 MHz) when it is compared with its standard (12 MHz). The clock frequency of a standard implementation on XILINX V300PQ240 Virtex FPGA is shown as 11.6 MHz by the designers [103] while the half of the area is loaded with memory. Therefore, burden of SMART scheme to a standard 8051 implementation will be much more smaller.

It can be observed by investigating hardware schematics that additional hardware requires very minor changes and it is very compact since its inputs and outputs can be connected to any processor in the same way.

The speed of the attestation is very important since it will be applied many times as a burden to main duty of the embedded system. Restricting the system implementation in a way that the attestation program can only use stack cannot reserve memory slows execution significantly as discussed in Chapter 6. Therefore most of the compilers of low cost embedded systems are tend to globally address variables as much as they can in default mode. Since the SMART scheme is proposed for very limited devices like 8051 modified versions of architecture are proposed. Modified versions of SMART

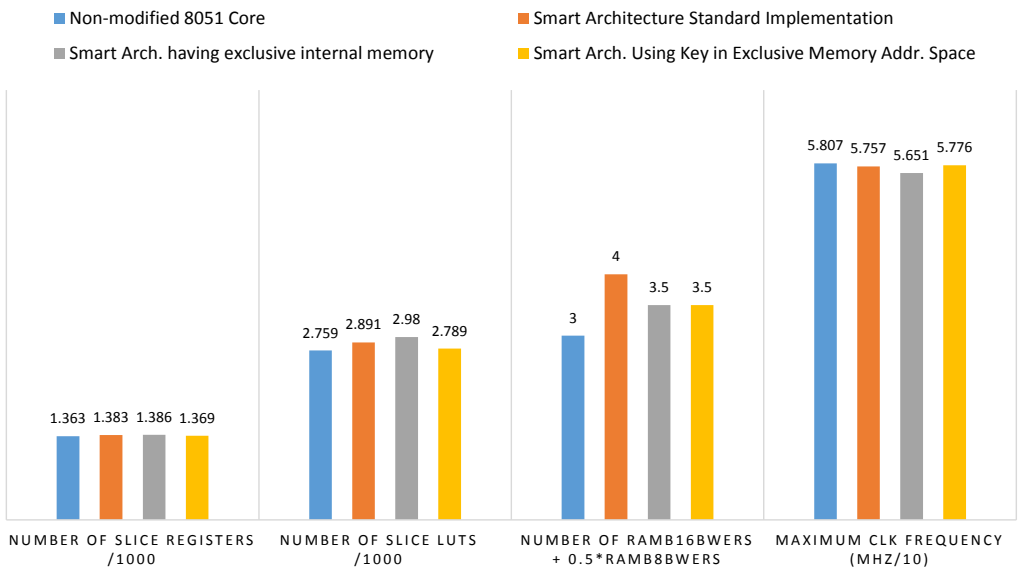

Figure 7.11: Comparison of hardware implementations of SMART scheme and non-modified 8051 core.

scheme is implemented and their cost are shown as compared in the Figure 7.11. An additional RAM, whose accesses are restricted as key address space, is added for the use of attestation code in a secure way. Moreover, the key is also placed to same address space with the additional RAM and it is presented that proposed architecture is better in terms of area and speed. As presented in the Figure 7.11 that proposed approach uses smaller memory while the combinational hardware decreases.

8. CONCLUSIONS AND RECOMMENDATIONS

This thesis deals with the analysis and design of trusted computing platforms on lightweight embedded devices. We primarily focused on properties of implementation of remote attestation on constrained embedded devices. SMART (Secure and Minimal Architecture for Root of Trust) scheme, which is one of the state of the art methods proposed by El Defrawy et al. [7], is examined and implemented using a soft-core 8051 microprocessor.

In Chapter 1, the importance of securely gaining assurance that a local or remote computing platform behaves as expected is discussed and remote attestation is explained as a solution. The lack of secure lightweight remote attestation scheme for constraint embedded systems is discussed as an important problem.

In Chapter 3, the term software vulnerability is explained and some common attacks are explained in order to show how easily a program can be successfully attacked. Then, an overview of defense systems are shown.

In Chapter 4, the related work of defense mechanism for both high end devices and low cost embedded devices are explained. It is shown that although there are a lot of work for powerful computing devices like servers, there are only a small amount of work on low cost devices. These techniques are discussed in three classes as software based systems, hardware based systems and hardware-software based systems. It is explained that a lot of successful attacks are realized on software based systems while hardware based systems are too costly for the computing devices considered in the scope of this thesis. Finally, it is shown that the idea of partitioning the programs and network devices as modules and creating isolated processes for them is considered to be very powerful. Recently proposed hardware-software based systems targeting the low cost embedded devices [2, 5, 7] are explained. After investigating their properties, the idea of integrating a PC based memory controller hardware to create isolated modules is observed as very promising since it is effective and lightweight.

In Chapter 5, it is explained that SMART is a very promising approach using the idea of creating isolated process using PC based memory controller in order to produce an architecture that supports dynamic root of trust. An abstract architecture is given and design rules for a secure design are explained.

In Chapter 6, software design of SMART scheme is explained. Hash algorithms are compared and a lightweight version of SHA3 algorithm is chosen for the checksum function of attestation code. Then, implementation trade-offs of SHA3 is presented for 8051 platform using SDCC compiler. Ten different codes are designed using C and ASM programming languages. Compiler properties are optimized using 8051 platform specific properties. Finally, two different SMART attestation codes are designed and considered as efficient after they are compared with literature. Although reserving memory for the attestation code is not allowed in the proposed scheme in order to keep minimality, the performance loss of designing a software that only uses stack for its variables is shown in the Figure 6.7.

In Chapter 7, Dalton core [103] is used for the implementation of 8051 microprocessor on a Xilinx Spartan 6 CGS324 FPGA. The architecture is modified for the need of SMART scheme and the system is modified by obeying design rules. It is observed that the SMART scheme doesn't need too many changes to the existing architecture and the cost mostly comes from the memories that attestation program and key are stored.

In addition, a novel approach for the implementation of SMART scheme is proposed. Not only the proposed architecture requires smaller additional hardware but also it provides the capability of faster attestation algorithms. In the proposed implementation, SMART architecture is modified in a way that attestation code can reserve memory for its use. In order to achieve this, an exclusive additional RAM is deployed into the internal RAM address space as an alternative memory for the use of RC of SMART. As a second optimization, key memory is placed after RC exclusive RAM in a same way and unified memory area of the key and RC exclusive RAM are treated like the key address space of SMART architecture. This decreased the size of additional memory controller hardware. Pure implementation of 8051, standard SMART implementation and proposed architecture are compared, which is given in the

Figure 7.11. It is presented that the proposed architecture fastened attestation program by 12% with a smaller code by 36% for this design.

As a result, a very efficient implementation of SMART scheme is achieved using a lightweight SHA3[r=40, c=169] algorithm for 8051 on a Xilinx Spartan 6 FPGA. The design properties of the scheme are investigated. To the best of our knowledge this is the first implementation of SMART scheme. The SMART scheme is published presenting two implementation examples. Although they talk about their implementations on AVR and MSP430, the overhead of the system is not clearly mentioned while the effort of implementation is discussed. Moreover, very efficient SHA3[r=40, c=160] programs are designed for 8051 using ASM and C programming languages. No other SHA3[200] implementation for 8051 can be found in the literature at the time of this thesis is written. In addition to these, another approach that optimizes both the hardware and software of SMART architecture is proposed and implemented on 8051.

REFERENCES

[1] **Erlingsson, Ú.**, **Younan, Y. and Piessens, F.**, 2010. Low-level software security by example, Handbook of Information and Communication Security, Springer, pp.633–658.

[2] **Strackx, R.**, **Piessens, F. and Preneel, B.**, 2010. Efficient isolation of trusted subsystems in embedded systems, Security and Privacy in Communication Networks, Springer, pp.344–361.

[3] Memory Types, `http://www.keil.com/support/man/docs/c51/c51_le_memtypes.htm`, last checked on 01.05.2013.

[4] Memory Models, `http://www.keil.com/support/man/docs/c51/c51_le_memmodels.htm`, last checked on 01.05.2013.

[5] **Schulz, S.**, **Sadeghi, A.R. and Wachsmann, C.**, 2011. Short paper: lightweight remote attestation using physical functions, Proceedings of the fourth ACM conference on Wireless network security, ACM, pp.109–114.

[6] **Noorman, J.**, **Agten, P.**, **Daniels, W.**, **Strackx, R.**, **Van Herrewege, A.**, **Huygens, C.**, **Preneel, B.**, **Verbauwhede, I. and Piessens, F.**, 2013. Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base, Proceedings of the 22nd USENIX conference on Security symposium. USENIX Association.

[7] **El Defrawy, K.**, **Francillon, A.**, **Perito, D. and Tsudik, G.**, 2012. Smart: Secure and minimal architecture for (establishing a dynamic) root of trust, Proceedings of the Network & Distributed System Security Symposium (NDSS), San Diego, CA.

[8] Implementations of hash functions in Atmel AVR devices, `http://perso.uclouvain.be/fstandae/source_codes/hash_atmel/`, last checked on 01.12.2013.

[9] **Bertoni, G.**, **Daemen, J.**, **Peeters, M.**, **Assche, G.V. and Keer, R.V.**, 2012, KECCAK implementation overview, `http://keccak.noekeon.org/`.

[10] **Strackx, R.**, **Noorman, J.**, **Verbauwhede, I.**, **Preneel, B. and Piessens, F.**, 2013. Protected Software Module Architectures, ISSE 2013 Securing Electronic Business Processes, Springer, pp.241–251.

[11] **Schellekens, D.**, 2013. Design and Analysis of Trusted Computing Platforms, Ph.D. thesis, PhD thesis, KU Leuven.

[12] **Falliere, N.**, **Murchu, L.O. and Chien, E.**, 2011. W32. stuxnet dossier, *White paper, Symantec Corp., Security Response*.

[13] **Radcliffe, J.**, 2011. Hacking medical devices for fun and insulin: Breaking the human SCADA system, Black Hat Conference presentation slides.

[14] **Checkoway, S.**, **McCoy, D.**, **Kantor, B.**, **Anderson, D.**, **Shacham, H.**, **Savage, S.**, **Koscher, K.**, **Czeskis, A.**, **Roesner, F. and Kohno, T.**, 2011. Comprehensive experimental analyses of automotive attack surfaces, Proceedings of the 20th USENIX conference on Security, USENIX Association, pp.6–6.

[15] **Halperin, D.**, **Heydt-Benjamin, T.S.**, **Ransford, B.**, **Clark, S.S.**, **Defend, B.**, **Morgan, W.**, **Fu, K.**, **Kohno, T. and Maisel, W.H.**, 2008. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses, Security and Privacy, 2008. SP 2008. IEEE Symposium on, IEEE, pp.129–142.

[16] **Francillon, A. and Castelluccia, C.**, 2008. Code injection attacks on harvard-architecture devices, Proceedings of the 15th ACM conference on Computer and communications security, ACM, pp.15–26.

[17] **Giannetsos, T.**, **Dimitriou, T. and Prasad, N.R.**, 2009. Self-propagating worms in wireless sensor networks, Proceedings of the 5th international student workshop on Emerging networking experiments and technologies, ACM, pp.31–32.

[18] Trusted Platform Module Specifications, `http://www.trustedcomputinggroup.org/developers/trusted_platform_module/`, last checked on 01.05.2013.

[19] Intel Trusted Execution Technology (Intel TXT), `http://www.intel.com/technology/security`, last checked on 01.12.2013.

[20] **Seshadri, A.**, **Perrig, A.**, **Van Doorn, L. and Khosla, P.**, 2004. Swatt: Software-based attestation for embedded devices, Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on, IEEE, pp.272–282.

[21] **Seshadri, A.**, **Luk, M.**, **Shi, E.**, **Perrig, A.**, **van Doorn, L. and Khosla, P.**, 2005. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems, ACM SIGOPS Operating Systems Review, volume 39, ACM, pp.1–16.

[22] **Seshadri, A.**, **Luk, M.**, **Perrig, A.**, **van Doorn, L. and Khosla, P.**, 2006. SCUBA: Secure code update by attestation in sensor networks, Proceedings of the 5th ACM workshop on Wireless security, ACM, pp.85–94.

[23] **Seshadri, A.**, **Luk, M. and Perrig, A.**, 2008. SAKE: Software attestation for key establishment in sensor networks, Distributed Computing in Sensor Systems, Springer, pp.372–385.

[24] **Kil, C.**, **Sezer, E.C.**, **Azab, A.M.**, **Ning, P. and Zhang, X.**, 2009. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence, Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on, IEEE, pp.115–124.

[25] **Castelluccia, C.**, **Francillon, A.**, **Perito, D. and Soriente, C.**, 2009. On the difficulty of software-based attestation of embedded devices, Proceedings of the 16th ACM conference on Computer and communications security, ACM, pp.400–409.

[26] **Li, Y.**, **McCune, J.M. and Perrig, A.**, 2011. VIPER: verifying the integrity of PERipherals' firmware, Proceedings of the 18th ACM conference on Computer and communications security, ACM, pp.3–16.

[27] **Ravikanth, P.S.**, 2001. Physical one-way functions, Ph.D. thesis, Massachusetts Institute of Technology.

[28] **Stinson, D.R.**, 2006. Cryptography: theory and practice, CRC press.

[29] **FIPS, P.**, 1995. 180-1. Secure hash standard, *National Institute of Standards and Technology*, **17**.

[30] **FIPS, P.**, 2002. 180-2, *Secure Hash Standard*, **1**.

[31] **FIPS, P.**, 2001. 197, Advanced encryption standard (AES), *National Institute of Standards and Technology*.

[32] **Wang, X.**, **Yin, Y.L. and Yu, H.**, 2005. Finding collisions in the full SHA-1, Advances in Cryptology–CRYPTO 2005, Springer, pp.17–36.

[33] BLAKE Hash Function, `https://131002.net/blake`, last checked on 01.12.2013.

[34] GROESTL Hash Function, `http://www.groestl.info`, last checked on 01.12.2013.

[35] JH Hash Function, `http://www3.ntu.edu.sg/home/wuhj/research/jh/`, last checked on 01.12.2013.

[36] **Bertoni, G.**, **Daemen, J.**, **Peeters, M. and Van Assche, G.**, 2011. The Keccak reference, version 3.0, *NIST SHA3 Submission Document (January 2011)*.

[37] Skein Hash Function, `http://www.skein-hash.info`, last checked on 01.12.2013.

[38] **Krawczyk, H.**, **Canetti, R. and Bellare, M.**, 1997. HMAC: Keyed-hashing for message authentication.

[39] **Krawczyk, H.**, **Canetti, R. and Bellare, M.**, 1997. HMAC: Keyed-hashing for message authentication.

[40] **Agten, P.**, **Nikiforakis, N.**, **Strackx, R.**, **De Groef, W. and Piessens, F.**, 2012. Recent developments in low-level software security, Information Security Theory and Practice. Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems, Springer, pp.1–16.

[41] **Baltopoulos, I.G. and Gordon, A.D.**, 2009. Secure compilation of a multi-tier web language, Proceedings of the 4th international workshop on Types in language design and implementation, ACM, pp.27–38.

[42] **Bailey, M.**, **Cooke, E.**, **Jahanian, F.**, **Watson, D. and Nazario, J.**, 2005. The blaster worm: Then and now, *Security & Privacy, IEEE*, **3(4)**, 26–31.

[43] **One, A.**, 1996. Smashing the stack for fun and profit, *Phrack magazine*, **7(49)**, 365.

[44] **Pierce, B.C.**, 2002. Types and programming languages, The MIT Press.

[45] **Cowan, C.**, **Pu, C.**, **Maier, D.**, **Hinton, H.**, **Walpole, J.**, **Bakke, P.**, **Beattie, S.**, **Grier, A.**, **Wagle, P.**, **Zhang, Q.** *et al.*, 1998. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks, Proceedings of the 7th USENIX Security Symposium, volume 81, pp.346–355.

[46] **Bhatkar, S.**, **DuVarney, D.C. and Sekar, R.**, 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits, Proceedings of the 12th USENIX security symposium, volume120, Washington, DC.

[47] **Barrantes, E.G.**, **Ackley, D.H.**, **Palmer, T.S.**, **Stefanovic, D. and Zovi, D.D.**, 2003. Randomized instruction set emulation to disrupt binary code injection attacks, Proceedings of the 10th ACM conference on Computer and communications security, ACM, pp.281–289.

[48] **Roglia, G.F.**, **Martignoni, L.**, **Paleari, R. and Bruschi, D.**, 2009. Surgically returning to randomized lib (c), Computer Security Applications Conference, 2009. ACSAC'09. Annual, IEEE, pp.60–69.

[49] **Shacham, H.**, **Page, M.**, **Pfaff, B.**, **Goh, E.J.**, **Modadugu, N. and Boneh, D.**, 2004. On the effectiveness of address-space randomization, Proceedings of the 11th ACM conference on Computer and communications security, ACM, pp.298–307.

[50] **Strackx, R.**, **Younan, Y.**, **Philippaerts, P.**, **Piessens, F.**, **Lachmund, S. and Walter, T.**, 2009. Breaking the memory secrecy assumption, Proceedings of the Second European Workshop on System Security, ACM, pp.1–8.

[51] **Bulba, K.**, 2000, Bypassing stackguard and stackshield.

[52] **Wojtczuk, R.**, 1998, Defeating solar designer nonexecutable stack patch.

[53] **Shacham, H.**, 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), Proceedings of the 14th ACM conference on Computer and communications security, ACM, pp.552–561.

[54] **Wahbe, R.**, **Lucco, S.**, **Anderson, T.E. and Graham, S.L.**, 1994. Efficient software-based fault isolation, ACM SIGOPS Operating Systems Review, volume 27, ACM, pp.203–216.

[55] **Levis, P.**, 2012. Experiences from a decade of TinyOS development, Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (Berkeley, CA, USA, 2012), OSDI, volume 12, pp.207–220.

[56] **Han, C.C.**, **Kumar, R.**, **Shea, R.**, **Kohler, E. and Srivastava, M.**, 2005. A dynamic operating system for sensor nodes, Proceedings of the 3rd international conference on Mobile systems, applications, and services, ACM, pp.163–176.

[57] **Dunkels, A.**, **Finne, N.**, **Eriksson, J. and Voigt, T.**, 2006. Run-time dynamic linking for reprogramming wireless sensor networks, Proceedings of the 4th international conference on Embedded networked sensor systems, ACM, pp.15–28.

[58] **Gu, L. and Stankovic, J.A.**, 2006. t-kernel: Providing reliable OS support to wireless sensor networks, Proceedings of the 4th international conference on Embedded networked sensor systems, ACM, pp.1–14.

[59] **Simon, D.**, **Cifuentes, C.**, **Cleal, D.**, **Daniels, J. and White, D.**, 2006. Java™ on the bare metal of wireless sensor devices: the squawk Java virtual machine, Proceedings of the 2nd international conference on Virtual execution environments, ACM, pp.78–88.

[60] **Leontiadis, I.**, **Efstratiou, C.**, **Mascolo, C. and Crowcroft, J.**, 2012. SenShare: transforming sensor networks into multi-application sensing infrastructures, Wireless Sensor Networks, Springer, pp.65–81.

[61] **Aucsmith, D.**, 1996. Tamper resistant software: An implementation, Information Hiding, Springer, pp.317–333.

[62] **Robertson, W.K.**, **Kruegel, C.**, **Mutz, D. and Valeur, F.**, 2003. Run-time Detection of Heap-based Overflows., LISA, volume 3, pp.51–60.

[63] **Collberg, C.S. and Thomborson, C.**, 2002. Watermarking, tamper-proofing, and obfuscation-tools for software protection, *Software Engineering, IEEE Transactions on*, **28(8)**, 735–746.

[64] **Linn, C. and Debray, S.**, 2003. Obfuscation of executable code to improve resistance to static disassembly, Proceedings of the 10th ACM conference on Computer and communications security, ACM, pp.290–299.

[65] **Wroblewski, G.**, 2002. General Method of Program Code Obfuscation (draft), Ph.D. thesis, Citeseer.

[66] **Abadi, M. and Plotkin, G.D.**, 2012. On protection by layout randomization, *ACM Transactions on Information and System Security (TISSEC)*, **15(2)**, 8.

[67] **Cowan, C.**, **Beattie, S.**, **Johansen, J. and Wagle, P.**, 2003. Pointguard TM: protecting pointers from buffer overflow vulnerabilities, Proceedings of the 12th conference on USENIX Security Symposium, volume 12, pp.91–104.

[68] **Bhatkar, S. and Sekar, R.**, 2008. Data space randomization, Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, pp.1–22.

[69] **Chew, M. and Song, D.**, 2002. Mitigating buffer overflows by operating system randomization.

[70] **Billet, O.**, **Gilbert, H. and Ech-Chatbi, C.**, 2005. Cryptanalysis of a white box AES implementation, Selected Areas in Cryptography, Springer, pp.227–240.

[71] **Goubin, L.**, **Masereel, J.M. and Quisquater, M.**, 2007. Cryptanalysis of white box DES implementations, Selected Areas in Cryptography, Springer, pp.278–295.

[72] **Jacob, M.**, **Boneh, D. and Felten, E.**, 2003. Attacking an obfuscated cipher by injecting faults, Digital Rights Management, Springer, pp.16–31.

[73] **Chow, S.**, **Eisen, P.**, **Johnson, H. and Van Oorschot, P.C.**, 2003. A white-box DES implementation for DRM applications, Digital Rights Management, Springer, pp.1–15.

[74] **Michiels, W. and Gorissen, P.**, 2007. Mechanism for software tamper resistance: an application of white-box cryptography, Proceedings of the 2007 ACM workshop on Digital Rights Management, ACM, pp.82–89.

[75] **Ceccato, M.**, **Dalla Preda, M.**, **Nagra, J.**, **Collberg, C. and Tonella, P.**, 2009. Trading-off security and performance in barrier slicing for remote software entrusting, *Automated Software Engineering*, **16(2)**, 235–261.

[76] **Kennell, R. and Jamieson, L.H.**, 2003. Establishing the genuinity of remote computer systems, Proceedings of the 12th USENIX Security Symposium, Washington, DC, USA, pp.295–308.

[77] **Shankar, U.**, **Chew, M. and Tygar, J.**, 2004. Side effects are not sufficient to authenticate software, USENIX Security Symposium, volume 8.

[78] **Seshadri, A.**, **Luk, M.**, **Perrig, A.**, **Doorn, L.v. and Khosla, P.**, 2006. Externally verifiable code execution, *Communications of the ACM*, **49(9)**, 45–49.

[79] **Zhang, X. and Gupta, R.**, 2003. Hiding program slices for software security, Code Generation and Optimization, 2003. CGO 2003. International Symposium on, IEEE, pp.325–336.

[80] **Corbató, F.J. and Vyssotsky, V.A.**, 1965. Introduction and overview of the Multics system, Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I, ACM, pp.185–196.

[81] **Kostiainen, K.**, **Ekberg, J.E.**, **Asokan, N. and Rantala, A.**, 2009. On-board credentials with open provisioning, Proceedings of the 4th International Symposium on Information, Computer, and Communications Security, ACM, pp.104–115.

[82] **Tygar, J. and Yee, B.**, 1995. Secure coprocessors in electronic commerce applications, Proceedings 1995 USENIX Electronic Commerce Workshop.

[83] **Smith, S.W. and Weingart, S.**, 1999. Building a high-performance, programmable secure coprocessor, *Computer Networks*, **31(8)**, 831–860.

[84] **PUB, N.F.**, 2001. 140-2: Security requirements for cryptographic modules, *Information Technology Laboratory, National Institute of Standards and Technology*.

[85] `http://www.trustedcomputinggroup.org`, last checked on 01.01.2014.

[86] **Mitchell, C.**, **Mitchell, C. and Mitchell, C.**, 2005. Trusted computing, Springer.

[87] **McCune, J.M.**, **Parno, B.J.**, **Perrig, A.**, **Reiter, M.K. and Isozaki, H.**, 2008. Flicker: An execution infrastructure for TCB minimization, ACM SIGOPS Operating Systems Review, volume 42, ACM, pp.315–328.

[88] **McCune, J.M.**, **Li, Y.**, **Qu, N.**, **Zhou, Z.**, **Datta, A.**, **Gligor, V. and Perrig, A.**, 2010. TrustVisor: Efficient TCB reduction and attestation, Security and Privacy (SP), 2010 IEEE Symposium on, IEEE, pp.143–158.

[89] **Azab, A.M.**, **Ning, P. and Zhang, X.**, 2011. Sice: a hardware-level strongly isolated computing environment for x86 multi-core platforms, Proceedings of the 18th ACM conference on Computer and communications security, ACM, pp.375–388.

[90] **Li, Y.**, **McCune, J.M. and Perrig, A.**, 2010. SBAP: Software-based attestation for peripherals, Trust and Trustworthy Computing, Springer, pp.16–29.

[91] **Sadeghi, A.R.**, **Visconti, I. and Wachsmann, C.**, 2010. PUF-enhanced RFID security and privacy, Workshop on Secure Component and System Identification (SECSI).

[92] **Oztiirk, E.**, **Hammouri, G. and Sunar, B.**, 2008. Towards robust low cost authentication for pervasive devices, Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on, IEEE, pp.170–178.

[93] **Rührmair, U.**, 2009. SIMPL Systems: On a Public Key Variant of Physical Unclonable Functions., *IACR Cryptology ePrint Archive*, **2009**, 255.

[94] **Avonds, N.**, **Strackx, R.**, **Agten, P. and Piessens, F.**, 2013. Salus: Non-hierarchical Memory Access Rights to Enforce the Principle of Least Privilege, Security and Privacy in Communication Networks, Springer, pp.252–269.

[95] **Francillon, A.**, **Instutute, E.**, **Nguyen, Q.**, **Rasmussen, K.B. and Tsudik, G.** Systematic Treatment of Remote Attestation.

[96] **Balasch, J.**, **Ege, B.**, **Eisenbarth, T.**, **Gérard, B.**, **Gong, Z.**, **Güneysu, T.**, **Heyse, S.**, **Kerckhof, S.**, **Koeune, F.**, **Plos, T.** *et al.*, 2013. Compact implementation and performance evaluation of hash functions in attiny devices, Smart Card Research and Advanced Applications, Springer, pp.158–172.

[97] **Bertoni, G.**, **Daemen, J.**, **Peeters, M. and Assche, G.V.**, 2007, Sponge functions, Ecrypt Hash Workshop 2007.

[98] **Bertoni, G.**, **Daemen, J.**, **Peeters, M. and Van Assche, G.**, 2008. On the indifferentiability of the sponge construction, Advances in Cryptology–EUROCRYPT 2008, Springer, pp.181–197.

[99] Reference and optimized code in C v3.2, `http://keccak.noekeon.org/files.html`, last checked on 01.05.2013.

[100] SDCC Compiler User Guide v.3.2.1., `http://sdcc.sourceforge.net/doc/sdccman.pdf`, last checked on 01.05.2013.

[101] **Gouicem, M.**, 2010. Comparison of seven SHA-3 candidates software implementations on smart cards., *IACR Cryptology ePrint Archive*, **2010**, 531.

[102] **Guido, B.**, **Joan, D.**, **Michaël, P.**, **Gilles, V. and Ronny, V.**, 2011. K implementation overview.

[103] Dalton 8051, `http://www.cs.ucr.edu/~dalton/8051/`, last checked on 01.05.2013.

CURRICULUM VITAE

Name Surname: Mehmet Akif Özkan

Place and Date of Birth: Ankara, 24.07.1989

Adress: Sisli/Istanbul

E-Mail: akif.ozkan@itu.edu.tr

B.Sc.: Istanbul Technical University

M.Sc.: Istanbul Technical University

Professional Experience and Rewards:

- **Organizing committee member** *"The 3rd Turkish Symposium on Embedded Systems and Its Applications Symposium" , GOMSIS2012*.
- **3rd Best Graduation Project** *7. Graduation Project Design Competition organized by The Chamber of Turkish Electirical Engineers*, Istanbul, 2011.
- **Best Student Presentation Award**, *GOMSIS 2012 Symposium*, Istanbul, 2012

List of Publications and Patents:

- **Ozkan M.A.**, Ors B. & Saldamli G. (2011), "Secure Voice Communication via GSM Network", 7th International Conference on Electrical and Electronics Engineering (ELECO11), Bursa, Turkey, December 14, 2011.
- Tuncay S., **Ozkan M.A.** & Ors S. (2012), "HW/SW codesign of a data modem for GSM Voice Channel", 3rd Turkish Symposium on Embedded Systems and Its Applications Symposium (GOMSIS12), Istanbul, Turkey, November 29, 2011. (National)

PUBLICATIONS/PRESENTATIONS ON THE THESIS

- **M. Akif Ozkan**, A. Van Herrewege, I. Verbauwhede, S. Berna Ors, (2013) "Implementation of a Lightweight Trusted Platform Module", *34th WIC Symposium on Information Theory in the Benelux*, Leuven, Belgium, May 30, 2013.