Universidade Estadual de Campinas
Instituto de Computação

Henrique de Medeiros Kawakami

A Framework for Hardware Security Evaluation.

Um Framework para a Avaliação de Segurança de Hardware.

CAMPINAS
2015

Henrique de Medeiros Kawakami

A Framework for Hardware Security Evaluation.

Um Framework para a Avaliação de Segurança de Hardware.

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Ricardo Dahab

Este exemplar corresponde à versão final da Dissertação defendida por Henrique de Medeiros Kawakami e orientada pelo Prof. Dr. Ricardo Dahab.

CAMPINAS
2015

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

K179f

Kawakami, Henrique de Medeiros, 1979-
A framework for hardware security validation / Henrique de Medeiros Kawakami. – Campinas, SP : [s.n.], 2015.

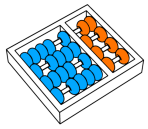
Orientador: Ricardo Dahab.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Computadores - Medidas de segurança. 2. Rede de computadores - Medidas de segurança. 3. Arquitetura de computador - Medidas de segurança. I. Dahab, Ricardo,1957-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Um framework para a avaliação de segurança de hardware
Palavras-chave em inglês:
Computer system security
Computer networks - Security measures
Computer architecture - Security measures
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Luiz Eduardo Buzato
Edson Borin
Hao-Chi Wong
Data de defesa: 21-08-2015
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação



Henrique de Medeiros Kawakami

A Framework for Hardware Security Evaluation.

Um Framework para a Avaliação de Segurança de Hardware.

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (presidente)
  Instituto de Computação (IC) - UNICAMP
- Prof. Dr. Edson Borin
  Instituto de Computação (IC) - UNICAMP
- Prof. Dr. Hao-chi Wong
  Intel

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 21 de agosto de 2015

Acknowledgements

First and foremost, I would like to thank my advisor, Prof. Dr. Ricardo Dahab. His guidance and support greatly contributed to this work, and taught me how good research should be done.

I also thank David Ott and Hao-chi Wong for the several fruitful discussions that we have had since the beginning. I greatly appreciate their insights and the knowledge that they shared throughout the various phases of this research.

I am also thankful to Roberto Gallo, who actively contributed to this research and to the published paper in the ARES conference.

Lastly, I am very grateful for the financial support of Intel Corporation by the Intel Strategic Research Alliance (ISRA) program. Our participation in the ISRA conferences were of the most importance to interact with other research groups and other researchers from Intel.

Resumo

O *hardware* de sistemas computacionais possui uma função crítica na segurança de sistemas operacionais e aplicativos. Além de prover funcionalidades-padrão, tal como o nível de privilégio de execução, o *hardware* também pode oferecer suporte a criptografia, boot seguro, execução segura, e outros.
Com o fim de garantir que essas funcionalidades de segurança irão operar corretamente quando juntas dentro de um sistema, e de que o sistema é seguro como um todo, é necessário avaliar a segurança da arquitetura de todo sistema, durante o ciclo de desenvolvimento do *hardware*.
Neste trabalho, iniciamos pela pesquisa dos diferentes tipos existentes de vulnerabilidades de *hardware*, e propomos uma taxonomia para classificá-los. Nossa taxonomia é capaz de classificar as vulnerabilidades de acordo com o ponto no qual elas foram inseridas, dentro do ciclo de desenvolvimento. Ela também é capaz de separar as vulnerabilidades de *hardware* daquelas de *software* que apenas se aproveitam de funcionalidades-padrão do *hardware*.

Focando em um tipo específico de vulnerabilidade - aquelas relacionadas à arquitetura - apresentamos um método para a avaliação de sistemas de *hardware* utilizando a metodologia de Assurance Cases. Essa metodologia tem sido usada com sucesso para a análise de segurança física e, tanto quanto saibamos, não há notícias de seu uso para a análise de segurança de *hardware*. Utilizando esse método, pudemos identificar corretamente as vulnerabilidades de sistemas reais.
Por fim, apresentamos uma prova de conceito de uma ferramenta para guiar e automatizar parte do processo de análise que foi proposto. A partir de uma descrição padronizada de uma arquitetura de *hardware*, a ferramenta aplica uma série de regras de um sistema especialista e gera um relatório de *Assurance Case* com as possíveis vulnerabilidades do sistema-alvo. Aplicamos a ferramenta aos sistemas estudados e pudemos identificar com sucesso as vulnerabilidades conhecidas, assim como outras possíveis vulnerabilidades.

Abstract

The hardware of computer systems plays a critical role in the security of operating systems and applications. Besides providing standard features such as execution privilege levels, it may also offer support for encryption, secure execution, secure boot, and others. In order to guarantee that these security features work correctly when inside a system, and that the system is secure as a whole, it is necessary to evaluate the security of the architecture during the hardware development life-cycle.
In this work, we start by exploring the different types of existing hardware vulnerabilities and propose a taxonomy for classifying them. Our taxonomy is able to classify vulnerabilities according to when they were created during the development life-cycle, as well as separating real hardware vulnerabilities from software vulnerabilities that leverage standard hardware features.
Focusing on a specific type of vulnerability - the architecture-related ones, we present a method for evaluating hardware systems using the Assurance Case methodology. This methodology has been used successfully for safety analysis, and to our best knowledge there are no reports of its use for hardware security analysis. Using this method, we were able to correctly identify the vulnerabilities of real-world systems.
Lastly, we present the proof-of-concept of a tool for guiding and automating part of the proposed analysis methodology. Starting from a standardized hardware architecture description, the tool applies a set of expert system rules, and generates an Assurance Case report that contains the possible security vulnerabilities of a system. We were able to apply the tool to the studied systems, and correctly identify their known vulnerabilities, as well as other possible vulnerabilities.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Compared to software, computer hardware has received less attention from both security researchers and practitioners. One of the possible reasons for this is that most hardware products are designed and manufactured by a small number of companies, who hold the responsibility for both checking and correcting security issues. Thus, the broader research community would not have as much information about hardware development, compared to what it has about software.

Another possible reason is that, although hardware complexity and performance has been steadily increasing to accommodate more demanding software applications, hardware architectures might still be less complex than the software that they run, and that performance issues are still the primary concern during hardware development.

However, with the ongoing demand for new features and integration, hardware architectures will continue to become increasingly complex. As an example, the average number of system-on-chip (SoC) IP cores is increasing at a rate of 20% per year, and was estimated to be around 100 per chip in 2014.

Besides increasing the necessary effort to verify that a system is functional, there has been an increasing pressure to verify that the interaction between its subsystems will not undermine the security of the system as a whole. And although the methods and tools for system functionality verification are relatively well established and ever developing, the same cannot be said about security verification.

Like other types of products, computing hardware products go through a development life-cycle that includes architecture/design and implementation phases [2]. In each phase, security vulnerabilities may arise. Fixing security vulnerabilities during the architecture/design phase is usually much less costly than doing so once its implementation has started. Likewise, fixing vulnerabilities while the product is still in pre-silicon validation is much less costly than when it is post-silicon[1]. It therefore makes economical sense to evaluate the product from the security standpoint during the architecture review or pre-silicon validation [3].

Currently, most of the security evaluation of hardware architectures is manual and ad-hoc. Evaluators try to form a mental model of the system by going through (natural language-based) product architecture/design specifications, and/or by interacting with

[1] In many cases, solving a hardware issue after production may be even impossible

product architects and designers. The goal is to extract an abstract model of the system that contains only the security-relevant aspects of the system. They then try to identify security issues in this abstract model, based on their security expertise. The output of such a process is usually a list of vulnerabilities found, or an informal statement that no issues have been found. The main problems with this approach are:

- the extraction of the abstract model is a very time-consuming process that is repeated every time an evaluator analyses a new system;
- the requirements on security evaluators are very demanding since their knowledge of security and hardware must be comprehensive and deep;
- the resulting abstract model is not documented, which prevents it from being reused, shared, or reviewed;
- because the evaluation process is largely informal, it is not straightforward to determine how complete an evaluation is, and consequently, how secure or vulnerable the design is.

Another important aspect is that the very definition of hardware vulnerability is not as straightforward as the definition of "classical" software vulnerabilities, such as buffer overflow. From the computer user point of view, a hardware vulnerability may arise from the improper use of hardware features, such as the PXE Ethernet boot. For software developers, a hardware vulnerability can exist when malicious code makes non-standard use of hardware features (this in fact is a common understanding in CVE bug descriptions). Finally, low-level hardware designers may consider that such vulnerability only exists when there is a privilege access control bug in a microprocessor's RTL code, for example.

1.1 Objectives

This work's primary objective is to develop a framework for hardware security evaluation during the architecture review or pre-silicon validation phase. From this objective we derived a methodology with four key results that were necessary to achieve our objective:

1. the definition of hardware vulnerability, and the development of a taxonomy for the classification of hardware vulnerabilities;
2. the development of an abstract framework for modeling security aspects of hardware architectures;
3. the development of a language for expressing security requirements and algorithms for evaluating them;
4. the development of a proof-of-concept tool to showcase and evaluate the framework.

1.2 State-of-the-art

The literature on general hardware security is not as abundant as that on software security. Furthermore, the literature about hardware security validation during architecture review and pre-silicon phases is even more scarce.

Potlapally's model checking

One of very few and possibly the most relevant work on hardware security during the architecture review and pre-silicon phases is a paper by Potlapally [2], in which the author uses a formal approach for hardware validation employing a high-level formal model of the hardware (DESIGN) developed in the TLA+ language. The attacker model (ADVERSARY) was also formally described, and the system was evaluated against a given security goal (SECURITY PROPERTY):

$$((DESIGN \vee ADVERSARY) => SECURITY PROPERTY)$$

If the model checker finds a counterexample for the logical condition, it will output it and show that the system does not hold the security property. If it does not find a counterexample, the property can be considered a theorem, and the system is secure against the given adversary model.

The system was tested against a known vulnerability in one of the early versions of the Intel Core CPU. The vulnerability allowed an attacker to override a series of protection mechanisms and successfully modify the contents of the SMM (System Management Mode) memory. This attack will be latter described in Chapter 2, and was first presented by Loïc Duflot [4]. The result from this test was that the system was able to successfully detect the vulnerability from both a TLA+ model of the CPU subsystem and from the CPU RTL code.

Potlapally's approach is fairly similar to ours, since it relies on an evaluation of an architecture model versus an attacker model. The main difference is that it uses formal models. As advantages, we can consider that the checker was able to identify a relatively complex attack (SMM cache), and that it can potentially be fully automated after the formal models are coded. As disadvantages, the formal models are not easy to maintain and require intricate manual work that is prone to errors. It is not clear whether this system was able to find the other vulnerabilities that also affected the SMM subsystem.

Gallo's Fortuna framework

Fortuna[5] is a framework for the security analysis of computing systems. It is aimed at helping developers during the conception and implementation phases of secure systems, but can also serve as a post-design validation tool. It is an agent-oriented methodology, in the sense that the system modeling and analysis is built according to assumptions on how an agent (in this case, an attacker) will act upon the characteristics of the system.

The framework consists of two phases. First, the system is modelled as a graph, according to some specific rules and properties. As an example, the cryptographic token

architecture depicted in Figure 1.1 is modelled as the graph shown in Figures 1.2 and 1.3.

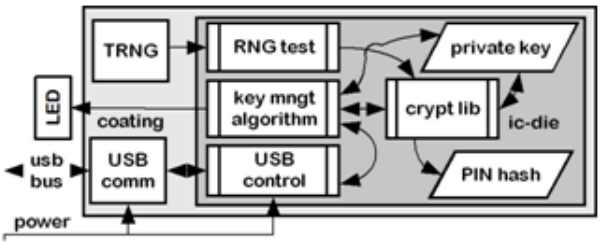


Figure 1.1: Sample cryptographic token architecture.

In the representation of Figure 1.1, the external box represents the physically protected (i.e. tamper resistant or evident) space of the device. The internal box contains the token's microcontroller and other security-sensitive components.

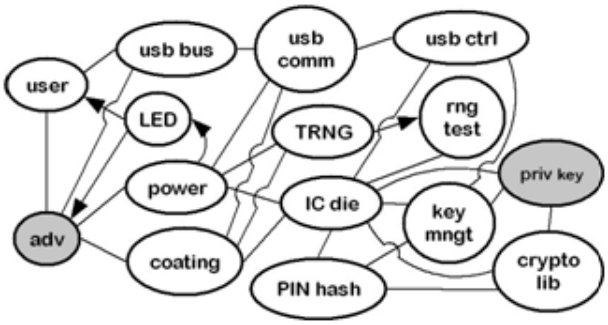


Figure 1.2: Fortuna-modelled graph of the sample token.

The graph in Figure 1.2 is the result of the modeling of the original architecture according to the Fortuna methodology. It was considered that the private key was the target of the attacker (agent).

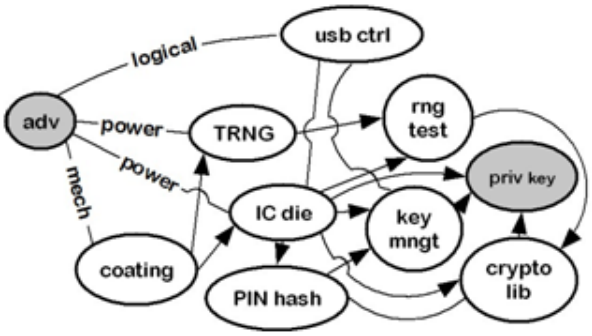


Figure 1.3: Fortuna-modelled graph of the sample token, containing the protection relationship between various nodes.

The second graph, represented in Figure 1.3 was derived from the first, but also considering the protection relationship between the various elements. As an example, we note that the private key security depends on the elements "crypto lib", "key management", and "IC die".

These graphs allow a qualitative analysis of the whole architecture. It is possible to visualize, for instance, if the security-critical elements are protected by multiple layers of security mechanisms, or if they are subject to a single point of failure.

The second phase of Fortuna consists of adding the probability of attack for each node of the system model, resulting in a weighted graph. These probabilities may represent a measurement of how likely it is to find a new security bug in a software component, or how likely it is to successfully penetrate through a tamper resistant hardware module. After the probabilities are added, the most likely (or easiest) path of attack can be calculated, as well as how likely or easy it would be to successfully attack the system.

Although Fortuna is aimed at a similar problem, it is not suitable for use in our work due the following reasons:

- Since Fortuna relies on probabilities that are estimated from low-level design statistics (number of lines of code, etc), we would not be able to have a model only from public datasheet information;
- it is difficult to model only the target subsystem and isolate the rest of the system. To model a secure memory region, for instance, it would be necessary to model all CPU subsystems, requiring a possibly unacceptable amount of workload.

Secure Tropos

Secure Tropos [6] is an extension of Tropos [7], which is an agent-oriented software development methodology. The Secure Tropos methodology could be applied to hardware security evaluation in a way similar to the Fortuna framework, but would have the same modeling complexity problem.

Jasper's Security Path Verification App

Jasper's Security Path Verification App [8] is a commercially available tool for formal security analysis of RTL code. The tool is based on a form of taint analysis, but there is no public information about its internal workings. This tool is able to detect conditions that would result in the leakage of protected registers or cryptographic keys. The user can specify the memory regions of the protected data, and the regions where this data can and cannot be transferred to.

The analysis of the current methods for the evaluation of hardware security indicates that there is in fact a need for new methods of evaluation. The existing tools either rely on extensive work of security analysts, or do not provide the depth of analysis that is needed for a complete evaluation.

Literature on the Assurance Case methodology for hardware security

This work relies on the use of the Assurance Case methodology for hardware security evaluation. Extensive bibliographic review showed no previous report of usage of this methodology for this specific use. There are, though, some related work regarding the use

of tools to automate part of the development of Assurance Cases, which is also a topic of this work.

Rushby [9] presented a tool to provide assistance to formal verification of the argument structure of assurance cases using the Prototype Verification System (PVS) language [10].

AdvoCATE [11] is a toolset developed by Denney et al. to support the automated construction and assessment of safety cases. Although it does not automatically generate assurance cases, it is able to merge auto-generated safety case fragments from other tools such as the AUTOCERT [12] formal verification tool.

In 2013, Denney et al. [13] also described a method for the formal description of safety case patterns. The formal description was then used to enable the automatic instantiation of the pattern in a safety case.

1.3 Summary of contributions

We studied various types of hardware vulnerabilities and created a taxonomy for their classification. Our taxonomy has some novel aspects: it allows the mapping of the type of vulnerability to the design phase of the hardware, and also allows a clear distinction between real hardware vulnerabilities and firmware or driver bugs.

After developing the taxonomy, we focused on a specific type of vulnerability - the architecture-related ones - and developed a new method for modeling and verifying the involved hardware architectures. The method for verifying the hardware architectures is based on the Assurance Case methodology, and is the subject of a paper [14] accepted in the ARES 2015 Conference (International Conference on Availability, Reliability and Security).

This resulting method was used to create a proof-of-concept tool to analyze real-world systems. We were able to detect most of the hardware vulnerabilities that were studied, including all of the architecture-related ones. We also detected some other possible vulnerabilities, but did not further investigate whether they existed in real-world systems.

The proof-of-concept tool was summarized in a paper that is going to be submitted to the HOST symposium (IEEE International Symposium on Hardware-Oriented Security and Trust).

1.4 Document organization

Chapter 2 describes real-world hardware attacks and vulnerabilities, and allows a deeper understanding of the involved problems and challenges. Chapter 3 presents the tools and methodologies that are used in our work. Chapter 4 describes a hardware vulnerability taxonomy that we created and that allows the definition and characterization of hardware problems. We also propose a new method for modeling hardware architectures. Chapter 5 presents a method for analyzing the security of hardware architectures, along with a method for automating part of the analysis process. Chapter 6 concludes the work, and gives a summary of the results and lessons learned during our research.

Chapter 2

Hardware attacks

In this chapter we present some of the hardware attacks that were studied during this work. They are important for understanding the hardware vulnerability problem, along with various types of vulnerabilities. For every attack, there is at least one associated vulnerability, which is classified into the vulnerability taxonomy that is presented in Chapter 4.

2.1 System Management Mode (SMM) Attacks

System Management Mode (SMM) is a x86 processor mode that is used to execute firmware for the control and management of critical features of modern motherboards, such as fan speed control and battery management. These critical, real-time functions, cannot rely on the responsiveness of the OS, since a malfunction can result in physical hazards to the system and even the user. If SMM did not exist, motherboard vendors would probably have to add a dedicated microcontroller to control these critical real-time functions. And in fact, SMM mode makes the CPU resemble a microcontroller, in the sense that SMM code runs in 16-bit mode, with unrestricted access to the system memory and I/O.

In our case study we assume an architecture consisting of the Intel Core 2 CPU and a Q35 express chipset. This chipset is composed of a northbridge and a southbridge, as shown in Figure 2.1. More information on SMM can be found on the IA-32 and Intel 64 Software Developer's Manual [1].

SMRAM and SMBASE

The System Management RAM (SMRAM) is a special region in memory that stores code and data of the SMM firmware. The SMM region typically resides in one of three standard address regions for SMM code, but it can also occupy other addresses ranges. The base address of the SMRAM is stored in the SMBASE register inside the CPU (thus, only the CPU knows where SMM code actually resides during runtime).

The chipset's northbridge provides a mechanism to block access to the SMRAM memory region after it is loaded by the BIOS. This mechanism is controlled by 3 bits of a register inside the Northbridge, as shown in Figure 2.2. It is the BIOS' responsibility to

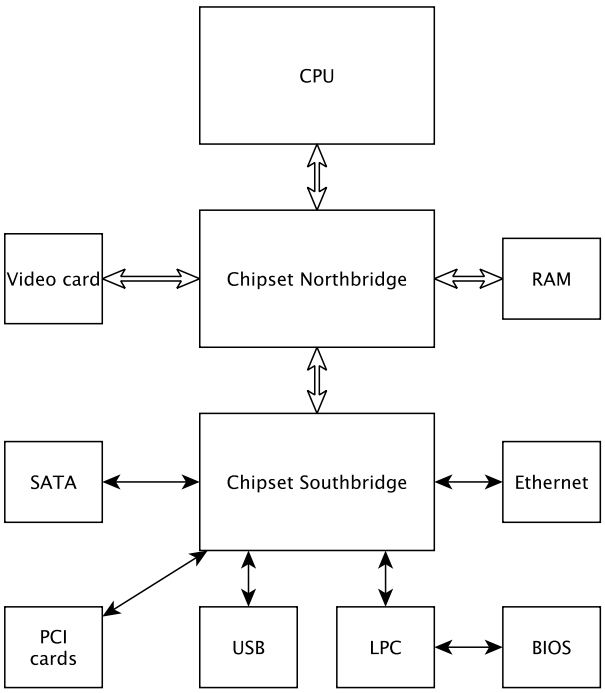


Figure 2.1: Typical northbridge/southbridge chipset architecture.

properly configure these registers before the OS starts execution. It is important to note that this mechanism only blocks access to the 3 standard address regions for SMM code.

2.1.1 First attack - SMM remap attack

Attack Explanation

This attack was presented by Loïc Duflot [4], and consists of a way to bypass hardware security mechanisms in order to write to the SMRAM protected memory region. It is a very good case of how the interaction between different subsystems can result in a security vulnerability.

The attack consists of using the memory remapping mechanism to bypass the SMRAM protection that is provided by the chipset. The memory remap feature was created in order to recover physical memory regions that would otherwise be wasted, due to conflicts with system reserved address (e.g. I/O devices, video card memory). The memory remapping is done entirely within the chipset, according to its register settings. Once these registers are programmed by the CPU, the address translation is completely transparent to the processor, and works independently of other address translations mechanisms (e.g. MMU).

The remapping address translation table is configured by registers inside the northbridge. The problem is that the northbridge also checks and blocks accesses to specific address regions, such as the SMRAM address range. Thus, if the protected address check is done before the memory remapping occurs, an unauthorized access may take place, since the access may be remapped from an allowed address to a protected address. This

SMRAM—System Management RAM Control Register (D0:F0)

Address Offset: 9Dh
Default Value: 02h
Attribute: RO, R/W
Size: 8 bits

The SMRAMC register controls how accesses to Compatible and Extended SMRAM spaces are treated. The Open, Close, and Lock bits function only when G_SMRAME bit is set to a 1. Also, the OPEN bit must be reset before the Lock bit is set.

Bits	Default, Access	Description
7		Reserved
6	0b R/W	**SMM Space Open (D_OPEN).** When D_OPEN=1 and D_LCK=0, the SMM space DRAM is made visible, even when SMM decode is not active. This is intended to help BIOS initialize SMM space. Software should ensure that D_OPEN=1 and D_CLS=1 are not set at the same time.
5	0b R/W	**SMM Space Closed (D_CLS).** When D_CLS = 1, SMM space DRAM is not accessible to data references, even if SMM decode is active. Code references may still access SMM space DRAM. This allows SMM software to reference through SMM space to update the display even when SMM is mapped over the VGA range. Software should ensure that D_OPEN=1 and D_CLS=1 are not set at the same time. **NOTE:** D_CLS only applies to compatible SMM space.
4	0b R/W	**SMM Space Locked (D_LCK).** When D_LCK is set to 1, D_OPEN is reset to 0 and D_LCK, D_OPEN, G_SMRARE, H_SMRAME, TSEG_SZ and T_EN become read only. D_LCK can be set to 1 via a normal configuration space write but can only be cleared by a Full Reset. The combination of D_LCK and D_OPEN provide convenience with security. The BIOS can use the D_OPEN function to initialize SMM space and then use D_LCK to "lock down" SMM space in the future so that no application software (or BIOS itself) can violate the integrity of SMM space, even if the program has knowledge of the D_OPEN function.
3	0b R/W/L	**Global SMRAM Enable (G_SMRARE).** When this bit is 1, Compatible SMRAM functions are enabled, providing 128 KB of DRAM accessible at the A0000h address while in SMM (ADS# with SMM decode). To enable the Extended SMRAM function, this bit must be set to 1. Refer to the section on SMM for more details. **NOTE:** Once D_LCK is set, this bit becomes read only.
2:0	010b RO	**Compatible SMM Space Base Segment (C_BASE_SEG).** This field indicates the location of SMM space. SMM DRAM is not remapped. It is simply made visible if the conditions are right to access SMM space; otherwise, the access is forwarded to the hub interface. Since the MCH supports only the SMM space between A0000h and BFFFFh, this field is hardwired to 010.

Figure 2.2: Details of the SMRAM protection register inside the Intel Q35 Chipset (excerpt from the IA-32 and Intel 64 Software Developer's Manual [1].

is the case with the Q35 chipset, as reported by the attack described in Duflot et al. [15].

Figure 2.3 shows how the chipset control access to SMRAM memory. The 3 standard SMRAM memory regions have their access blocked, unless the CPU asserts the SMM_MODE signal, indicating that it has entered SMM mode.

If an attacker is able to gain kernel mode privileges, he can use the memory remap feature to remap a non-blocked memory region to the SMRAM memory region, as depicted in Figure 2.4. Then, it would be possible to access the SMRAM contents, eventually gaining access to a more privileged mode of execution (i.e. SMM mode).

Associated vulnerability

This attack explores the fact that the SMRAM protection mechanism did not take into consideration other features of the memory control subsystem. We can consider that the underlying vulnerability of this attack is that the memory remap address translation was done after the SMRAM memory address check, allowing the SMRAM protection

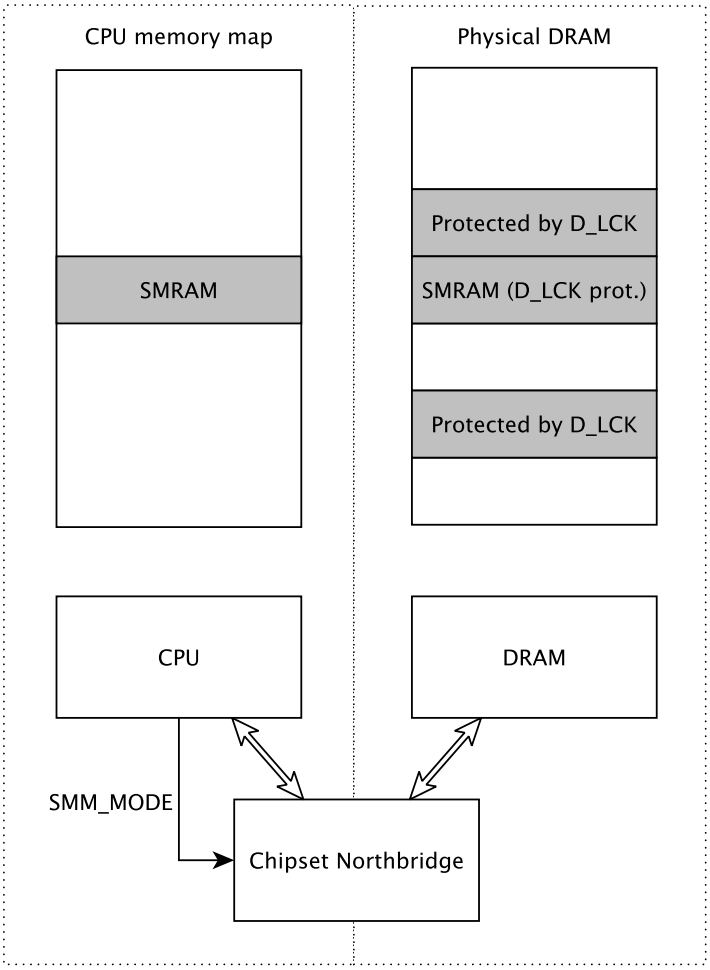


Figure 2.3: SMRAM memory protection mechanism.

mechanism to be bypassed.

Since the vulnerability arises from the interaction between these two subsystems, is classified as (1.a.ii) *Incompatibility between the security mechanisms of subsystems.*

2.1.2 Second attack - SMM attack using cache memory

This attack was also presented by Duflot [4], and is another method for exploiting the SMM subsystem.

Attack Explanation

The attack consists of writing to SMRAM code when it is inside the CPU cache memory, effectively bypassing the SMRAM protection of the chipset. Although there are various details that need to be taken into consideration in order to have a successful attack, the attack is based in setting the SMRAM memory range as a write-back cacheable in the MTRR registers.

The MTRR registers control how different memory regions are cached inside the CPU. The write-back cache policy, for instance, consists of processing memory-write operations

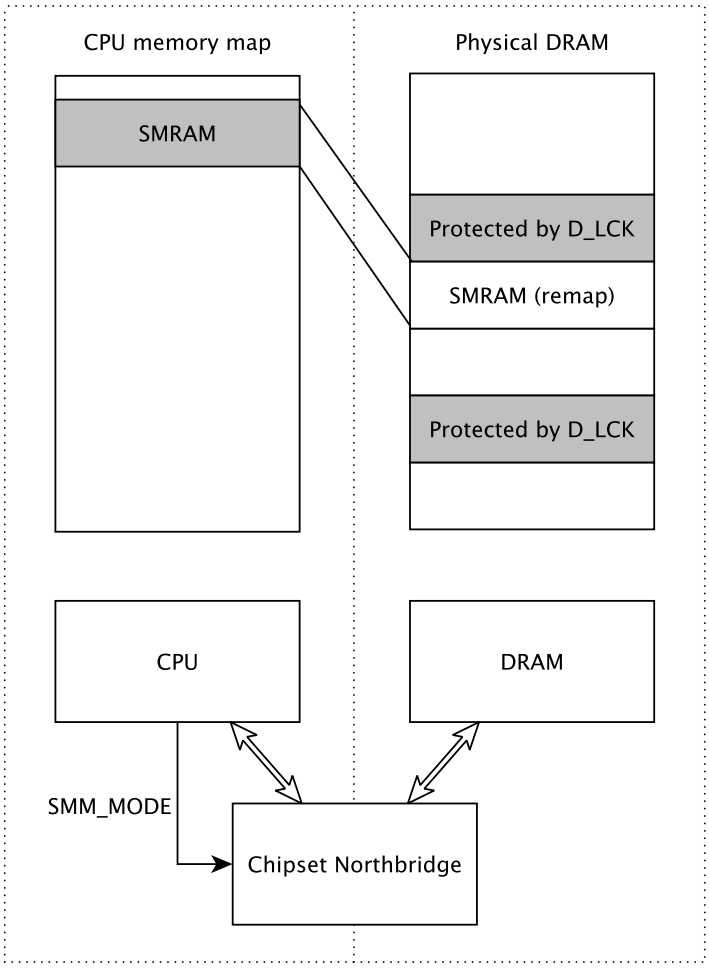


Figure 2.4: SMRAM memory protection mechanism bypassed by memory remapping.

first inside the cache memory, and only writing them to external memory when the cache line is flushed.

If the SMRAM memory is cached with the write-back policy, all write operations to its data will occur entirely within the CPU, and the chipset does not "know" that the SMRAM code is being modified. The SMRAM code will eventually be flushed from cache and written back to external memory, and the chipset will be able to block the write operation. The problem is that the attacker could already have used the SMM privilege to modify the SMRAM base address (SMBASE) to a another memory region, and filled it with malicious code.

This attack would allow a program executing in ring-0 to bypass the hardware security mechanisms that protect the SMRAM (System Management RAM) against writing. Since the code executing in SMRAM has a higher privilege than ring-0 code, it can be considered a privilege escalation attack.

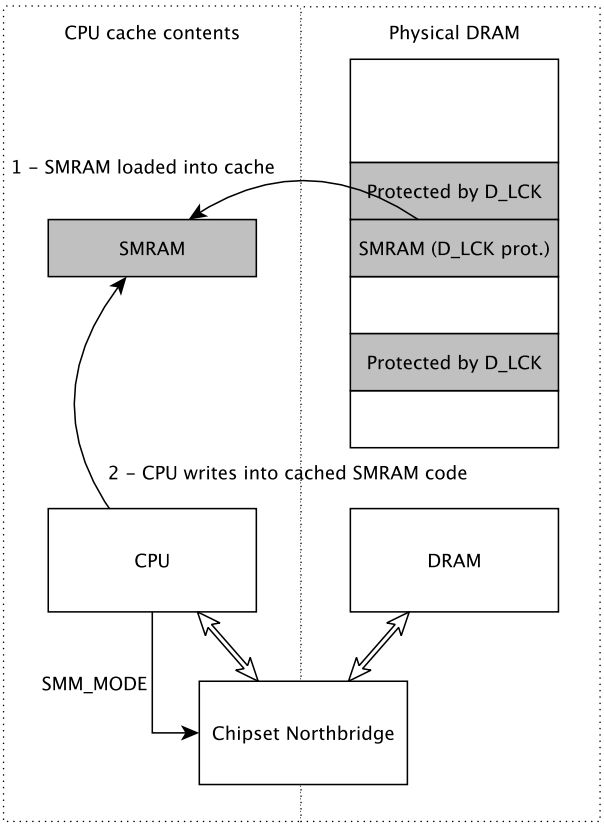


Figure 2.5: Illustration of the SMRAM cache attack.

Associated vulnerability

We can consider that the attack is the result of incoherence between CPU and chipset security mechanisms. While the chipset properly implemented its SMRAM write protection mechanisms, it was not considered the fact that the CPU could write to the SMRAM contents without generating write operations to the chipset.

In order to fix this vulnerability, more recent CPUs have specific registers to control the SMRAM cache policy (the System Management Range Registers - SMRR). If set properly, then the SMRAM region cannot be set to have a Write Back cache policy, and this attack cannot be carried out.

Since the vulnerability arises from the interaction between these two subsystems, it is classified as (1.a.ii) *Incompatibility between the security mechanisms of subsystems.*

2.2 MSI and SIPI attack

Attack Explanation (first part - MSI)

This attack was presented by ITL [16], and consists of a clever way of using the PCIe bus protocol to bypass the Intel VT-d security mechanisms that were present in older systems.

The attack works by using a driver in a virtualized driver domain (a VM that has

direct access to various hardware devices) to execute specific PCIe bus transactions that induce the generation of interrupt in the hypervisor, as depicted in Figure 2.6.

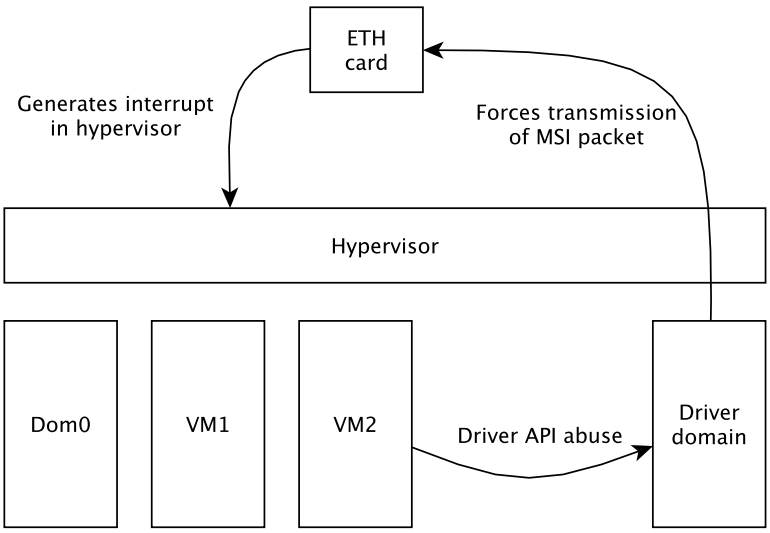


Figure 2.6: Hypervisor exploit overview.

Introduction to Intel VT-d

Intel VT-d [17] (Virtualization Technology for Directed I/O) is a set of hardware features that were created to increase the performance and security in virtualized environments. It allows guest systems to have direct access to PCI devices, without overhead from the hypervisor. It provides the following features:

- I/O device assignment - Allows the hypervisor to assign I/O devices to guest machines.
- DMA remapping - Supports address remapping for device DMA data transfers.
- Interrupt remapping - Provides routing and isolation of device interrupts to specific guest machines.
- Reliability features - Reports and records errors that may otherwise corrupt memory or break guest machine isolation.

Using a standard data packet to generate an interrupt

The mechanism that allows the generation of the PCIe interrupt is called Message Signaled Interrupt. Instead of using specific physical pins to signal interrupts, modern PCI peripherals generate interrupts by sending special packets of data in the standard PCIe communication channel. In order to generate an interrupt, the device has to send a packet in the format given in Figure 2.7. This packet is the same packet that would be

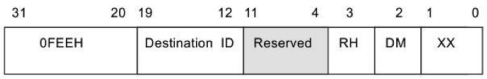


Figure 2.7: MSI packet format.

generated if the device were to write to an address starting with 0xFEEh.

If a malicious program can induce a device to write to this address range (i.e. starting with 0xFEEh), an interrupt will be generated. As most devices have some kind of DMA controller, it can be relatively easy to program such a controller to write to this address and thus generate an interrupt from outside of the driver domain VM. This is what this attack does, in this specific case using a network card, and using a scatter-gather DMA controller from the card chipset.

As these interrupts are generated by the PCI cards, they cannot be blocked by the standard VT-d security mechanisms. These interrupts can be carefully crafted by the guest machine in order to force the host to execute malicious code with hypervisor privileges, as shown in the second part of the attack.

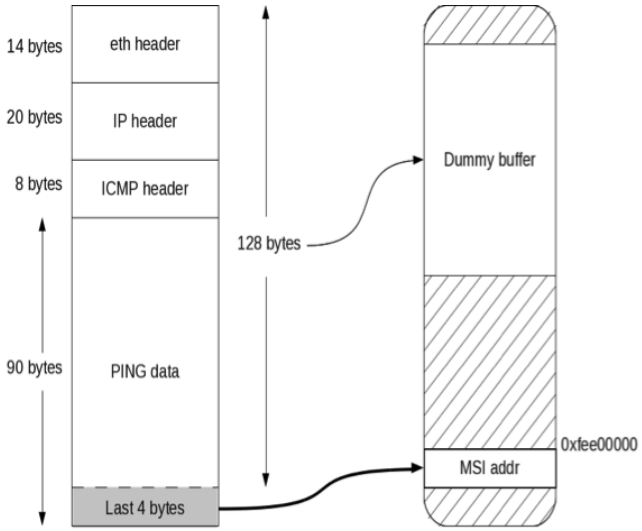


Figure 2.8: Example of a data transfer that causes an MSI, when scatter-gather DMA is properly configured to enable the attack.

Attack Explanation (second part - SIPI)

Generating an interrupt on the hypervisor domain is the first part of the attack. The second part of the attack deals with how to gain privilege benefits from this interrupt. The original paper suggests three ways to exploit these interrupts, and we will focus on the more interesting way from the hardware perspective, which is through the generation of a SIPI interrupt.

A SIPI is a Start-up Inter Processor Interrupt, which is an interrupt that is used during the initialization process of a multi-core system. When this interrupt occurs, the affected CPU starts to execute code at an address specified by the interrupt, in the range

(0x000000 to 0xFF0000). The architecture documentation manual [18] states that the SIPI interrupts can only be generated from a CPU, by writing to registers in the local APIC (LAPIC). It should not be possible to generate this interrupt from PCI cards or other bus devices. Figure 2.9 contains the LAPIC register description from the original architecture manual [18]. The SIPI interrupt is generated when the Delivery Mode field is set to 110b.

Analyzing the MSI packet format that generates interrupts from the PCI bus, the attacker noticed that it had a resemblance with the LAPIC register. Specifically, the delivery field contained the same number of bits, and the allowed values matched those from the LAPIC register, with the exception that the SIPI value (110b) was marked as "reserved". The attacker then tried to generate an interrupt from a PCI device, using the MSI packet with the 110b value in the Delivery Mode field, and a SIPI interrupt was generated.

Combining the two steps of this attack, it would be possible for an attacker to execute any code in the 0-1MB region of memory, with hypervisor privileges.

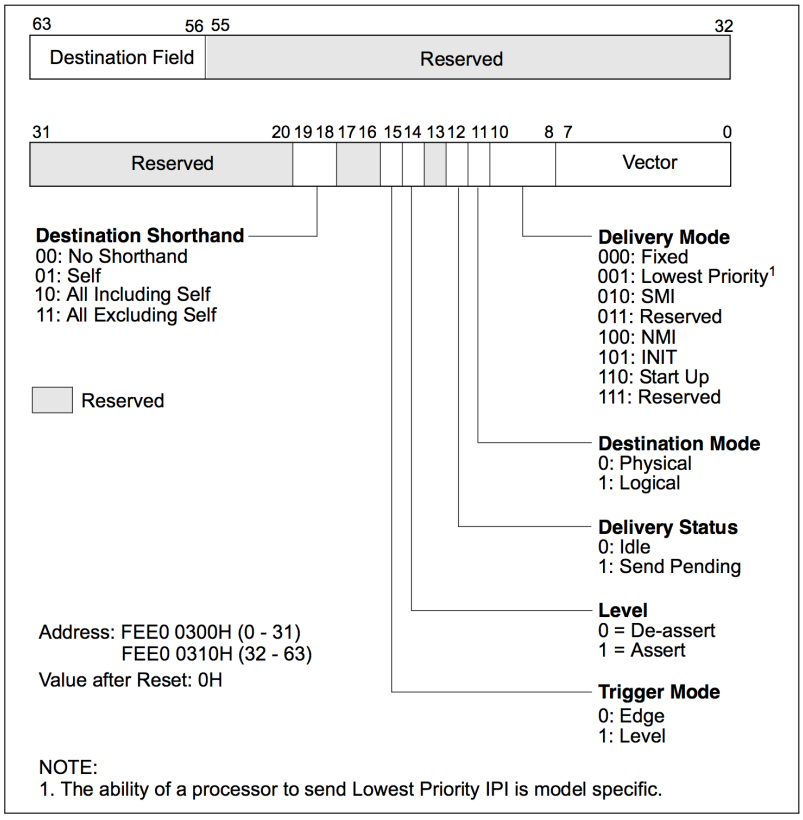


Figure 2.9: Interrupt Command Register (ICR) from Intel SDM.

Countermeasures

There is a hardware mechanism called Interrupt Remapping that can protect the system against this attack. This mechanism is part of the VT-d specification, and has to be supported by the CPU and operating system. This feature was not implemented in the early VT-d systems, but it implemented in the current Intel CPUs. Thus, the attack

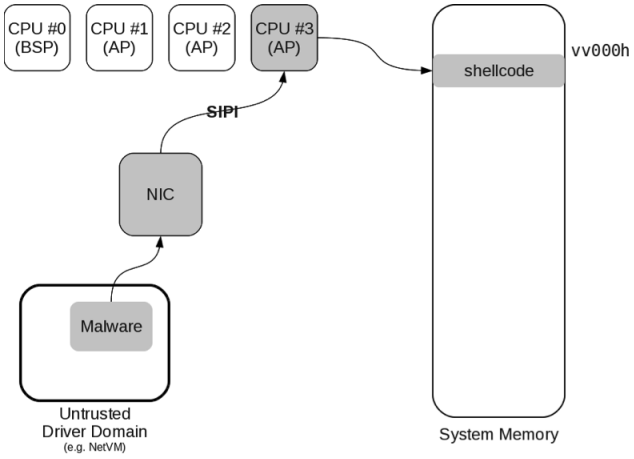


Figure 2.10: Summary of the MSI and SIPI attack.

should not work on the newer architectures, as long as the operating system correctly configures the Interrupt Remapping feature.

Vulnerability classification

This attack explores two distinct vulnerabilities: the interrupt generation from a standard PCIe packet that is processed as a MSI packet, and the generation of a SIPI interrupt from a PCI device.

The first vulnerability is classified as a (1.a.i) *Architectural flaw of a single subsystem* of the involved PCI peripheral, since it should not allow the issuing of data packets that have the same syntax as MSI packets.

The second vulnerability is classified as a (1.b) *Architecture implementation* of the MSI packet processor, since it was not implemented as specified in the architecture manual [18].

2.3 GART privilege escalation

This attack was presented by Loïc Duflot in 2007 [19]. It consists of using the AGP Graphics Aperture mechanism to bypass security mechanisms, allowing processes to access any physical memory location. In the same work, Duflot also described a similar attack that used the USB Host Controller to allow privilege escalation. Although this later attack is based in a different hardware mechanism, both attacks operate on the same principle, which is the exploit of an insecure interaction between two hardware subsystems.

Attack Explanation

The GART mechanism is a feature to allow memory locations to be remapped according to a translation table. It was useful to remap memory from an AGP video card into the physical RAM memory ranges, increasing transfer speeds. It is implemented inside the chipset northbridge, and is programmed by standard PCI configuration registers.

The attack consists of a process with access to the GART configuration registers to remap the video buffer memory to a specific privileged memory region, such as kernel memory. Then, it would be able to write to this memory using standard video API calls. Since standard processes do not have access to video memory, this attack would have to exploit video card drivers or video API calls.

If the IOMMU system is not properly configured, this attack could also be used for a virtual machine to have access to any memory location on its host platform.

Vulnerability classification

This vulnerability is classified as (1.b):*Incompatibility between the security mechanisms of subsystems*, because it results from a unplanned interaction between the GART and MMU memory protection mechanisms.

2.4 Pentium FOOF bug

This bug was published in a 1998 edition of the Dr. Dobbs Journal [20]. FOOF is the shorthand of the hexadecimal encoding of an instruction that caused the halt of the affected Pentium processors. Recovery was only possible after the assertion of the processor reset signal pin.

Since the instruction could be executed by any process with user privileges, this bug could be effectively used as means for a denial-of-service attack.

Attack Explanation

F00F is the shorthand of f0 0f c7 c8, which is the hexadecimal encoding of the lock CMPXCHG8B EAX instruction. The CMPXCHG8B instruction is used to compare the value in the EDX and EAX registers with an 8-byte value at some memory location. In this example a 4-byte register is used as the destination operand, which is not big enough to store the 8-byte result. The original Dr. Dobbs article provides a detailed explanation of the bug:

> "When any x86 processor from the 80186 and beyond encounters an invalid instruction, the processor is supposed to generate an invalid opcode exception. In Intel vernacular, the undefined opcode exception is known as a "#UD." This handler usually signals an error condition and terminates the errant program. When this mechanism works, the errant program can't harm the computer system. Should this mechanism fail, however, the errant program can bring down the entire computer. If the computer is a network server or ISP, then the errant program can bring down the entire network. That's what can, in fact, happen when the Pentium encounters the "F00F" bug, which maps to a LOCK CMPXCHG8B EAX instruction. CMPXCHG8B compares 64-bit memory contents with the contents in EDX and EAX. One of the operands

must be memory, and the other (implied) operand is EDX:EAX. It is possible to construct an instruction encoding that doesn't map to a memory operand. Since the non-memory form of this instruction is invalid, a compiler or assembler will not generate this code. Instead, assembly-language programmers must construct it by hand.

Such an illegal encoding should generate the requisite #UD. As you'd expect, a CMPXCHG8B EAX instruction generates a #UD. However, when this illegal encoding is prepended with a LOCK prefix, the processor fails to work correctly. Using the LOCK prefix on this form of CMPXCHG8B is illegal in and of itself. LOCK prefixes are only allowed on memory-based read-modify-write instructions. Hence a LOCK prefix on the register-based CMPXCHG8B EAX instruction should also generate an invalid opcode exception. Instead, the Pentium locks up and freezes the entire computer when it encounters this instruction. This bug is especially nasty, because any user can construct a program with this instruction, and upload it to a network computer, or incorporate it within an ActiveX applet. Once the program is run on the network, the network server crashes. The only possible recovery comes by hitting the big red switch. Suppose you download an ActiveX applet that contains this code. As soon as the code executes, your computer freezes up.

"When the processor encounters the instruction F0 0F C7 C8 (or anything from F0 0F C7 C8..CF), the F00F bug occurs. The processor recognizes that an invalid opcode has occurred and tries to dispatch the #UD handler. Because of the LOCK prefix, the processor is confused. When the processor issues the bus reads to get the #UD handler vector address, the processor erroneously asserts the LOCK# signal. The LOCK# signal can only be asserted for read-modify-write instructions that modify memory. When the bus is locked, a locked memory read must be followed by a locked memory write, lest unpredictable results may occur. But in this case, the LOCK# signal remains asserted for the two consecutive memory reads required to retrieve the #UD vector address. The processor never issues any intervening locked write, and then hangs itself."

Vulnerability classification

From this explanation, we can assume that the problem is within the CPU itself, and does not depend on other systems or software. Thus, it is classified as an *Architecture Implementation type of vulnerability.*

Chapter 3

Tools

In this chapter we present the tools that are used in our methodology.

3.1 Assurance Cases

The Assurance Case framework originated from a generalization of the Safety Case framework [21], which was created to ensure that mission-critical systems (in aerospace, nuclear power, defense, and other industries) were adequately designed. Assurance cases can relate to any aspect of a system, such as reliability, availability, and security. An assurance case about the security of a given system can be referred to as a *security assurance case*, or just as a *security case*.

According to the original definition [22], a Safety Case is:

> "A documented body of evidence that provides a convincing and valid argument that a system is adequately safe for a given application in a given environment."

According to Suleiman et al. [23], the major benefits of Safety Cases are:

- Making the implicit explicit:
  - easier to review the arguments, question the evidence and challenge the adequacy of the reasoning presented;
  - creating greater transparency in the overall assurance process;
- aiding communication among stakeholders;
- integrating and assessing evidence sources;
- aiding safety management and governance.

Similarly, an Assurance Case is a documented body of evidence that a system holds a given claim. Quoting from NATO's Allied Engineering Publication #67 (AEP-67) [24]:

> "System assurance is the justified confidence that the system functions as intended and is free of exploitable vulnerabilities, either intentionally or unintentionally designed or inserted as part of the system at any time during the life cycle. This ideal of no exploitable vulnerabilities is usually unachievable in practice, so *programmes*[1] must perform risk management to reduce the probability and impact of vulnerabilities to acceptable levels.
>
> The *Assurance Case* is the enabling mechanism to show that the system will meet its prioritized requirements[...] *It is a means to identify all the assurance claims, and from those claims (formally) trace through to their supporting arguments, and from those arguments to the supporting evidence.*"

As the name suggests, Assurance Cases share commonalities with legal cases. They both consist of a group of structured, objective evidence that supports a given claim. As with legal cases, the evidence can be compiled into a text document. However, Assurance Cases can also be graphically represented using standardized notations such as GSN (Goal Structuring Notation) [25] or ASCAD [22].

An example of an Assurance Case structured in GSN (Goal Structuring Notation) is shown in Figure 3.1. The goal at the top of the figure is called the topmost goal. The reasoning that is used to unfold the topmost goal into sub-goals is called a strategy. In the GSN notation, the strategy can be annotated between a claim and its sub-claim. There is also the concept of justification (for a strategy) that can be also annotated besides the strategy.

For each sub-goal, it is necessary to provide and describe a solution, which is by its turn supported by evidence. Typically, evidence consists of reported measurements, tests or statistical data. Thus, a finished Assurance Case should have supporting evidence for all aspects related to the topmost goal.

A graphical Assurance Case may also have a textual counterpart, or may be developed entirely in textual form. Figure 3.2 contains a simple textual Assurance Case relating to the security of a subsystem (SMM) that exists in several Intel Core CPUs.

3.1.1 Assurance Case definition and use

Formally, an Assurance Case can be defined as a tree in which:

- the root node contains the topmost (i.e. higher level) claim that has to be assured;
- the inner nodes contain sub-claims that need to be true, in order to assure that the topmost claim is true;
- each claim (parent) has to be supported by either other claims or evidence (childs);
- the Assurance Case is said to be complete (and the system, secure) if all the tree leaves are composed of evidence that support their corresponding parent nodes (claims).

[1] A set of related measures or activities with a particular long-term aim: e.g. the British nuclear power programme

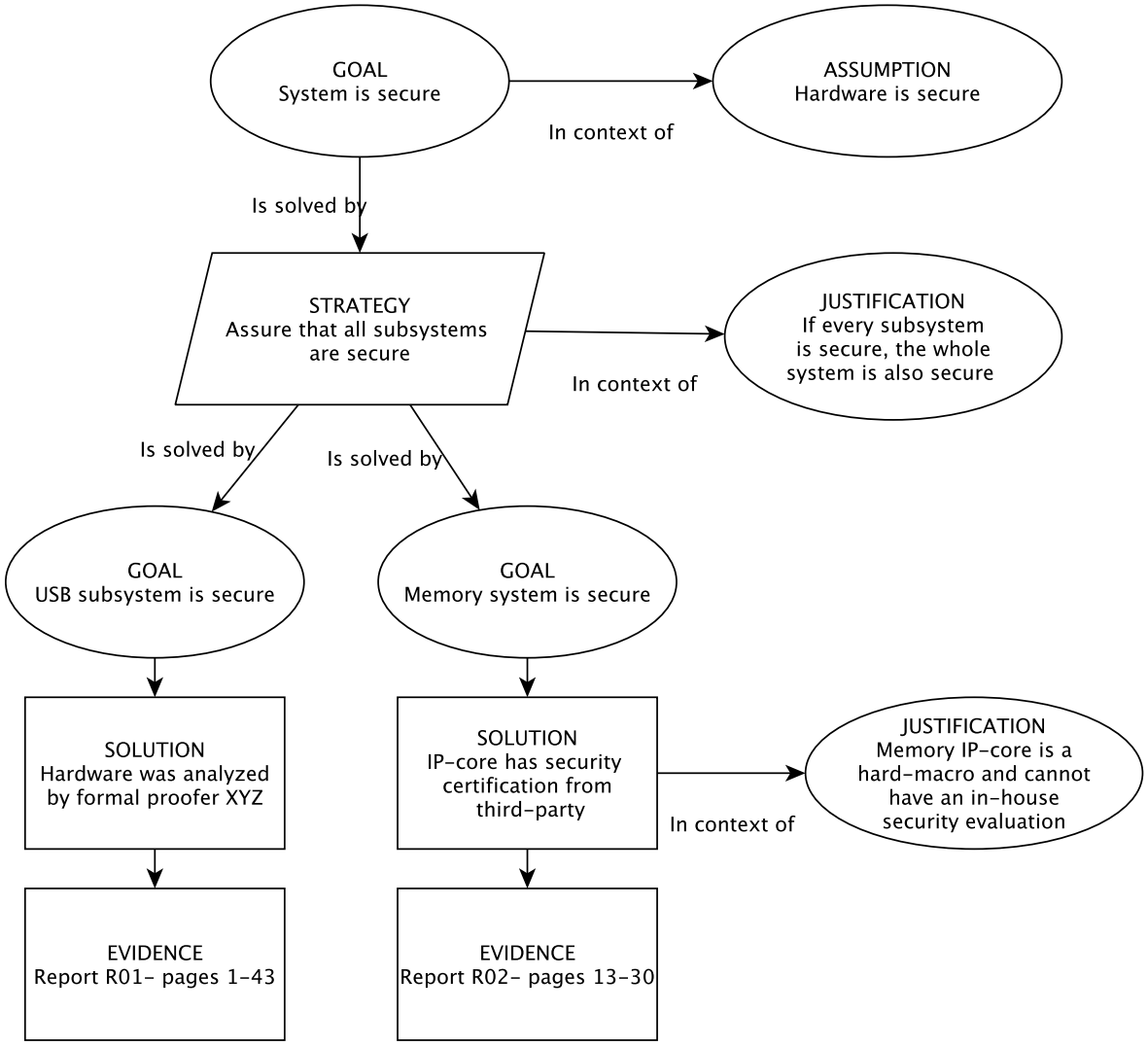


Figure 3.1: Sample Assurance Case represented in the Goal Structuring Notation (GSN).

An Assurance Case starts with the statement of a topmost claim. This claim represents the ultimate security or safety goal that is expected or desired from the system. The topmost claim is supported by a series of sub-claims, which are in turn supported by evidence. Other than claims and evidence, an Assurance Case may also contain Strategies, Assumptions, and Justifications. We present the details of each of these elements, as well as the best practices for creating them.

Claims or goals

The topmost claim should provide a clear statement about what is expected from the system. It should not be a complex or a conditional statement. An acceptable topmost claim for a hardware security assurance case could be, for example, "The system's hardware is acceptably secure". If there are some aspects of the system that cannot be assured according to the topmost claim, they will eventually appear as unsupported sub-claims (i.e. sub-claims that do not have supporting evidence).

Claim: SMM code cannot be modified after boot.

Context: By modified we mean that the SMM code cannot be modified and then executed with SMM privileges, except if the modification is made by the original SMM code.

Strategy: Analyze system architecture and include hardware components that can store and modify SMM code.

Subclaim 1: SMM code residing in DRAM cannot be modified if the CPU is not in SMM mode.

Subclaim 1.1: After boot process, chipset only allows access to SMRAM memory region if CPU is in SMM mode.

Argument 1.1.1: During boot process, chipset registers are correctly set to secure SMRAM.
Evidence 1.1.1: Intel IA-32 Architecture Manual: SMRAM D_LCK bit is set to 1 by BIOS.

Argument 1.1.2: Both D_OPEN and D_CLOSE must not be set to 1 at the same time.
Evidence 1.1.2: Intel IA-32 Architecture Manual: BIOS correctly sets D_OPEN and D_CLOSE before setting D_LCK.

Figure 3.2: Example of Assurance Case in the textual form. This example relates to the security of a subsystem (SMM) that exists in various Intel Core CPUs.

The sub-claims that follow the topmost claim should be declared as statements that can be classified as true or false, such as "The interrupt system is secure". They have to support the topmost claim, and will be naturally more detailed and technical. When there is no evidence to directly support a given sub-claim, more sub-claims should be created down the hierarchy.

Evidence

The evidence is any type of information or reasoning that can support a given sub-claim. As an example, they can consist of reference to test reports, documentation, and source code. There should be not doubt that the evidence is sufficient to support the sub-claim.

Strategy

The strategy is an optional field that describes the reasoning used to create sub-claims from a given claim. By making the strategy explicit, it can be easier for a reader to understand and to eventually find faults in the used reasoning.

Assumption

This is an optional note that describes high level assumptions that are necessary for the correctness of the assurance case, but that are not under the control of the design team. An assumption can be related to any other elements, such as strategies or evidence.

Justification

The justification element is optional, and can provide a detailed explanation of why a given strategy was chosen or why a claim was created.

3.2 Expert systems

As the name suggests, an expert system is designed to reproduce the diagnostics and decisions of a human expert. It is typically implemented as an inference engine that processes information from a knowledge base. The inference engine is responsible for making all the reasoning, using the data and applying the rules programmed in the knowledge base. In this way, expert systems can be programmed just by coding the knowledge base according to information provided by a human expert. The inference engine can then process this knowledge base and output the decisions or diagnostics about the input, as shown in Figure 3.3.

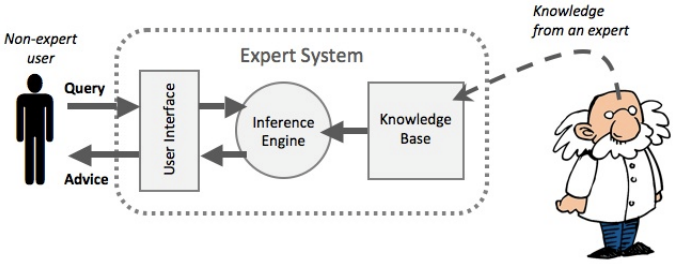


Figure 3.3: Expert systems general architecture.

Expert systems were used in our work as a tool to automate part of the Assurance Case (AC) generation process. The automation process also contributed to solving part of the following weaknesses in the AC methodology:

- Management of argument patterns: argument patterns are templates that can be used to provide guidance and speed up the process of building Assurance Cases. As the number of different patterns is increased, the effort necessary to find which pattern should be used and when it should be used also increases. An expert system can automatically find the patterns that are applicable to a given assurance case.
- Statement of patterns and goals: the statements in a standard AC should be clear and well defined, but they do not need to necessarily follow a specific syntax. Although this can have a positive impact allowing more flexibility, it also makes it difficult to represent the AC as a structured set of knowledge. On the other hand, if we represent the statements in a way that is close to the way that data and rules are represented in an expert system, the resulting AC will have a structured representation that could be further machine-processed.

3.2.1 CLIPS

CLIPS [26] is a an acronym of "C Language Integrated Production System", and consists of a software tool designed to build expert systems. It is an open source project with ports to various platforms, and it is believed that it is one of the most used expert system tools. The software consists of an interpreter of the CLIPS object oriented language interpreter, together with an IDE for coding and debugging.

The CLIPS language is roughly equivalent to the Prolog language, in the sense that both are used in the context of AI, Expert Systems and Language Processing. For this specific application in AC, the CLIPS language seems to be a better fit, specially because of its way of describing facts and rules, as well as its native support for object orientation. An example of a CLIPS program is shown in Figure 3.4.

```
; CLIPS fact definition
(assert (animal-is duck))


; rule example: if the "animal-is duck" fact is present,
; "quack" is printed
(defrule duck
(animal-is duck)
=>
(printout t "quack" crlf))
```

Figure 3.4: A CLIPS language program.

3.3 yEd

yEd is a general-purpose software for drawing diagrams. In this work we use yEd to draw Assurance Case patterns and hardware architectures. It was chosen because it is a free software that is capable of saving diagrams in the GraphML format.

The GraphML format is an XML-based format to save graphs. It is useful to transfer graph information between different software platforms. Figure 3.5 contains a sample GraphML XML description, and Figure 3.6 shows its corresponding graph in the graphical form.

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="n0"/>
    <node id="n1"/>
    <edge id="e1" source="n0" target="n1"/>
  </graph>
</graphml>
```

Figure 3.5: Sample graph described in the GraphML format.

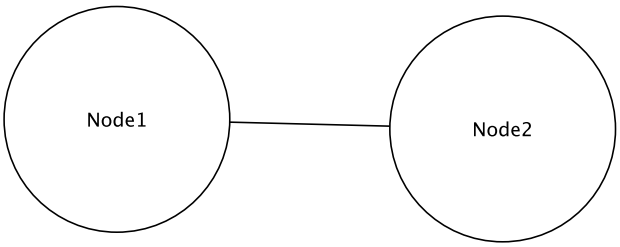


Figure 3.6: Corresponding graph from sample GraphML description.

Chapter 4

Hardware modeling framework

In this chapter, we start by presenting a taxonomy for classifying hardware vulnerabilities. Then, we select a specific type of vulnerability to focus our work on. Finally, we present a method for modeling hardware architectures that will enable us to analyze systems against this type of vulnerability.

4.1 Hardware vulnerabilities taxonomies

4.1.1 Literature Review

To the best of our knowledge, there is no vulnerability taxonomy that comprehensively addresses hardware-involved vulnerabilities. A good vulnerability classification could provide to security analysts information such as statistical data about vulnerability frequency, trends, incident correlations, and assessment of countermeasures effectiveness.

Furthermore, a taxonomy could allow an instant view of where and when the vulnerability was created during the design phase. This is our primary concern for a taxonomy, since we want to identify and study those that were created before the pre-silicon validation process.

Classification schemes are usually linked to the concept of taxonomy. Quoting from Gregio's PhD thesis[27]:

> "... a number of taxonomies have been proposed to cover diverse computer security topics. For example, the works of Aslam[28], Krsul[29] and Landwehr address system vulnerabilities and attacks[30]. ... The Neumann-Parker taxonomy addresses intrusions, whereas other taxonomies related to intrusion or threats to computer security are summarized by Lindqvist and Jonsson[31] ... Many are the requirements [32] that a taxonomy has to meet in order to accomplish its goal, i.e. to be clear, adaptive and applicable. Howard and Longstaff[33] and Amoroso[34] mention important properties of good taxonomies, which are listed as follows:
>
> - **mutually exclusive**, to assure that a sample fits into only one category;
> - **exhaustive (or complete)**, so that the predefined categories include all possibilities of the subject under analysis;

- **unambiguous**, to eliminate uncertainty and to allow the taxonomy application to be a clear process;
- **repeatable**, so that others can repeat the taxonomic process and get the same results;
- **acceptable and useful**, to allow it to be used by the community, serving as a reference and source of knowledge in the field.

In addition, Bishop states that a taxonomy requires well defined terms, so that there is no confusion about the meaning of a term[35]. Another interesting property of a taxonomy is that it should be comprehensible, i.e. either a field's expert or a merely interested person are able to understand it[31]. Therefore, a taxonomy should provide the discernment ability to its "users". Thus, it is possible to clearly discern which features segregate the analyzed samples into distinguished classes and which are the features that make given classes to be closer to each other."

Previous attempts to classify hardware attacks have been made, most of them focusing exclusively on physical or electrical attacks, such as chip micro-probing, analysis of electromagnetic emanations, and counterfeiting attacks[36].

A broader taxonomy that includes an extensive range of software attacks, including the ones that involve hardware, is maintained by the CWE, but it is focused on the software aspect of the attacks, and there are no specific categories for classifying hardware attacks.

Forristal [37] presented a "starting taxonomy" focused on hardware-involved software attacks. In order to qualify as a hardware-involved software attack, the following characteristics must be present:

- originate in a low-privilege system component;
- leverage or depend upon a hardware operation;
- achieve a vulnerability in a higher-privileged software layer or a peer in the current software layer.

Forristal also presented various specific attack scenarios which could be considered subcategories of hardware-involved software attacks. Although relevant in trying to *classify* defects, neither Abraham's nor Forristal's proposals represent good taxonomies according to Amoroso, Howard & Longstaff, Bishop, and Lindqvist & Jonsson. We believe, however, that fulfilling every "good taxonomy" requirement for hardware issues may not be feasible, as exemplified by the fact that, in spite of many attempts, there is still none for software[27].

Potlapally [2] presented a classification of hardware attacks into 4 types:

1. Active adversarial manipulation of hardware control signals.
2. Security gap in interaction of multiple platform features.
3. Insecure platform initialization by boot-up firmware.

4. Ability of untrusted or lesser privileged entities to maliciously influence hardware operation.

These types are able to classify most of the hardware-involvement vulnerabilities, but are not very helpful in our scenario, where the taxonomy should be able to pinpoint when and where the vulnerability was created. As an example, an insecure platform initialization could be caused by a firmware bug, but could also be the result of a problem with the chipset architecture.

4.1.2 Hardware vulnerability taxonomy

Since publicly available hardware-related vulnerabilities are scarce, and thus insufficient for statistical purposes, we focused our attention on the "usefulness" criterion. As our final objective is to evaluate security, our classification scheme pays special attention to vulnerability root causes.

Although previous works contribute new ideas for the classification of hardware-involved software attacks, they lack specific aspects that are important for our research goal. Namely, they do not differentiate attacks that arise from (i) hardware architectural problems and (ii) software misuse of sound architectures.

This fact led us to analyze hardware-involved software attacks to isolate the ones that were caused by architectural problems. We found that there were relatively few attacks that exploit architectural problems, but this led us to useful insights about how to differentiate them. In the following, "H" ("hardware-only") represents case (i) above, while "S/H" ("Software with Hardware involvement") represents case (ii):

- Scope: an H attack can be exploited in multiple platforms and operating systems (wherever the same chip/chipset is used), while S/H attacks can be carried only on specific applications or platforms;
- Countermeasures: an H problem is potentially much more difficult to correct (e.g. requires silicon or ROM revision) while S/H can be corrected by software (or firmware, in the worst case);
- Accountability: the hardware manufacturer is probably the sole responsible for H problems, while the S/H attacks may involve multiple vendors;
- Documentation: an H attack can be elaborated by learning the (public) documented features and details of a given architecture and then trying to find security holes. A S/H typically involves learning from binary code disassembling or other types of analysis that intend to discover undocumented aspects of a software.

More specifically we also found that:

- Detection: it should be possible to detect H attacks by the careful analysis of an architecture specification by vulnerability extrapolation, probabilistic analysis and/or formal methods; which method is the most promising is an open subject;

- Frequency: H attacks seems to be relatively rare. The majority of attacks published as "hardware attacks" are in fact S/H attacks;

- Complexity: At least one of the H attacks (SMM attack) does not rely on hardware bugs or undocumented features. They rather exploit specific and non-obvious interactions between various parts of the system, e.g., CPU and northbridge, CPU and PCIe bus;

- Definition: From a hardware engineer point of view, it seems to be very difficult to know whether a given unintended system operation can lead to a security breach, since it may be necessary to combine multiple unintended operations from various components in order to have a security breach. Since any unintended system operation could possibly lead to a security breach, it would be safer to define any unintended system operation as an attack.

From this analysis (H and S/H) it became clear that all vulnerabilities could be clearly classified between one of these two classes, and would help us to address the vulnerabilities that we were interested in. Thus, we created the "Hardware architecture design" class for the H vulnerabilities, and "Hardware misuse" for the S/H ones. [1].

Inside the category of architectural problems (H), we also found that there are two very distinctive types of attacks: the ones caused by the interaction between various parts of a system and the ones that are caused by a specific "defect" in a component. Thus, the former attacks are classified as "Incompatibility between the security mechanisms of subsystems", and the later as "Architectural flaw of a single subsystem".

The distinction between hardware problems (H) and hardware-involved problems (S/H) is very meaningful for the analysis of the vulnerabilities, but brings the following challenges to the creation of a taxonomy:

- there are relatively fewer hardware-involved attacks, compared to software-only attacks. With fewer attack samples, it is hard to group vulnerabilities together into categories and being complete;

- it is relatively difficult to define how to describe a hardware-involved attack: opposed to most of the software-only attacks (e.g. buffer-overflow, SQL injection), hardware-involved attacks tend to be relatively more intricate, and have a strong dependency on the system architecture. Taking too much detail into account may force the taxonomy to have too many categories, limiting its usefulness for gaining insight into the attacks;

- small variations of a given vulnerability may cause drastic changes in its classification in a taxonomy when the "deterministic" requirement is mandatory. However, if this requirement is dropped, the methods of vulnerability extrapolation, specially regarding the combination of root causes, could be used to group issues by similarity.

[1] The grouping proposed by Forristal are in fact all subclasses of S/H (or "Hardware misuse") class of vulnerabilities

Still, we were able to create a taxonomy that is satisfactory for our purposes, since it differentiates the main classes of attacks and incorporates root-cause information, which we expect to be useful towards our objective.

Our proposed hardware vulnerability taxonomy

1. **Hardware architecture**:
   (a) **Hardware architecture design**: Vulnerabilities that are arise from a insecure protocol or hardware architecture design;
      i. **Architectural flaw of a single subsystem**: vulnerabilities that arise from a security problem from a single module. It does not depend or is interfered by other modules;
      ii. **Incompatibility between the security mechanisms of subsystems**: vulnerabilities that arise from the interaction between various modules or subsystems;
   (b) **Architecture implementation**: Vulnerabilities that arise from flawed implementation of a secure hardware architecture.
2. **Hardware misuse**: vulnerabilities that arise from the misuse of hardware features by the system software or firmware. From our literature study, this is where most of the hardware-involved attacks reside. Most CVE-listed hardware attacks relates to vulnerabilities that allow an attacker to leverage a hardware feature to gain privileged access to some system resource, but that could be avoided by the correct implementation of drivers or other software;

Although it is not the primary focus of our work, we further classified the Architecture implementation vulnerabilities:

1. **Architecture implementation**
   (a) **Inadvertent**:
      i. **System description bug**: hardware bugs inserted during high-level system description (e.g Verilog);
      ii. **System synthesis bug**: hardware bugs inserted during circuitry synthesis (RTL/GDS);
   (b) **Intentional**:
      i. **Malicious**: backdoors inserted by a third party;
      ii. **Test or recovery modes**: non-documented manufacturer test modes that allow a privileged access to a system resource.

Table 4.1 shows some of the studied attacks classified under our proposed taxonomy.

Attack	Exploited vulnerability	Classification
SMM Memory (cache)	Interaction between cache write-back mechanism and chipset	Incompatibility between the security mechanisms of subsystems
Q35 memory remapping (VM)	Interaction between chipset memory remaping and CPU MMU	Incompatibility between the security mechanisms of subsystems
Q35 memory remapping (AMT)		
MSI/SIPI	Abuse of the PCI bus protocol	Architectural flaw of a single subsystem
UHCI USB		
Pentium FOOF		Architecture implementation

Table 4.1: Classification of some studied attacks under our proposed taxonomy.

Taxonomy and design phases

The types of vulnerabilities of the proposed taxonomy can be mapped to hardware design phases, from the architecture design phase up to the integration with firmware and drivers. Figure 4.1 shows how each type correlates with a specific design phase. The design phases that are shown represent a high-level view of the design process, and can be summarized as follows:

- Architecture design: the design of an architecture that satisfy the product desired specifications, resulting in a series of text documents, block diagrams, and reports;
- Front-end design: the RTL coding (in Verilog or VHDL, for instance) and of the specified architecture. Also includes the high-level synthesis of the design;
- Back-end design: encompasses the activities involved in the low-level synthesis and layout of the chip, such as floorplanning, placement, routing, and physical verification;
- Firmware/driver: the coding of the firmware and drivers that are necessary for the use of the chip. This is not strictly a hardware design phase, but is an activity with close involvement with the hardware, and is typically carried out by the hardware manufacturer.

Although the mapping between the taxonomy and design phases is not perfect, it can be used to pinpoint the most probable source of origin for any given vulnerability.

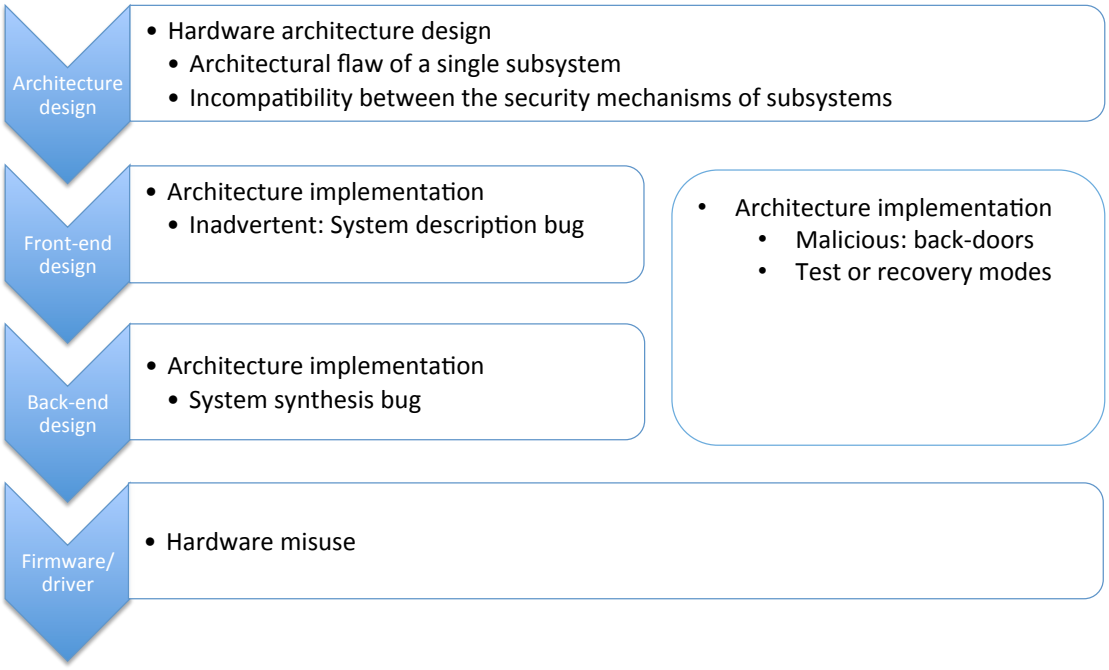


Figure 4.1: Taxonomy and design phases.

4.2 Methodology for Modeling Hardware Architectures

After developing the hardware vulnerability taxonomy, we focused on the "Hardware architecture design: Incompatibility between the security mechanisms of subsystems" type of vulnerabilities, since it is the type of vulnerability that is most relevant during the architecture review phase.

In order to allow a more structured and automated analysis of the involved subsystems, we created a way to model the hardware architecture. The main desired characteristics of this model were:

- incorporates all possible security aspects of each subsystem, so that an automated analysis would be possible;
- it is easy to transcribe from the architecture manuals: since the modeling phase is a manual procedure, it should be as intuitive as possible, and also easy to check;
- does not use hardware source files (e.g. Verilog, RTL); this is one of the premises of this work

4.2.1 Literature Review

We did a literature review of the existing methods and proposals for the modeling of hardware architectures, and analyzed if they could be used for our purpose of hardware security evaluation.

UMLsec [38] presents a UML profile and a method to evaluate the security of general systems, but it is targeted at another case scenario. Some of its most important stereo-

Stereotype	**Base Class**
Internet	Link
Encrypted	Link
LAN	Link
Secure links	Subsystem
Secrecy	Dependency
Integrity	Dependency
High	Dependency
Secure dependency	Subsystem
Critical Object	Subsystem secrecy

Table 4.2: List of various UMLsec stereotypes.

types [39] are shown in Table 4.2. Thus, it lacks sufficient support for describing hardware with the necessary detail level.

Embedded UML [40] is another UML profile for the specification and analysis of real-time software and hardware systems, but also lacks support for detailed hardware description.

SysML [41] is probably the best known and most used system modeling language, and can also be used to model hardware-only systems. However, we found it to be not optimized for the description of hardware architectures in our scenario. The resulting diagrams would be fairly complex and difficult to visualize. We also would have to extend it, in order to represent some of the security-related aspects of the architecture.

Taha et al. [42] presented an open framework for detailed hardware modeling, which is a part of the OMG standard for Modeling and Analysis of Real-Time and Embedded systems (MARTE). This framework consists of a set of UML extensions to allow the description of physical and logical properties of a hardware design, but is more oriented to embedded system design, and suffers from the same complexity issues of SysML.

Harmless [43] is a hardware architecture description language dedicated to real-time embedded system simulation, and can be used to model and simulate real-time systems down to the instruction set and the micro-architecture level. It does not have a specific focus on security aspects of the underlying architecture, and is not meant to model a whole system besides the CPU core.

The Fortuna framework was developed by our research group (Gallo et al.[44]) for the very purpose of analyzing hardware and software architectures. Although the Fortuna framework is able to provide a meaningful model of a given system, it still requires a relatively large amount of manual work. Mainly, we still do not have an automated way of calculating the probabilities that the framework requires for its quantitative analysis.

We evaluated the Fortuna framework for the same hardware architecture and vulnerabilities that we used in the toy case (SMRAM memory). In our analysis we did not model the entire system, since the workload would have been substantially greater than for the Assurance Case framework. Our conclusions about the Fortuna framework are:

- since Fortuna relies on probabilities that are estimated from low-level design statistics (number of kloc, etc), we would not be able to have a model only from public datasheet information;

- it is difficult to model only the target subsystem and isolate the rest of the system. To model the SMRAM, in this specific case, would require modeling other CPU subsystems, contributing to the increase in workload.

4.2.2 Our hardware modeling approach

Our model is similar to a hardware block diagram, and tries to mimic the way that hardware diagrams are represented in standard documentation, such as datasheets. Figure 4.2 depicts a typical diagram. In these diagrams, each subsystem is represented by different boxes, with arrows representing data buses that connect devices with each other. Devices can also have logical partitions inside the same hardware subsystem (e.g. bootloader memory regions), and they can also be represented by different boxes. Each box can represent different types of entities (i.e physical or logical), which could be confusing for some types of analysis. For security validation, though, this mix is necessary in order to better represent the underlying system.

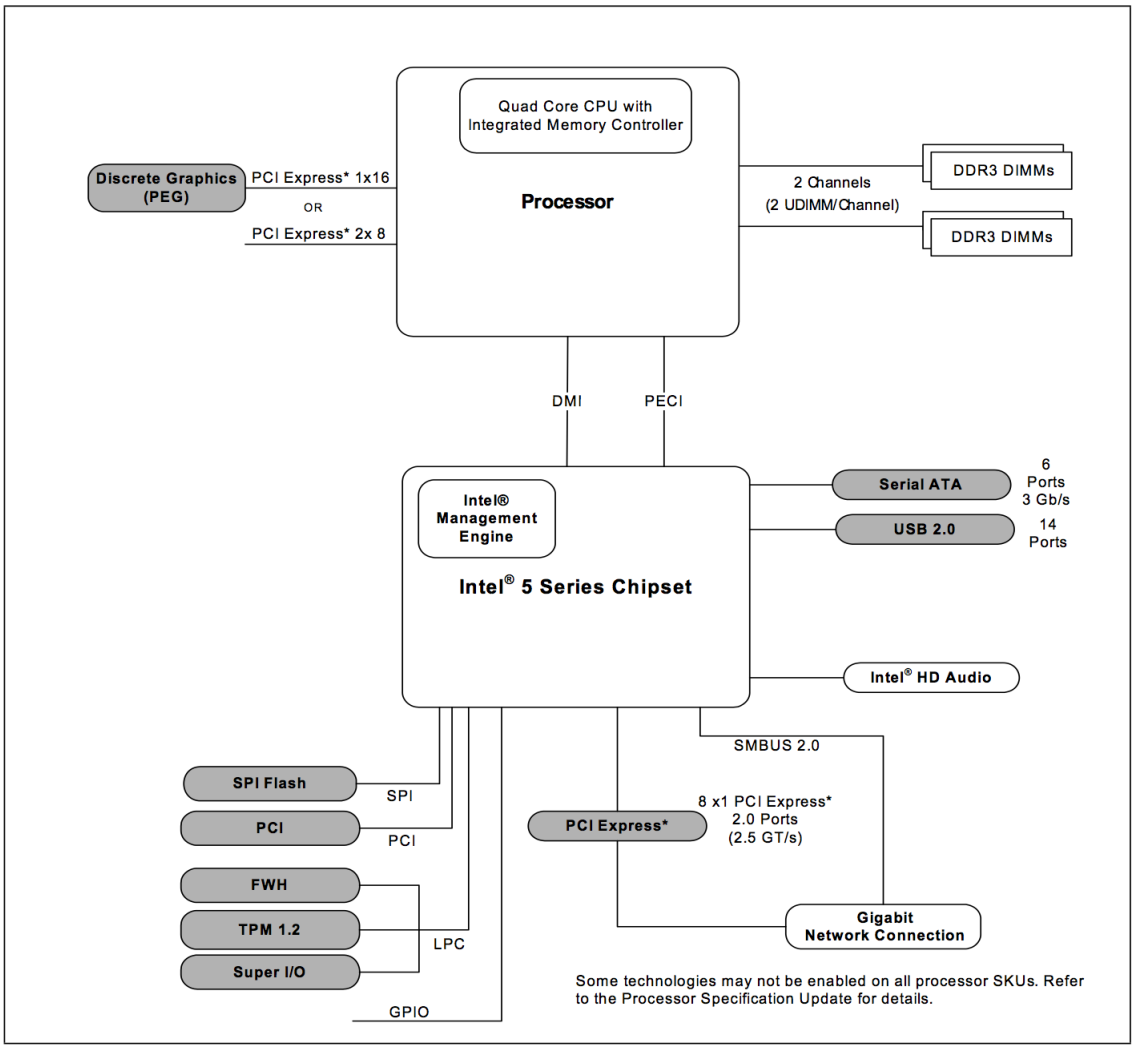


Figure 4.2: Intel i7 platform diagram.

Our modeling framework uses the same general format of the datasheet diagrams, but

Device type	Description
peripheral	An unknown peripheral that can be attached to the system
timer	System timer
rtc	Real Time Clock
bus	Data bus
bus-controller	Device that configures the mode of operation of a data bus
cpu	Processing unit
interrupt-controller	Interrupt controller
bus-device	Generic peripheral
dma-controller	DMA controller
rtc	Real-time clock
memory	Physical memory
memory-region	Memory region inside physical memory clock
io-controller	Inputs/outputs to external world

Table 4.3: List of possible device types for HW architecture description.

has a structured and well-defined way of describing devices types and capabilities. In this way, the resulting diagram is still familiar and easy to understand for hardware designers, but contains enough structured information to enable a manual or automated security analysis.

Our hardware representation consists of interconnected blocks, similar to the one depicted in Figure 4.3.

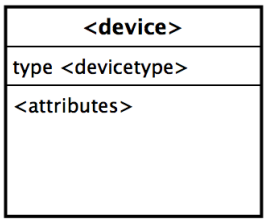


Figure 4.3: Hardware description block.

The DEVICE field is used to inform the reference name of the device represented by the block (e.g. `SATA-controller`).

The DEVICETYPE field is used to describe the device type. It contains the keyword `type`, followed by another keyword that describes the device type (e.g. `bus-controller`). The list of valid types is presented in Table 4.3.

The ATTRIBUTES field describes the capabilities and other attributes of the device. A device can have one or many items in this field, and they should be described using specific keywords. Examples of keywords are `read-mem` (the device can issue memory read requests), `generate-interrupts` (the device can generate interrupts), `store-data-non-volatile` (the device can store non-volatile data in itself).The list of valid attributes is presented in Table 4.4.

Figure 4.4 shows part of a hardware architecture with various node types.

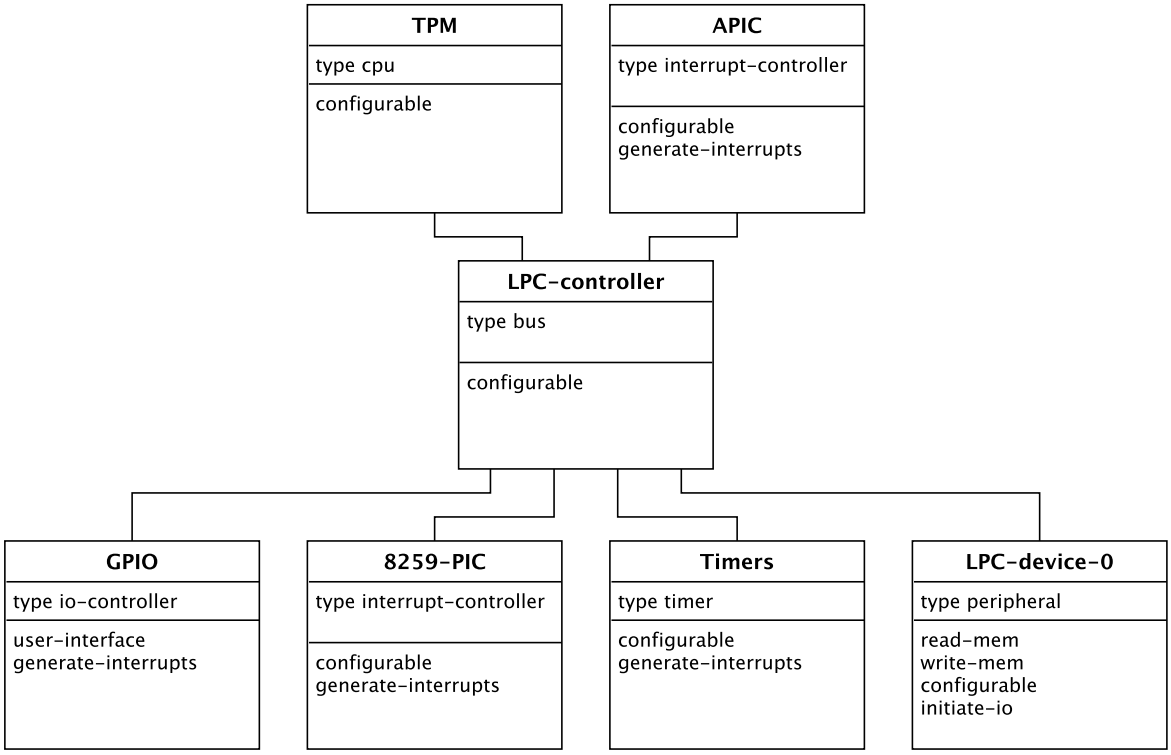


Figure 4.4: Hardware architecture description nodes.

Attribute	**Description**
store-data-volatile	Device that has the capability of storing data that is lost after system powerdown (e.g. registers, DRAM)
privileged-mem	Memory region that stores keys, CSPs, or privileged code and data
data-transfer	Device is able to transfer data from one external device to another external device
cache	Cache memory
configurable	Device has configuration capabilities via registers
read-mem	Device is able to issue memory reads
write-mem	Device is able to issue memory writes
initiate-IO	Device is able to initiate I/O transfers
mem-remap	Device can act as a bus bridge with memory remapping capabilities
store-data-non-volatile	Device that has the capability of storing data that is not lost after system powerdown (e.g. NVRAM, SSD)
bus-master	Device can take control of a bus
generate-interrupts	Device can issue interrupt requests
executable-code	Device contains executable code (e.g. BIOS, BIOS extensions)
user-interface	Device has interface to user controled I/O (e.g. keyboard interface, ethernet)

Table 4.4: List of possible device attributes for HW architecture description.

Chapter 5

Security Analysis Framework

In the previous chapter we proposed a method for modeling the characteristics of the hardware architecture. In this chapter, we present a method for systematically analyzing the security of resulting hardware models, followed by a method for automating part of the analysis process.

5.1 Assurance Case applied to Hardware Security

Considering that hardware vulnerabilities are generally difficult and expensive to patch, it is easy to understand why hardware functionality validation is a very rigorous process. The same rigour exists in the safety analysis of critical systems that may pose a life risk to its users. Since the Assurance Case methodology has already been successfully used for safety analysis (in its more restrict Safety Case variant), it has a potential for other types of rigorous analysis such as the hardware architecture security validation.

There is not much information about previous use of assurance case for hardware systems, but there has been some use of it in the software industry [25][45]. However, it still is not a widespread practice. In fact, it is not clear whether the necessary work to build and maintain an Assurance Case pays off in the long term. Software can be very complex and fast evolving, meaning that developing a case tends to be very expensive and has to be updated very often. Besides that, there is a large interdependency between software modules that would require the Assurance Case to be built for the entire system including the OS, drivers, etc.

However, we believe that hardware systems share more commonalities with the safety case. Besides being less complex than software, hardware is also easier to modularize. In fact, most hardware is designed and separated into modules that are physically isolated, with a well defined inter-communication interface (i.e. there is no way for a subverted module to bypass its communication interface and subvert another module that is physically separated). Hence, unlike software components, these modules can be separately analyzed.

Table 5.1 summarizes this comparison between the characteristics of safety systems with hardware and software systems. In this table, locality is defined by how distinct subsystems can be isolated. Physical systems are bound by the physical laws that apply

	Software security	Hardware security	Mechanical systems (safety)
Adversary	The adversary applies a set of coordinated actions to attack the system.	The adversary applies a set of coordinated actions to attack the system	No adversary - the system is stressed in an uncoordinated way.
Complexity	Highest (e.g. MS Windows XP had 11M lines of code [46])	Higher (e.g. LEON3 SoC has 600k lines of code [47])	High
Locality	Low	Medium	High
Cost of patches	Lower	Higher	Higher

Table 5.1: Comparison of software, hardware, and safety systems.

to the mechanics of the components. Hardware systems have electrical interconnects that limit the ways in which one subsystem can influence and communicate with another. Software systems can have logical separation between subsystems, but, unless they are hardware-enforced, a security breach can give a subsystem the possibility of influencing potentially any other subsystem.

From our understanding of the hardware security validation problem, we expect the following benefits from the use of the Assurance Case methodology:

- Build a well documented security analysis that can be re-used, shared, and reviewed.
- Establish a well defined evaluation process with an indication of the analysis completeness.
- Allow the evaluation work to be split between different teams.

Since a complete assurance case should contain all the conditions (i.e. evidence) necessary to the security of a system, we also consider that it can be used to guide the generation of test cases for a system. Thus, the methodology can be used both during pre-silicon and post-silicon validation.

In the next session we demonstrate how the methodology can be applied to security analysis during the architecture review phase of a CPU design.

5.1.1 Example of an Assurance Case applied to hardware

In this example we show how Assurance Cases can be used in a real-world architecture analysis. The case was built using the D-Case editor [48], which is an Eclipse-based tool to build dependability cases using GSN (Goal Structuring Notation). D-Case has evolved to support argument patterns and integration with other dependability analysis tools, but for this case we are using it to develop a security-related case using GSN.

We analyzed the System Management Mode (SMM) subsystem of the Intel CPU architecture, from the point of view of an architecture review analyst. SMM is the system management mode of most modern Intel CPUs. It allows the computer's BIOS to control

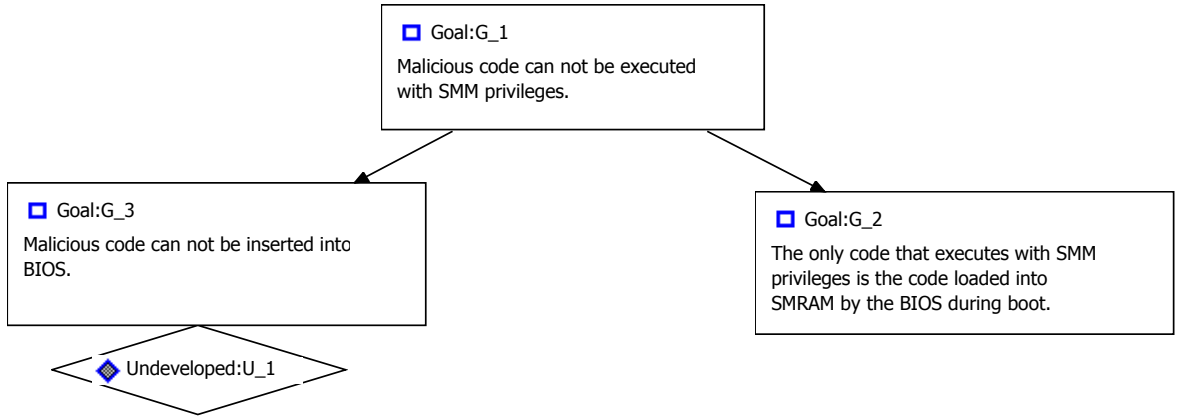


Figure 5.1: SMM Assurance Case - Step 1.

system variables such as fan speed control, independently from the operating system. For this reason, SMM code runs at a higher privilege than the OS itself.

This architecture was chosen because it is relatively simple and had two known vulnerabilities [4]. The target architecture refers to the first-generation Intel Core processor with the Q35 chipset, which can be considered obsolete but is still valid for this exercise.

The first known vulnerability is that the CPU only restricts SMM memory access from external memory read/write operations. And yet, once inside the CPU cache, the SMM code could be overwritten, effectively bypassing the restriction mechanism [49].

The second vulnerability is that the memory which stores the SMM code had its access restricted by the CPU. However, the external chipset was able to remap that memory to a unrestricted address, overriding the CPU protection mechanism [50].

In order to facilitate the readability and explanation, the analysis was divided into four distinct steps that are shown in Figures 5.1 to 5.4. Note that the D-Case editor already inserts a unique identifier to each element (e.g. "G_1"), so that it may be later referenced in a textual version of the case, or used by other tools.

We already had previous knowledge of the SMM system, and it took approximately 20 hours to build this case, using 483 pages of architecture documentation. Most of the effort was spent in the search for specific information in the architecture manuals, but we consider that the time was used in an efficient way; instead of browsing through the manuals searching for possible vulnerabilities (as in the standard manual analysis process), we had a clear goal of searching for evidence to support our case.

SMM Assurance Case - Step 1 (Figure 5.1).

The analysis process starts by stating the global desired system security property that we are interested to assure. In our case, it is the security of the SMM system: we want to assure that malicious code cannot run with SMM privileges. This is stated in goal G_1.

It is important to note that this goal is described with a negative sentence. This is due to the nature of the security assurance problem: since there are many ways in which an attacker may try to compromise the system, we have to systematically list and analyze

each of these potential ways. This is different from standard requirement specifications, where most features are listed as positive sentences (e.g. “The SMM system must control fan speed").

The topmost goal is then supported by two sub-goals, where one of them, G_3, is not developed in our case. This means that we do not have evidence to support it. In this case, the responsible party for this specific goal would be the BIOS vendor. This undeveloped goal would be useful for communicating to all BIOS vendors what is expected from the BIOS code in order to guarantee the SMM system security.

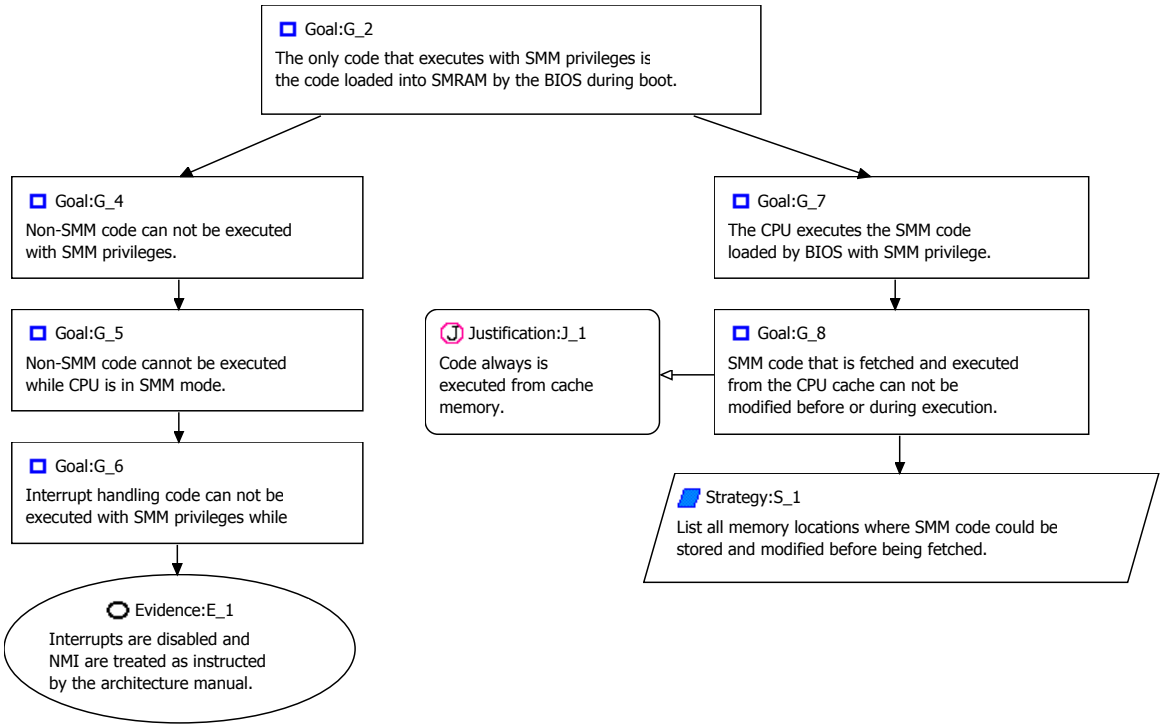


Figure 5.2: SMM Assurance Case - Step 2.

SMM Assurance Case - Step 2 (Figure 5.2).

In this step we expand goal G_2 from the previous step. Two sub-goals are derived, G_4 and G_7. G_4 is developed into G_5, which is itself developed into G_6. While it would be possible to state a single goal that would encompass G_4, G_5, and G_6, we prefer to keep them separate, so that their relationship is explicit. This makes it easier for a reviewer to detect possible inconsistencies or wrong assumptions.

Goal G_6 is supported by our first evidence, E_1, which is, in our example, a fictional check that our system correctly disable interrupts in a certain way that is required by the architecture reference manual. In a real-world scenario, this evidence could be linked to the associated interrupt-disabling code, and to the architecture manual paragraphs that state this need. In such a way, it becomes explicit that a possible SMM vulnerability may arise if that code is eventually disabled for some reason.

The other sub-goal from G_2, G_7, is developed into G_8, followed by a strategy box S_1. The strategy does not state a goal by itself, but only states the strategy that was used to develop the following sub-goals in step 3. Justification J_1 is a note that makes explicit why goal G_8 was developed in a specific way.

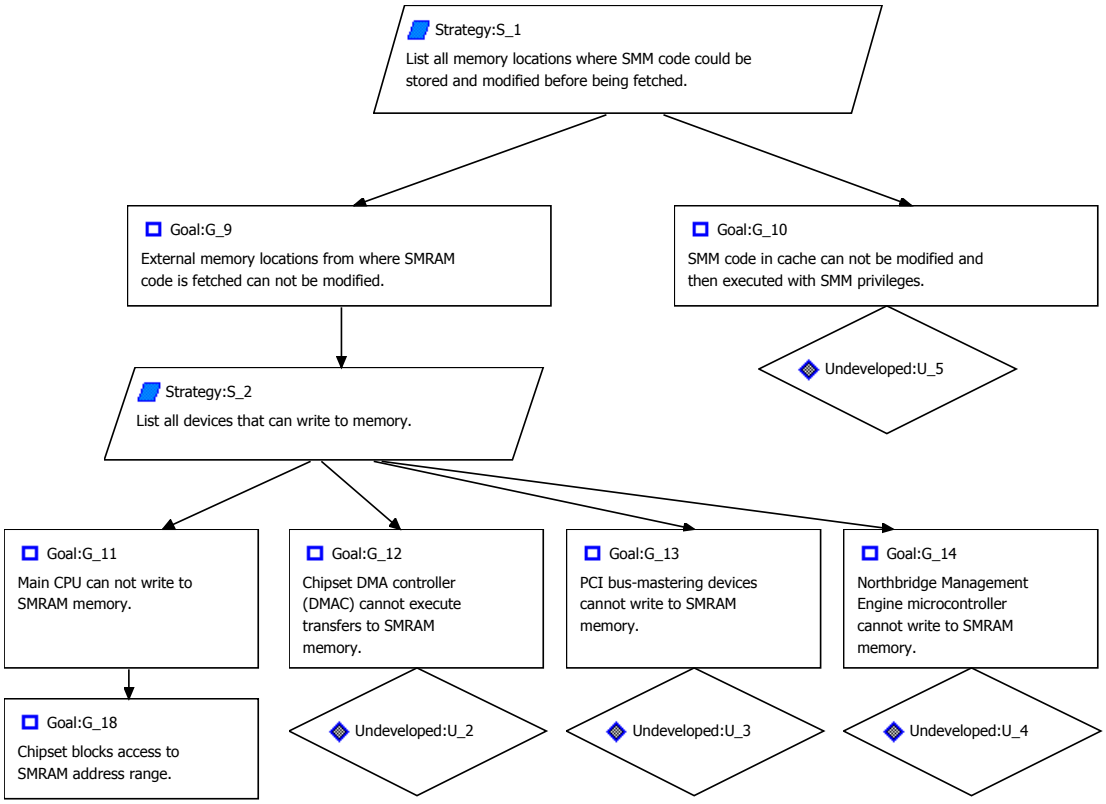


Figure 5.3: SMM Assurance Case - Step 3.

SMM Assurance Case - Step 3 (Figure 5.3).

In step 3, we show how strategy S_1 was used to derive sub-goals from goal G_8. In this step, we ran into a series of undeveloped goals that are directly related to possible vulnerabilities of the hardware architecture.

For U_2, U_3, and U_4, we did not have the necessary documentation to provide an evidence. Thus, they can be considered as possible unknown vulnerabilities that affect this system. Since uncovering new vulnerabilities was not the primary objective of this work, they were not further investigated.

For the undeveloped claim U_5, there is a known attack that overwrites the SMM code after it is cached inside the processor [49]. This is the first vulnerability that we were able to detect during the case's development.

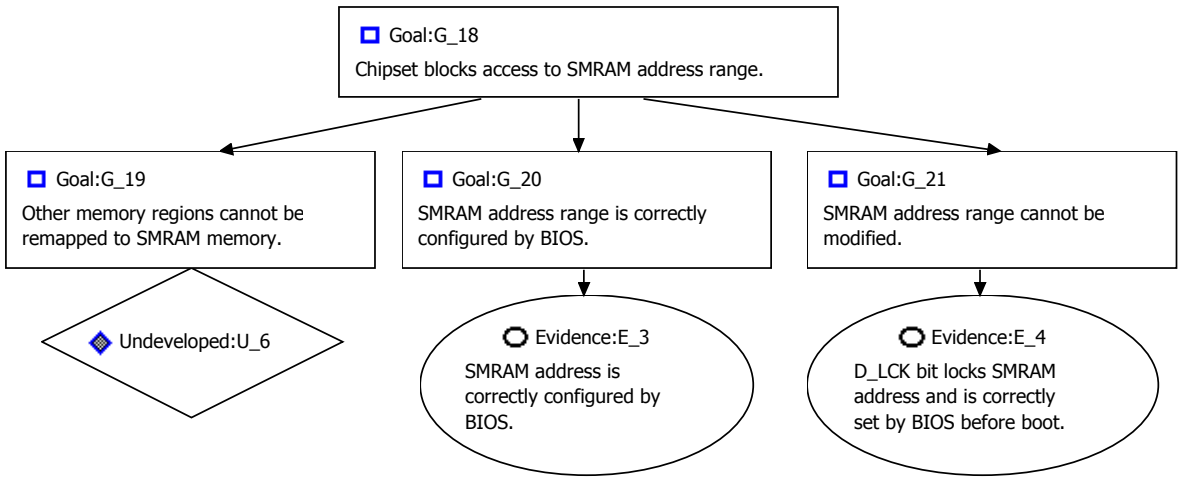


Figure 5.4: SMM Assurance Case - Step 4.

SMM Assurance Case - Step 4 (Figure 5.4)

Step 4 continues the analysis of goal G_{18} from step 3, and ultimately results in the undeveloped claim U_6.

This claim refers to the vulnerability that is described in Rutkowska [50], which is the memory remapping mechanism exploit to overwrite SMM code. Thus, we found the second known vulnerability that affected the SMM subsystem.

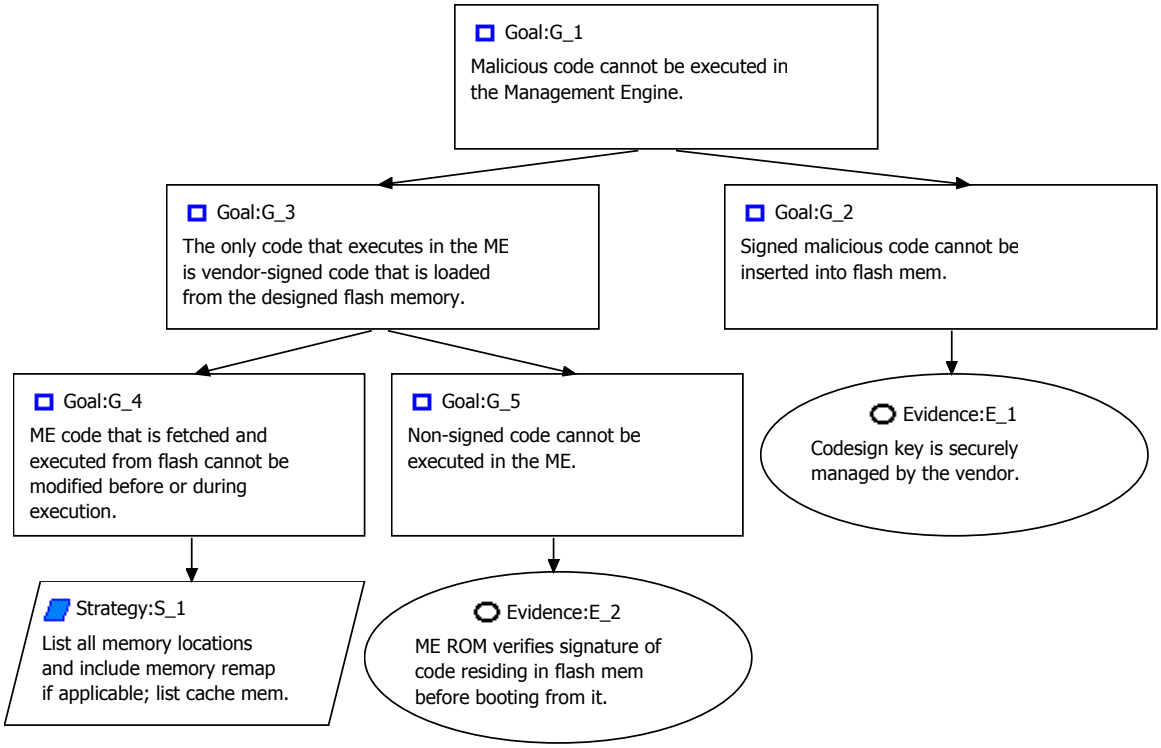


Figure 5.5: ME Assurance Case - Step 1.

ME Assurance Case - Step 1 (Figure 5.5).

In this assurance case, our goal is to guarantee that malicious code cannot be executed in the Management Engine microcontroller.

We start by asserting that malicious code cannot be executed in the ME (G_1), and then expand it by claiming that malicious code cannot be signed and stored into the flash memory (G_2), and that the only code that is executed is the signed code from flash memory (G_3). For G_2, we generate an evidence from the assumption that the vendors correctly store and manage their code-signing private keys.

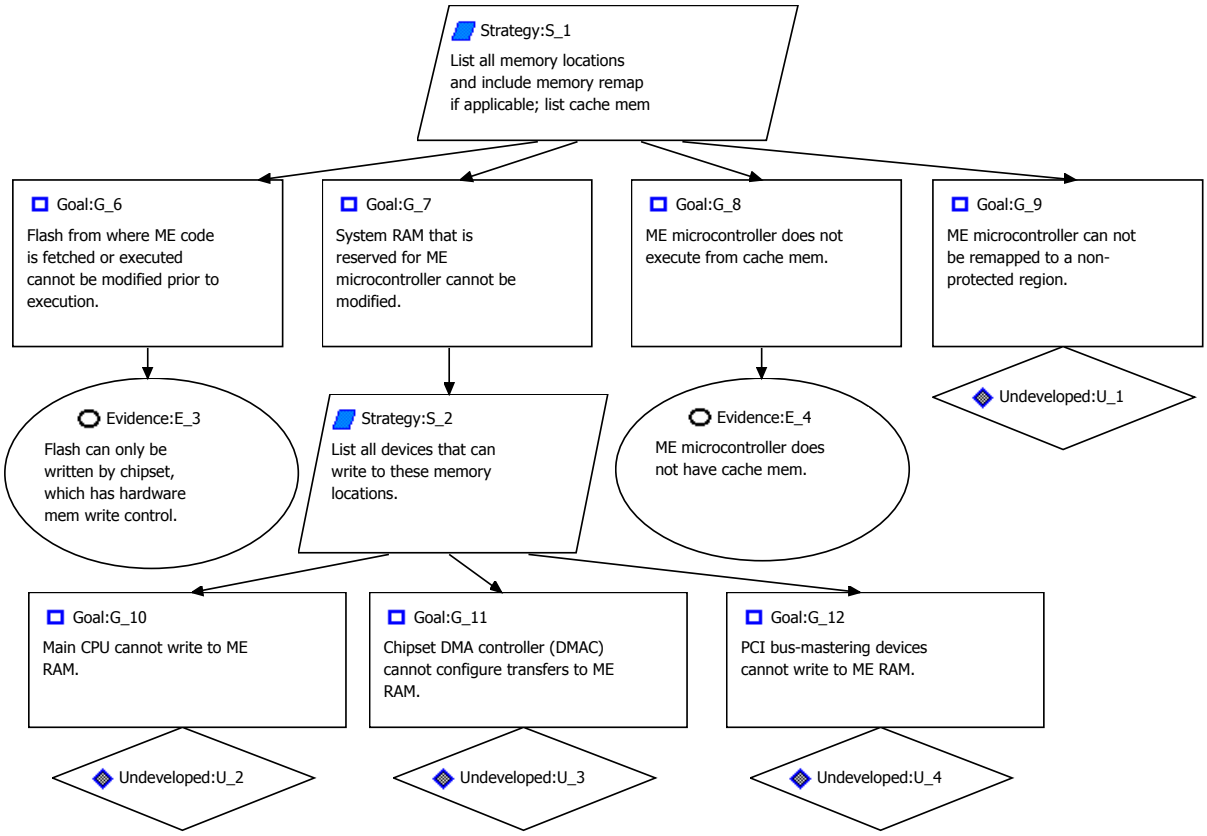


Figure 5.6: ME Assurance Case - Step 2.

ME Assurance Case - Step 2 (Figure 5.6).

In this step we develop the S_1 strategy, listing all memory locations from where the ME code can be fetched. This is a strategy similar to the one that was used in the previous example, where the SMM code had to be protected in a similar way.

From the sub-claims that follow, we have that G_6 and G_8 have supporting evidence. However, we did not find evidence in our documentation for G_9 and all the sub-claims that follow G_7. As in the SMM analysis, the lack of evidence may be just lack of proper documentation or be an indication of a real vulnerability.

For claim G_9, we know that there is a published attack [51] about the use of the memory remapping mechanism to exploit the ME system. For the other undeveloped

claims, we were unable to find any published attacks that could exploit these possible vulnerabilities.

An important note is that the ME documentation that is publicly available does not provide much detail about its architecture—it is more focused on programming models for software developers. While we were able to understand the basic architecture and develop this case, we are not sure that all the important hardware aspects of the system were captured.

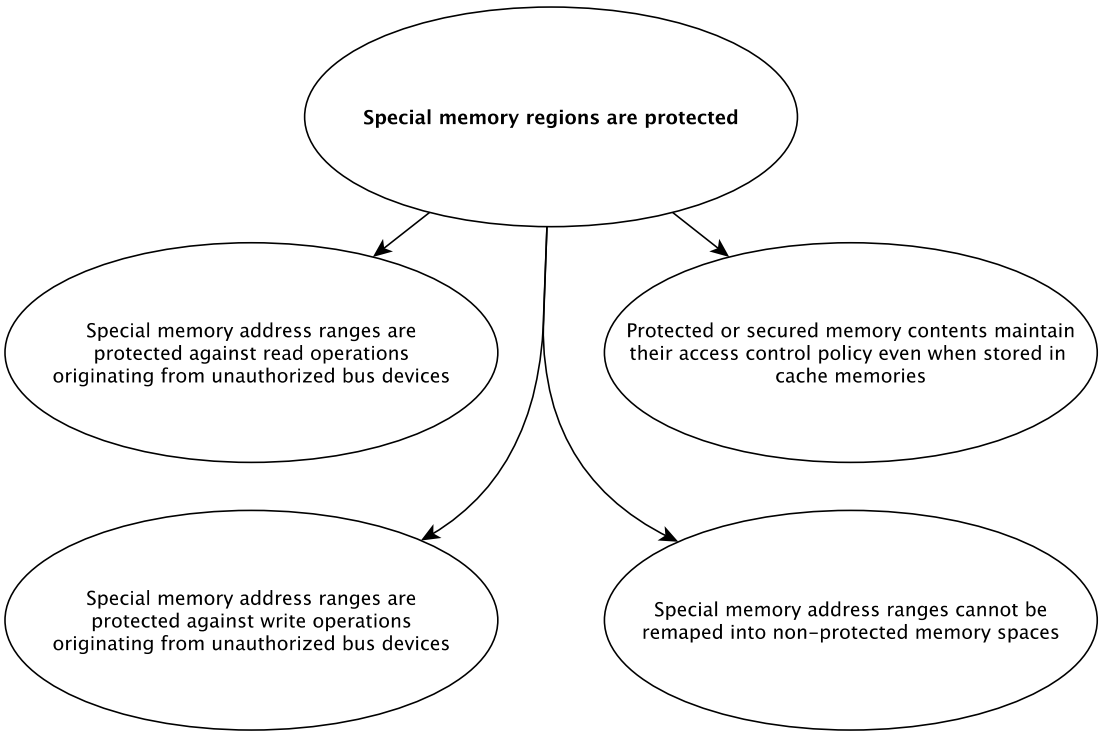


Figure 5.7: Assurance case pattern for memory protection.

5.1.2 Use of Assurance Case Argument Patterns

There are two important issues with Assurance Cases. The first is that it is very important to use the right strategies to develop the top level claim. An inexperienced analyst could miss important points during the case development, and end with an incomplete analysis. The other issue is that the case development tends to be a very time-consuming task, specially when the strategies have to be created from scratch.

To alleviate these two problems, we resort to the concept of Assurance Case argument patterns. Patterns are constructions similar to templates that can be used to provide guidance and speed up the process of building Assurance Cases. Ideally, these patterns should be created by experts, and then be used by less experienced analysts to build more solid Assurance Cases. There is already support for creating and applying argument patterns in some of the available tools [52][53].

As an example, we extracted an argument pattern from strategies that were used in the presented SMM and ME assurance cases. The pattern in Figure 5.7 states necessary conditions to guarantee that a certain memory location cannot be overwritten. It could be used to inform the security analyst some important details that have to be taken into account. For instance, in our example it is critical to know that memory contents can also be stored and modified in the cache memory without necessarily being modified in the external memory. This information is embedded in the argument pattern, and would be useful for analysis of similar systems.

5.2 “ACBuilder" - a tool for automated hardware security evaluation

ACBuilder is an application that we developed to guide part of the process of hardware architecture analysis, according to the proposed methodology. It is able to guide a security analyst through the development of the security assurance case, while also providing a certain degree of automation. It uses the following information as input:

- the target hardware architecture to be analyzed - described according to the method in Section 4.2.
- a top-level assurance case template - containing all the security goals that the system has to reach;
- a set of rules for the evaluation of possible security issues - one rule for each security claim.

As an output, it generates a text-based assurance case containing the necessary conditions and evidence that has to be collected in order to reach the security goals defined in the assurance case template.

In order to finish the analysis process, the security analyst must search and gather this required evidence, typically found in architecture manuals, and complete the assurance case.

Figure 5.8 shows an overview of the security analysis process with the help of the ACBuilder software.

5.2.1 Implementation

ACBuilder was implemented in Python version 2.7.5 and validated in the Linux platform. It uses the PyCLIPS version 1.0.7 module as a CLIPS engine.

The modeling of hardware architecture is done in the yEd graphical editor. yEd has native support for UML diagrams that can be directly used to model hardware architectures. The resulting model can be saved in the GraphML format, which is an open-source format that can be read and parsed by the ACBuilder application.

The top-level assurance case template is also drawn on the yED graphical editor and saved in the GraphML format.

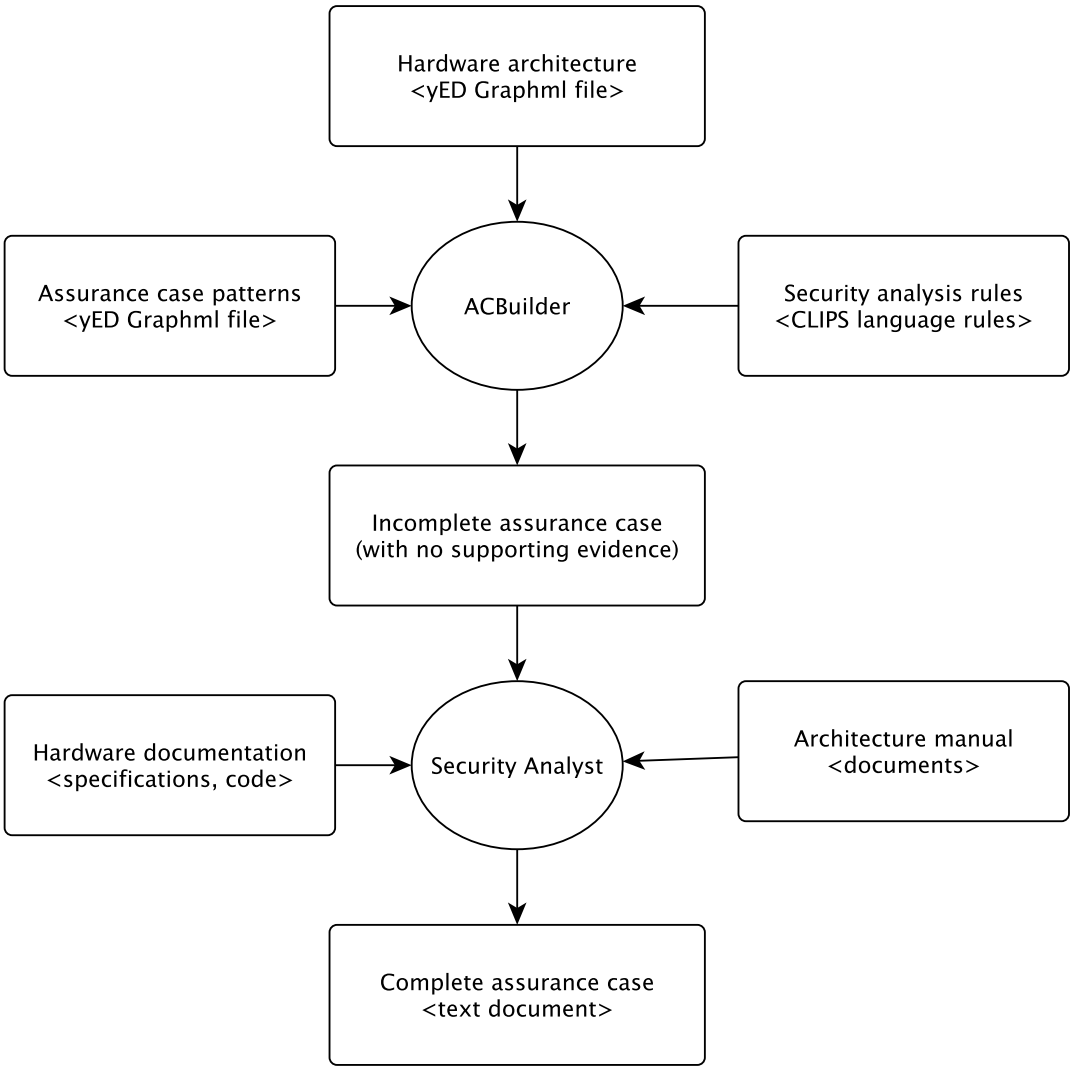


Figure 5.8: Overview of the security analysis process with the ACBuilder software.

The expert system rules can be edited using any text editor. The rules can be tested and simulated using any CLIPS interpreter. We used the CLIPS IDE version 6.3 for OS X.

The next subsections describe further details of the implementation, and how external software is used for the description of the hardware architecture and assurance case template.

5.2.2 Hardware modeling in CLIPS

In order to automate part of the analysis process using the CLIPS system, it is necessary to represent the hardware architecture description as proposed in Section 4.2 into a representation that can be parsed by a CLIPS program.

Describing components and the relationships between components is a straightforward task in the CLIPS language. Since they are part of the input to the expert system, they have to be described as facts. We use the following fact template to describe each com-

ponent:
(component (name < *device* >) (type < *devicetype* >) (can-do < *devicemethods* >) (connected-to < *listofadjacentconnecteddevices* >)

Figure 5.9 shows part of a real-world architecture, and Figure 5.10 shows how it is represented as CLIPS facts.

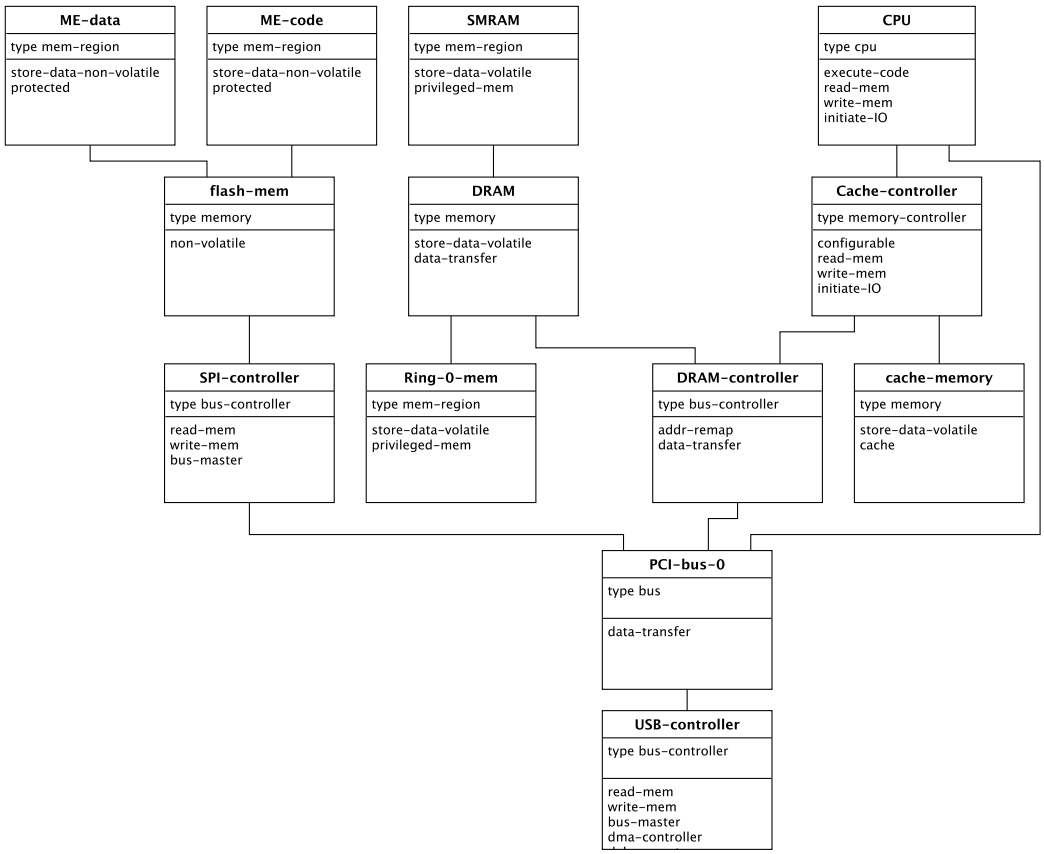


Figure 5.9: Sample hardware architecture to be converted to CLIPS facts.

In our implementation, the ACBuilder software is able to convert a hardware description in a graphML file into a corresponding set of CLIPS facts. The graphML file is obtained by saving a graph drawn in the yED software.

Describing Assurance Case Claims in CLIPS

The CLIPS language has a template element that is similar to structures in the C language. We used this language construction to describe the claims of an AC, forcing them to follow a structured notation. The code below is a sample of how a strategy may be described.

The above code defines a simple template that can be used to define a strategy to support a given claim in an assurance case. Basically, it defines that an aspect ("type" slot) of something ("name" slot) shall have a given attribute ("attribute" slot).

In this simple example, there would be only two aspects of the device ("memory" or "executes"), and two attributes ("is-read-only" and "only-intended-code"). This would

```
(component (name DRAM) (type memory )(can-do store-data-volatile data-transfer)(connected-to DRAM-controller Ring-0-mem SMRAM ))

(component (name CPU) (type cpu )(can-do execute-code  read-mem  write-mem initiate-IO)(connected-to PCI-bus-0 Cache-controller ))

(component (name Cache-controller) (type memory-controller)(can-do configurable read-mem  write-mem initiate-IO)(connected-to
DRAM-controller cache-memory CPU ))

(component (name PCI-bus-0) (type bus )(can-do data-transfer)(connected-to DRAM-controller SPI-controller CPU USB-controller ))

(component (name USB-controller) (type bus-controller )(can-do read-mem  write-mem bus-master dma-controller debug-port)
(connected-to PCI-bus-0 ))

(component (name SMRAM) (type mem-region )(can-do store-data-volatile privileged-mem)(connected-to DRAM ))

(component (name Ring-0-mem) (type mem-region )(can-do store-data-volatile privileged-mem)(connected-to DRAM ))

(component (name cache-memory) (type memory )(can-do store-data-volatile cache )(connected-to Cache-controller ))

(component (name SPI-controller) (type bus-controller)(can-do read-mem  write-mem bus-master)(connected-to PCI-bus-0 flash-mem ))

(component (name flash-mem) (type memory)(can-do non-volatile)(connected-to ME-code ME-data SPI-controller ))

(component (name ME-code) (type mem-region )(can-do store-data-non-volatile protected)(connected-to flash-mem ))

(component (name ME-data) (type mem-region )(can-do store-data-non-volatile protected )(connected-to flash-mem ))

(component (name DRAM-controller) (type bus-controller )(can-do addr-remap data-transfer)(connected-to Cache-controller DRAM PCI-bus-0 ))
```

Figure 5.10: CLIPS facts describing the sample architecture.

force the user to describe the claim using a standard syntax, and this would enable the expert system to automatically find applicable templates or sub-claims. If necessary, the user could combine various claims and connect them using logical operators.

Using CLIPS to suggest patterns and goals

Once a claim is described by using the template as described in the previous example, the CLIPS system can search if there are any rules that can be applied. These rules can either generate new sub claims, patterns or strategies. In order to be effective, these rules have to be created by a human expert, who will take into account not only the basic security principles, but also the aspects of the underlying system architecture.

The code in Figure 5.11 shows a rule that is activated when a claim is made that a processing system has to run only some intended code.

The security validation process using ACBuilder

There are five steps for the use of ACBuilder to run a hardware security validation:

1. Modeling the hardware architecture
2. Creating or customizing the Assurance Case Pattern
3. Creating or customizing the Expert System rules
4. Generating the Assurance Case
5. Collecting evidence

```
(reset)
(defglobal ?*index* = 0)

(deftemplate edge (slot from) (slot to) (slot cost))
(deftemplate cheapest\_paths (slot start) (slot stop))
(deftemplate path (multislot nodes) (slot cost))

(deftemplate component (slot name) (multislot type)(multislot can-do)(multislot connected-to))

(deftemplate writes-to (slot name))

(defrule extend\_path
 (path (nodes $?n ?y) (cost ?w))
 (component (name ?z)(connected-to $?a ?y $?b)(can-do $?c write-mem|data-transfer $?d))
 (test (and (neq ?z ?y)
 (not (member ?z $?n))))
 =>
 (assert (path (nodes $?n ?y ?z) (cost (+ ?w  1)))))

(defrule cheapest\_paths
 (declare (salience -10))
 (path (nodes $?n ?x) (cost ?w))
  (component (name ?x)(type ~bus)(can-do $?c bus-master $?d))
 =>
 (bind ?*index* (+ ?*index* 1))
(printout t crlf "=========== " crlf)
(printout t crlf "Evidence " ?*index* crlf crlf)
(printout t "Possible attack: \"" ?x "\" writes the \"" (nth$ 1 ?n) "\" privileged memory region using
the following path: " crlf )

(printout t "Path: " (implode$ $?n) " " ?x crlf)
(printout t "Solution (evidence): \"" ?x "\" cannot write the \"" (nth$ 1 ?n) "\" privileged memory
region."crlf )
(printout t crlf "Evidence description: " crlf))


(defrule test-privileged-mem-write
 (component (name ?x)(can-do $?a privileged-mem $?b))
 =>
(assert (writes-to (name ?x))))


(defrule initial\_path
 (writes-to (name ?x))
 =>
 (assert (path (nodes ?x) (cost 0))))

(load-facts arch.clp)
```

Figure 5.11: CLIPS rule to detect possible secure vulnerabilities regarding execution of code from non-secure sources.

Step 1 - Modeling the hardware architecture

The first step is to model the target hardware architecture. The modeling is done with the yEd graphical editor, using the built-in UML nodes. Each node should describe one of the following architecture elements:

- hardware device or peripheral (e.g. CPU, SATA controller)
- data bus (e.g. PCI, SMBus)
- logical device (e.g. memory region)

The NODE TITLE field should contain the reference of the device (e.g. SATA-controller). The title should not contain white spaces.

The ATTRIBUTE field should contain the keyword `type`, followed by another keyword that describes the device type (e.g. `bus-controller`). The list of accepted keywords is presented in table 4.3.

The METHODS field should contain the capabilities and other attributes of the device. A device can have one or many items in this field, and they should be described using specific keywords. Examples of keywords are `read-mem` (the device can issue memory read requests), `generate-interrupts` (the device can generate interrupts), `store-data-non-volatile` (the device can store non-volatile data in itself). Figure 5.12 shows part of a hardware architecture with various node types.

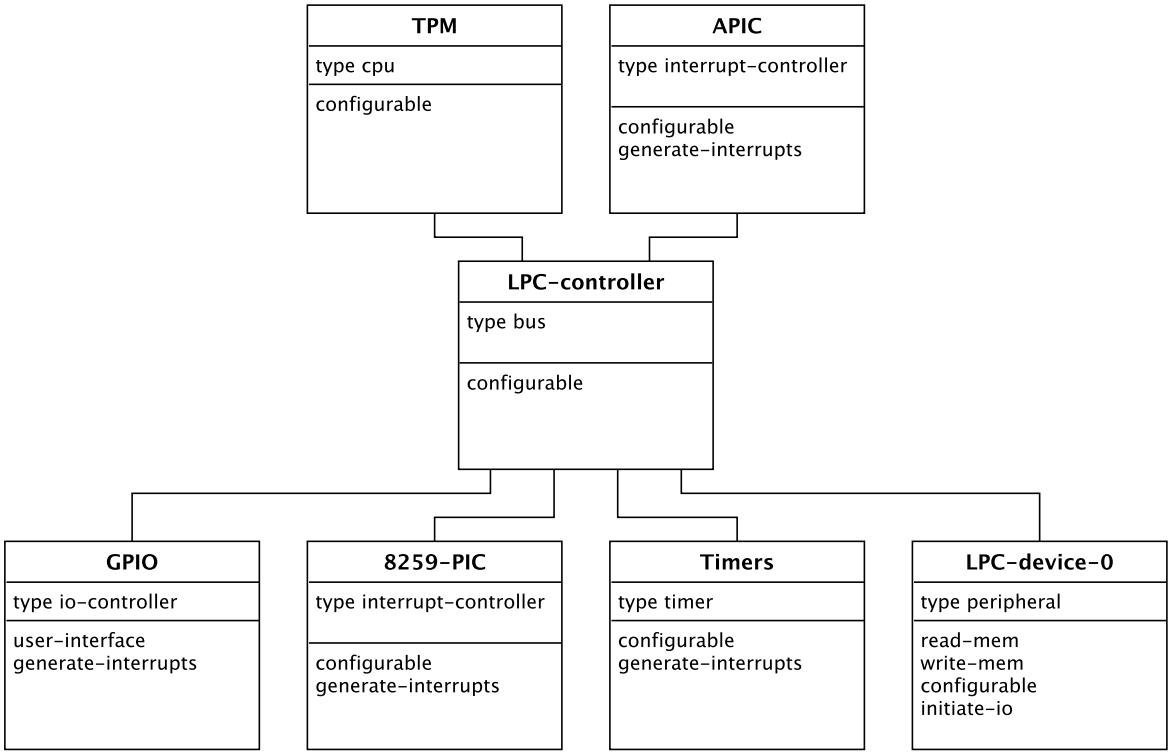


Figure 5.12: Hardware architecture description nodes.

Step 2 - Creating or customizing the Assurance Case Pattern

In order to execute the architecture automated analysis it is necessary to have an input Assurance Case pattern. This pattern contains the security aspects and rules that will be applied to the target architecture.

Similarly to the hardware architecture description, the Assurance Case pattern is composed of a graph of UML nodes.

The Assurance Case pattern contains a GSN (Goal Structuring Notation) structure representing the goals (claims) and solutions (evidence) to support the security objectives. The goal and sub-goals were created to provide coverage to the studied hardware attacks,

and the solutions (graph leaves) are associated with expert system rules that are applied to analyze the target architecture.

For each node, there is a title which should contain the identification of the goal or solution (e.g. `MEM-READ`). The title should not contain white spaces.

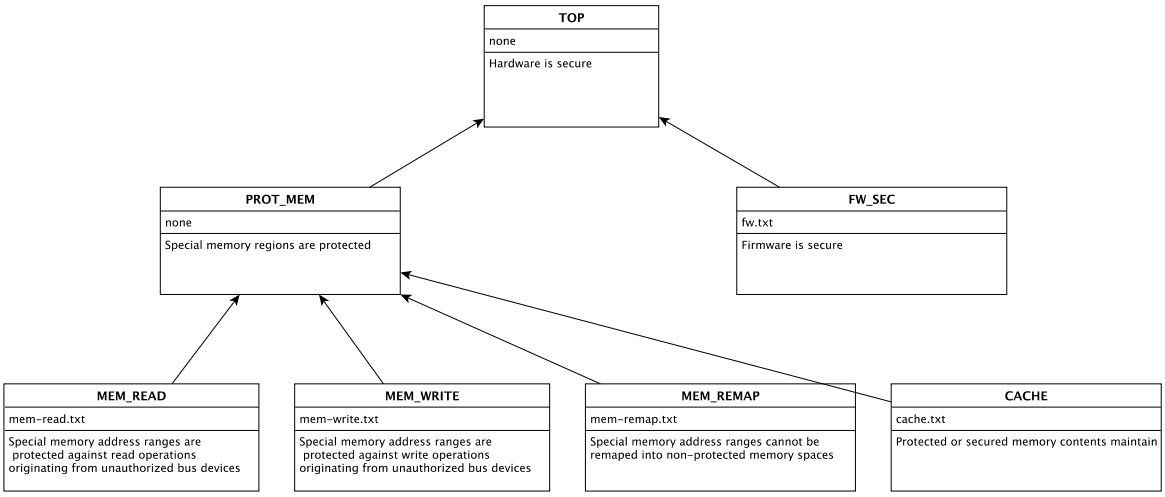


Figure 5.13: Assurance Case pattern.

The ATTRIBUTE field should contain the file name of the associated rule (in CLIPS language notation). Note that only the graph leaves (i.e. solutions) should contain a valid file name. The goals and sub-goals are not evaluated by the expert system, and should contain the `none` keyword. The list of provided rules keywords is presented in the Appendix.

The METHODS field should contain a description of the goal, sub-goal, or solution.

Figure 5.13 shows part of an architecture description with various node types.

Step 3 - Creating and customizing the Expert System rules

The expert system rules are used to generate the necessary arguments to support each of the goals stated in the assurance case pattern from Step 2. A sample rule is presented in Figure 5.11.

The rules have to be coded in the CLIPS language, and will be evaluated according to the Assurance Case Pattern from step 2. It will receive a description of the hardware architecture of step 1, in the format of CLIPS facts. Each architecture element is described as a CLIPS fact, as according to the syntax below:

```
(component (name <device name>) (type <device type>) (can-do
<attribute 0> <attribute 1> ...  )  (connected-to <connected device
0> <connected device 1> ...  ))
```

Figure 5.10 shows a sample architecture description that is provided for the CLIPS rule program.

Step 4 - Generating the Assurance Case

In this step, the assurance case pattern is applied to the target hardware architecture. This is done by the ACBuilder Python program. The ACBuilder program has the following syntax:

```
ACbuilder <assurance case pattern> <target architecture>
```

Both the assurance case and the target architectures must be encoded in the GraphML format. In our example case, the command is:

```
ACBuilder.py hw_template1.graphml uml_test.graphml
```

This command will generate the output shown in Figure 5.14.

```
==============================
Goal: Special memory address ranges are  protected against write operations
originating from unauthorized bus devices
Id: MEM_WRITE
Rule file: mem-write.txt

===========

Evidence 1

Possible attack: "USB-controller" writes the "SMRAM" privileged
memory region using the following path:
Path: SMRAM DRAM DRAM-controller Cache-controller CPU PCI-bus-0 USB-controller
Solution (evidence): "USB-controller" cannot write the "SMRAM" privileged memory region.

Evidence description:

===========

Evidence 2

Possible attack: "SPI-controller" writes the "SMRAM" privileged memory
region using the following path:
Path: SMRAM DRAM DRAM-controller Cache-controller CPU PCI-bus-0 SPI-controller
Solution (evidence): "SPI-controller" cannot write the "SMRAM" privileged memory region.

Evidence description:
```

Figure 5.14: Sample Assurance Case that is the output of the ACBuilder analysis.

The output starts with a header that references which sub-goal of the Assurance Case pattern is being analyzed. It includes the CLIPS rule file name and the assurance case ID for the sub-goal. The next lines contain the possible vulnerabilities that were found by the rule, and which conditions ("solution") are necessary for it not to occur. The last field, "Evidence description: " is a placeholder for the evidence description. It may contain logical arguments, references to the architecture manual, or any other type of evidence.

Step 5 - Collecting evidence

The last step is to manually analyze the resulting assurance case, and collect the necessary evidence to support the goals and sub-goals. After all evidence is collected, the assurance

case for the target system is complete. Eventually there will be a lack of evidence to support all the sub-goals. This means that the architecture may be vulnerable to the corresponding attacks. For our example, the first required evidence can be supported by a fact that is found in the architecture manual, as shown in Figure 5.15.

```
Evidence 1

Possible attack: "USB-controller" writes the "SMRAM" privileged
memory region using the following path:
Path: SMRAM DRAM DRAM-controller Cache-controller CPU PCI-bus-0 USB-controller
Solution (evidence): "USB-controller" cannot write the "SMRAM" privileged memory region.

Evidence description:

Document: core-i7-800-i5-700-desktop-datasheet-vol-2, p 289

"5.5.5 SMM Space Decode and Transaction Handling
Only the processor is allowed to access SMM space. PCI Express and DMI Interface originated
transactions are not allowed to SMM space."
```

Figure 5.15: Sample Assurance Case evidence that has to be provided by the security analyst.

5.3 Methodology validation

In order to test the effectiveness of the methodology and the ACBuilder software, we ran our analysis methodology using the ACBuilder tool on an Intel Core i7-800 platform. We chose this CPU because we knew beforehand some vulnerabilities that affected this platform, and would be able to test whether our methodology would be able to find them.

Following the ACBuilder workflow, we initially described the architecture using the yEd software, resulting in the diagram depicted in Figure 5.16. We then applied the Assurance Case pattern from Figure 5.7, which incorporated the following analysis rules:

- MEM_READ/MEM_WRITE: checks whether a bus device can read/write to privileged memory regions.
- CACHE: checks whether privileged memory regions retains its access control policy even when stored in cache memory.
- MEM_REMAP: checks whether privileged memory cannot be remapped into non-privileged memory regions.

Table 5.2 summarizes how each of the known vulnerabilities were discovered by each of these rules.

Our tests showed that the system would be able to guide a security analyst towards the detection of most of the studied known vulnerabilities. In particular, it was most effective for HW-only architectural vulnerabilities.

We can now list the following strengths and weaknesses perceived in our system. They regard specifically to our proposed workflow and the use of the ACBuilder software:

Vulnerability	**Type**	**Associated rule**	**Effectiveness**
Code execution in SMM mode from cache [4, 49]	HW	CACHE	Detects
Q35 memory remapping: code execution in SMM mode and Xen hypervisor memory writing [54]	HW-based	MEM_REMAP	Detects
Intel Management Engine host memory writing [51]	HW	CACHE, MEM_READ and MEM_WRITE	Detects
MSI attacks: SIPI interrupt triggering, syscall injection and #AC exception injection [16]	HW/SW	-	-
SINIT code execution hijacking [55]	SW	-	-
PXE preboot code execution [56]	SW	-	-

Table 5.2: How the sample rules detect the known vulnerabilities.

- Strengths:
  - using previous knowledge from known vulnerabilities, it is able to correctly identify similar vulnerabilities;
  - the hardware architecture description is relatively fast and simple, using the adapted UML diagrams;
  - most of the hardware description and rules can be reused for similar architectures.
- Weaknesses:
  - difficulty in modeling some of the architectural features, such as the Intel TXT or the PCI bus protocol (e.g., required for the MSI attacks);
  - lack of a custom graphical interface for editing the HW architecture and assurance case template; the user has to remember or manually check for all keywords.

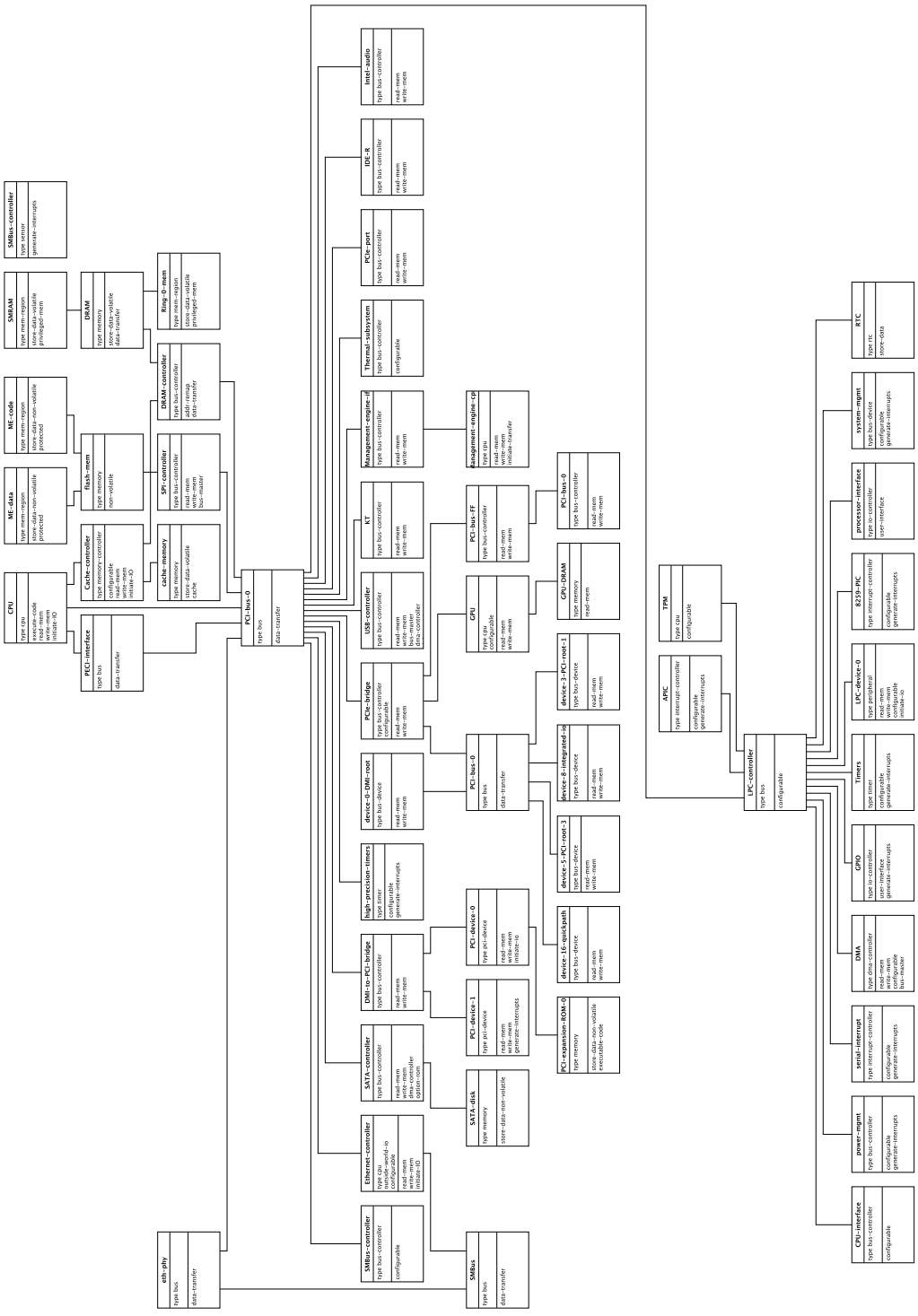


Figure 5.16: Architecture diagram illustrating the topology and complexity of the Core i7-800 hardware model.

Chapter 6

Conclusions

In this work we studied the hardware security analysis problem, together with various types of hardware vulnerabilities. Hardware attacks were presented, so that the associated vulnerabilities could be better understood.

We discovered that some of the *hardware attacks* were in fact originated from *software* problems that allowed standard hardware features to be used in favor of the attackers.

One of the causes of this misnaming problem might be the general lack of existing literature about hardware attacks and hardware security analysis. During our work, we could only find a few papers and web pages that effectively dove into the specifics of a real hardware attack. The most popular vulnerability publications, such as the CVE, only gave a brief description of the attack, in a way that it was not possible to pinpoint the real cause of the attack (hardware vs software).

The effects of wrongly classifying a software attack as a hardware attack can be many. First of all, there is a possible unfair damage to the public image of the affected hardware devices and vendors. Secondly, users and developers might be led to believe that every platform that uses the affected hardware is vulnerable to the attack. And worst of all, they might believe that systems with other types of hardware will not be affected by this vulnerability, when in fact they will.

We developed a taxonomy which allows for hardware vulnerabilities classification according to their type, and also for the identification of which design or production phase they were originated. Thus, this taxonomy can be used to correctly attribute the responsibility for a given vulnerability, either to hardware designers, IC manufacturers, software developers or other relevant entities. Our taxonomy also provided a clear separation of real hardware vulnerabilities from the hardware-involved vulnerabilities that originate from software problems, which are caused by the violation of security premises of the hardware.

Focusing on a specific type of vulnerabilities, i.e., architectural, we developed a methodology for the analysis of hardware architecture during the design phase, using the Assurance Case methodology. In this methodology, the security of the system is assured by a set of security goals and supporting evidence, all in a well-defined document structure.

We were able to build Assurance Cases for real-world hardware system architectures. In our examples we showed how three known vulnerabilities could be detected during the analysis process. We also found other possible unknown vulnerabilities that may affect

the same subsystem. Besides allowing for a deep analysis of the target architecture, there were also the benefits of guiding the analysis process and providing documentation that can be shared and peer-reviewed, which are two major drawbacks of the standard security evaluation process.

Lastly, we developed a tool to facilitate and automate part of the proposed security analysis methodology. Our initial tests show that the tool would be able to guide a security analyst towards the detection of most of the studied known vulnerabilities. In particular, it was most effective for architectural vulnerabilities, as expected.

For future work, the methodology could be applied and tested on a wider range of hardware architectures (e.g. ARM-based systems), and be fine-tuned to accommodate other types of hardware components and technologies. Examples of such technologies are Intel's SGX [57], TXT [58], and ARM's TrustZone [59]. Capturing the security aspects of components such as TPMs or smartcards can also be challenging, since they have a large number of security-related features that were not present in the systems that we analyzed.

The methodology can also be improved to capture the description of hardware attributes from VHDL or Verilog source code, as well as capturing hardware components connections from circuit schematics. The number and variety of ACBuilder's expert system rules can also be expanded, allowing it to also detect hardware aspects that could give rise to hardware-involved vulnerabilities (i.e. software attacks that leverage standard hardware features), which constitute the majority of hardware-related attacks.

Bibliography

[1] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes (Order Number: 325462-043US)*, ch. 33. May 2012.

[2] N. Potlapally, "Hardware security in practice: Challenges and opportunities," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pp. 93–98, IEEE, 2011.

[3] M. L. King, "Practical security validation," in *Microprocessor Test and Verification (MTV), 2013 14th International Workshop on*, pp. 35–38, IEEE, 2013.

[4] L. Duflot, O. Levillain, B. Morin, and O. Grumelard, "Getting into the smram: Smm reloaded," *CanSecWest*, 2009.

[5] R. Gallo, H. Kawakami, and R. Dahab, "Fortuna—a framework for the design and development of hardware-based secure systems," vol. 86, pp. pages 2063–2076, Elsevier, 2013.

[6] H. Mouratidis and P. Giorgini, "Secure tropos: a security-oriented extension of the tropos methodology," vol. 17, pp. pages 285–309, World Scientific, 2007.

[7] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos, "Tropos: An agent-oriented software development methodology," *Autonomous Agents and Multi-Agent Systems*, vol. 8, no. 3, pp. pages 203–236, 2004.

[8] Jasper Design Automation, "Jasper Design Automation - Security Path Verification App," September 2015.

[9] J. Rushby, "Mechanized support for assurance case argumentation," in *Proceedings of the 1st International Workshop on Argument for Agreement and Assurance. Springer*, 2013.

[10] S. Owre, J. M. Rushby, and N. Shankar, "Pvs: A prototype verification system," in *Automated Deduction—CADE-11*, pp. pages 748–752, Springer, 1992.

[11] E. Denney, G. Pai, and J. Pohl, "Advocate: An assurance case automation toolset," in *Computer Safety, Reliability, and Security*, pp. 8–21, Springer, 2012.

[12] E. Denney and S. Trac, "A software safety certification tool for automatically generated guidance, navigation and control code," in *Aerospace Conference, 2008 IEEE*, pp. 1–11, IEEE, 2008.

[13] E. Denney and G. Pai, "A formal basis for safety case patterns," in *Computer Safety, Reliability, and Security*, pp. 21–32, Springer, 2013.

[14] H. Kawakami, R. Gallo, R. Dahab, and E. Nascimento, "Hardware security evaluation using assurance case models," in *International Conference on Availability, Reliability and Security (ARES)*, IEEE, 2015.

[15] L. Duflot, D. Etiemble, and O. Grumelard, "Using cpu system management mode to circumvent operating system security functions," *CanSecWest*, 2006.

[16] R. Wojtczuk and J. Rutkowska, "Following the white rabbit: Software attacks against intel vt-d," 2011.

[17] Intel, *Intel Virtualization Technology for Directed I/O (Order Number: D51397-007, Rev. 2.3)*. October 2014.

[18] Intel, *Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes (Order Number: 325462-043US)*, ch. 10. May 2012.

[19] L. Duflot and L. Absil, "Programmed i/o accesses: a threat to virtual machine monitors," in *PacSec Applied Security Conference*, 2007.

[20] R. R. Collins, "The pentium foof bug," *Dr. Dobb's journal*, vol. 23, no. 5, 1998.

[21] T. S. Ankrum and A. H. Kromholz, "Structured assurance cases: Three common standards," in *High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on*, pp. 99–108, IEEE, 2005.

[22] R. Bloomfield, P. Bishop, C. Jones, and P. Froome, "Ascad—adelard safety case development manual," 1998.

[23] A. S. Ewen Denney, Ganesh Pai, "Advocate: An assurance case automation toolset (tool demonstration)," ICSE, 2013. `http://ti.arc.nasa.gov/m/profile/edenney/papers/sassur2012.pdf`.

[24] NATO, "Nato aep-67 - engineering for system assurance nato in programmes." `nsa.nato.int/nsa/zpublic/ap/aep-67`.

[25] T. Kelly and R. Weaver, "The goal structuring notation–a safety argument notation," in *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, Citeseer, 2004.

[26] J. C. Giarratano *et al.*, "Clips user's guide," 1993.

[27] A. R. A. Grégio, *Malware Behavior*. PhD thesis, University of Campinas (UNICAMP), 2012.

[28] T. Aslam, I. Krsul, and E. H. Spafford, "Use of a taxonomy of security faults," in *Proc. 19th NIST-NCSC National Information Systems Security Conference*, pp. 551–560, 1996.

[29] I. V. Krsul, *Software vulnerability analysis*. PhD thesis, West Lafayette, IN, USA, 1998.

[30] C. E. Landwehr, A. R. Bull, J. P. Mcdermott, and W. S. Choi, "A taxonomy of computer program security flaws," vol. 26, (New York, NY, USA), pp. pages 211–254, ACM Press, Sept. 1994.

[31] U. Lindqvist and E. Jonsson, "How to systematically classify computer security intrusions," in *Security and Privacy, 1997. Proceedings., 1997 IEEE Symposium on*, pp. 154 –163, may 1997.

[32] S. Hansman and R. Hunt, "A taxonomy of network and computer attacks," vol. 24, pp. pages 31–43, Elsevier, 2005.

[33] J. Howard and T. Longstaff, "A common language for computer security incidents," 1998.

[34] E. G. Amoroso, *Fundamentals of computer security technology*. 1994.

[35] M. Bishop, "Vulnerabilities analysis," in *Proceedings of the Recent Advances in intrusion Detection*, pp. 125–136, 1999.

[36] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens, "Transaction security system," vol. 30, pp. 206–229, IBM, 1991.

[37] J. Forristal, "Hardware involved software attacks,"

[38] J. Jürjens, "Umlsec: Extending uml for secure systems development," in *UML 2002—The Unified Modeling Language*, pp. 412–425, Springer, 2002.

[39] P. Shabalin and J. Jürjens, "Towards tool support for umlsec," 2003.

[40] G. Martin, L. Lavagno, and J. Louis-Guerin, "Embedded uml: a merger of real-time uml and co-design," in *Proceedings of the ninth international symposium on Hardware/software codesign*, pp. 23–28, ACM, 2001.

[41] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[42] S. Taha, A. Radermacher, S. Gerard, and J.-L. Dekeyser, "An open framework for detailed hardware modeling," in *Industrial Embedded Systems, 2007. SIES'07. International Symposium on*, pp. 118–125, IEEE, 2007.

[43] R. Kassem, M. Briday, J.-L. Béchennec, G. Savaton, and Y. Trinquet, "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation," vol. 58, pp. pages 318–337, Elsevier, 2012.

[44] R. Gallo, H. Kawakami, and R. Dahab, "FORTUNA - A Probabilistic Framework for Early Design Stages of Hardware-Based Secure Systems," in *Proceedings of the 5th International Conference on Network and System Security (NSS 2011)*, IEEE, 2011.

[45] T. Rhodes, F. Boland, E. Fong, and M. Kass, "Software assurance using structured assurance case models," vol. 115, p. 209, 2010.

[46] M. A. Cusumano and R. W. Selby, "How microsoft builds software," *Communications of the ACM*, vol. 40, no. 6, pp. 53–61, 1997.

[47] Sigasi, "Learning from LEON3/GRLIB," September 2015.

[48] Y. Matsuno, "D-case editor: A typed assurance case editor," *University of Tokyo*, 2011.

[49] R. Wojtczuk and J. Rutkowska, "Attacking smm memory via intel cpu cache poisoning," 2009.

[50] J. Rutkowska, "A quest to the core," *Blackhat Briefings USA*, 2009.

[51] A. Tereshkin and R. Wojtczuk, "Introducing ring -3 rootkits," in *Blackhat Briefings USA*, 2009.

[52] Y. Matsuno, H. Takamura, and Y. Ishikawa, "A dependability case editor with pattern library," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pp. 170–171, IEEE, 2010.

[53] Y. M. Shuichiro Yamamoto, "An evaluation of argument patterns to reduce pitfalls of applying assurance case," ICSE, 2013.

[54] J. Rutkowska and R. Wojtczuk, "Preventing and detecting xen hypervisor subversions," in *Blackhat Briefings USA*, 2008.

[55] R. Wojtczuk and J. Rutkowska, "Attacking intel txt via sinit code execution hijacking," 2011.

[56] M. Weeks, "Network nightmare: Ruling the nightlife between shutdown and boot with pxesploit," in *Defcon*, 2011.

[57] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, pp. 1–1, ACM, 2013.

[58] J. Greene, "Intel trusted execution technology," 2012.

[59] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security," *ARM white paper*, vol. 3, no. 4, pp. 18–24, 2004.