Security in Embedded Hardware

Daniel Ziener
Computer Architecture for Embedded Systems
University of Twente
Email: d.m.ziener@utwente.nl

5. Februar 2019

Contents

1 Motivation

Since the invention of the transistor, the complexity of integrated circuits continues to grow rapidly. First, only basic functions like discrete logic gates were implemented as integrated circuits. With improvements in chip manufacturing, the size of the transistors was drastically reduced and the maximum size of a die was increased. Now, it is possible to integrate more then one billion transistors [Xil03] on one chip.

In the beginning, electric circuits (e.g., a central processing unit) consisted of discrete electronic devices which were integrated on *printed circuit boards* (PCBs) and consumed a lot of power. The invention of integrated circuits in the end of the 1950s laid also the cornerstone of the development of embedded systems. For the first time, the circuits were small enough and consumed less power, so that applications embedded into a device, like production machines or consumer products became possible. An embedded system is considered as a complete special purpose computer that may consist of one or more CPUs, memories, a bus structure and special purpose cores.

The first integrated circuits were able to integrate basic logic functions (e.g., AND-, OR-gate) and flip-flops. With further integration, complex circuits, like processors, could be implemented into one chip. Today, it is possible to integrate a whole system with processors, buses, memories and specific hardware cores on a single chip, a so called *system-on-chip* (SoC).

These small, power and cost efficient, but manifolded applicable embedded systems finally took off on their triumphal course. Today, embedded systems are included in most electrical devices, from the coffee machine over stereo systems to washing machines. The application field of embedded systems spans from consumer products, like mobile phones or television sets, over safety critical applications, like automotive or nuclear plant applications, to security applications, such as smart cards or identity cards.

As integration density grew, problems with heat dissipation arose. The embedding of electronics into systems with small place and reduced cooling possibility, or the operation in areas with extreme temperature, intensify this problem. Furthermore, an embedded system which is integrated into an environment with moving parts is exposed to shock. Thermic and shock problems have a high influence on the reliability of the system. On the other hand, a system that steers big machines or controls a dangerous process must have a high operational reliability. These are all reasons

that design for reliability is gaining more and more influence on the development of embedded systems.

However, what is the need for reliability, if everyone may alter critical parameters or shut down important functions? To solve these problems, we need access control to the embedded system. But, today, embedded systems are also used to grant access to other systems or buildings. One example are chip cards. Inside these cards, a secret key is stored. It is important that no unauthorized persons or systems are able to read this secret key. Thus, an embedded system should not only be reliable but also secure.

Integration of functions for the guarantee of reliability and security features increases also the complexity of the integrated system enormously and thus design time. On the other hand, the market requires shorter product cycles. The only solution is to reuse cores, which have been designed for other projects or were purchased from other companies. The number of reused cores constantly increases. The advantages of IP core (Intellectual Property cores) reuse are substantial. For example, they offer a modular concept and fast development cycles.

IP cores are licensed and distributed like software. One problem of the IP cores distribution, however, is the lack of protection against unlicensed usage, as cores can be easily copied. Future embedded systems should also be able to prevent the usage of unlicensed cores or the core developers should be able to detect their cores inside an embedded system from third party manufactures.

Considering todays embedded systems, the integration of reliability and security increasing functions depends on the application field. In the area of security-critical systems (e.g., chip cards, access systems, etc.), several security functions are implemented. We find additional reliability functions in systems where human life or valuable assets are at stake (e.g., power plants, banking mainframes, airplanes, etc.). On the other hand, the problem of all these additional functions is the requirement for additional chip area. For cost-sensitive products which are produced in huge volumes, like mobile phones or chip cards, the developer must rethink to integrate such additional functions.

1.1 Security

Security becomes more and more important for computers and embedded systems. With the ongoing integration of personal computers and embedded systems into networks and finally into the Internet, security attacks on these systems arose. These networked, distributed devices may now compute sensitive data from the whole world and the attacker does not need to be physically present. Also, the increased complexity of these devices increases the probability of errors which can be used to break into a system. Figure 1.1 shows a classification of different types of attacks related to computer systems. This information is obtained form the CSI Computer Crime and

Security Survey [Ric08], where 522 US-companies reported their experience with computer crime. Further, the integration of networking interfaces into embedded devices, for which it would not be obviously necessary lead to strange attacks, for example that someone can break into the coffee machine over the Internet and alter the composition of the coffee [Wri08].

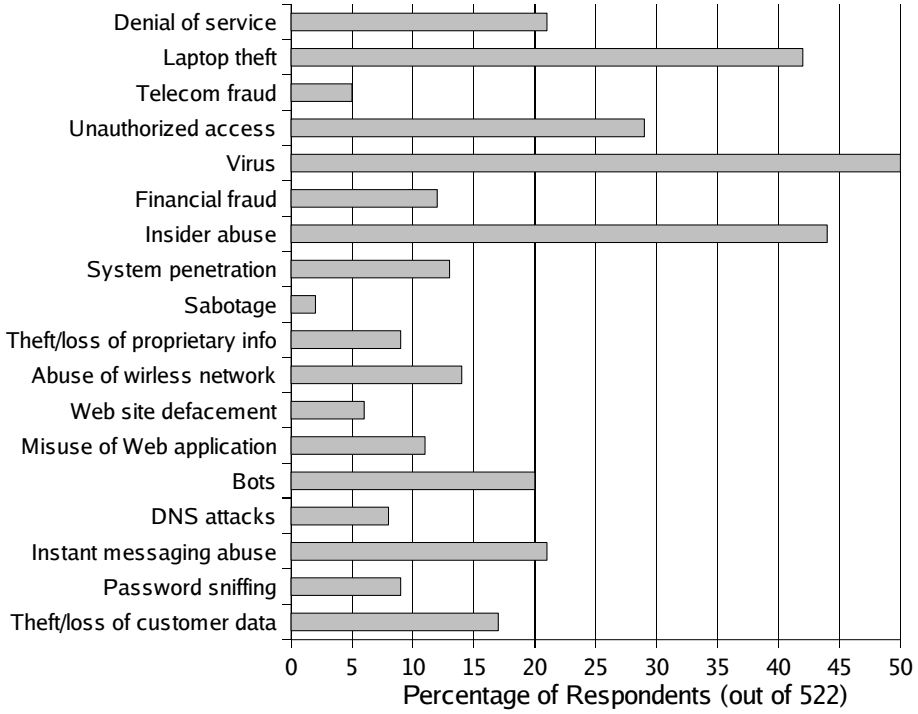


Figure 1.1: Security attacks reported in the CSI Computer Crime and Security Survey [Ric08], where 522 US-companies reported their experience with computer crime for the year 2008.

Within the last decade, the focus of the embedded software community paid more attention onto security of software-based applications. Today, most of the software updates fix security bugs and provide only little additional functionality. At the same time, the number of embedded electronic devices including at least one processor is increasing.

The awareness of security in digital systems lead to investigation of secure communication standards, for example SSL (Secure Socket Layer) [FKK96], the implementation of cryptographic methods, for example AES (Advanced Encryption Standard) [Fed01], a better review of software code to find vulnerabilities, and the integration of security measures into hardware. Nevertheless, Figure 1.2 shows that the vulnerability of digital systems increased rapidly over the last years. The main cause for vulnerability are software errors through which a system may be compro-

mised. The software of embedded systems moves from monolithic software towards module-based software organized and scheduled by an operating system. By means of modern communication structures like the Internet, the software on embedded systems may be updated, partially or completely. These update mechanisms and the different communication possibilities open the door for software based attacks on the embedded system. For example, the number of viruses and trojans on mobile phones increased rapidly over the last years. One main gateway for these attacks are buffer overflows. A wrong jump destination or a wrong return address from a subroutine might cause an execution of infiltrated code (see also Section 3.1).

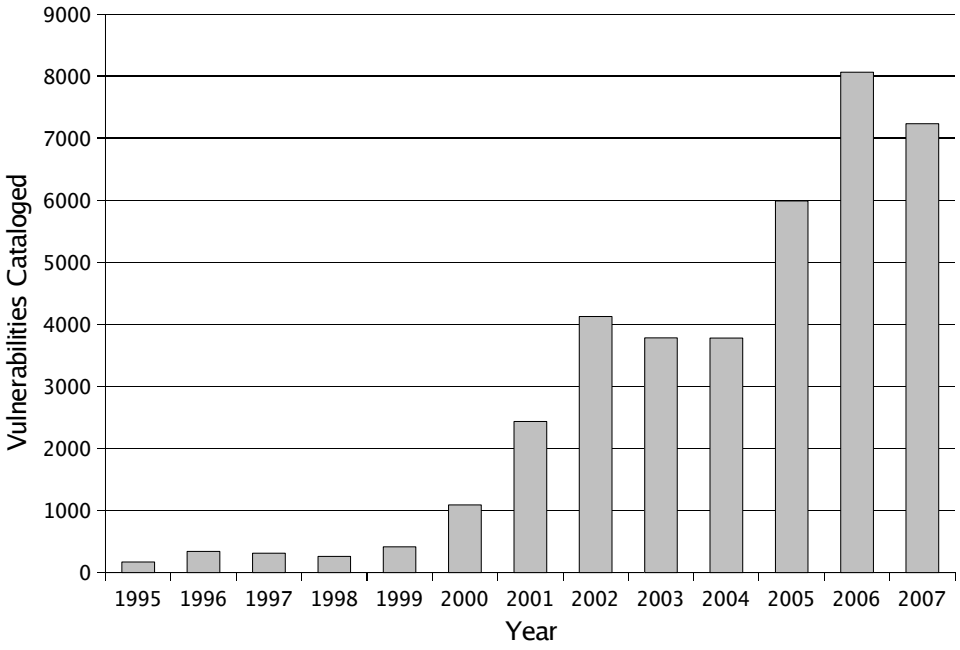


Figure 1.2: Vulnerability of digital systems reported to US-CERT between 1995 and 2007 [US-08].

However, also hardware errors can lead to the vulnerability of a system. For example, Kaspersky shows that it is possible that the execution of appropriate instruction sequences on a certain processor can lead to an adoption of control of the system by an attacker [KC08]. In this case, it does not matter which operation system or software security programs are running on the system.

A common objective for attackers are sensitive data, which are stored inside a digital system. To reach this objective, attackers are not only bound to software attacks. Hardware attacks, where the digital system is physically penetrated to gather information over the security facilities, or extract sensitive information are also practical. If an embedded device stores secure data, like a cryptographic key, attackers may try

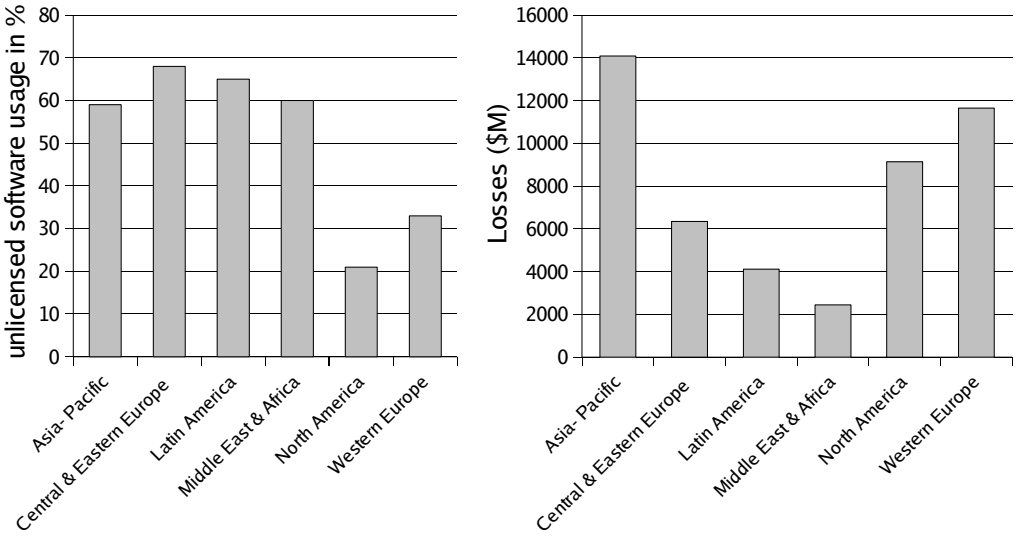


Figure 1.3: On the left side, the percentage of the usage of unlicensed software is shown in different areas of the world. On the right side the corresponding losses in million US-Dollars are depicted [All07].

to read out this secret data by physically manipulating the processor on the embedded device. This may be done by differential fault analysis (DFA) [BS97] or by specific local manipulation on control registers inside the processor (see also Section 3.2). The attackers goal thereby is to execute infiltrated code or deactivate the protection of the secured data which may result from the manipulation of the program counter.

Another relevant security aspect in embedded systems is *intellectual property protection* (IPP). Due to shorter design cycles, many products can only be developed with acquired hardware cores or software modules. Those companies selling these cores and modules naturally have a high interest in securing their products against unlicensed usage. Figure 1.3 shows the estimated percentage of unlicensed software used in different areas of the world. Also, calculated revenue losses are shown. Additionally, many unlicensed hardware IP cores are used in products. At the RSA conference in 1998, it was estimated, that the adversity of the usage of unlicensed IP cores approaches 1 Billion US$ per day [All00].

1.2 FPGAs

FPGAs (Field Programmable Gate Arrays) have their roots in the area of *PLDs* (Programmable Logic Devices), such as *PLAs* (Programmable Logic Arrays) or *PALs* (Programmable Array Logics). Today, FPGAs have a significant market segment in the microelectronics and, particularly in the embedded system area. The advantages

of FPGAs over ASICs are their flexibility, the reduced development costs, and the short implementation time. Also, developers have a limited implementation risk, a), because of the easy possibility to update an erroneous design and b), because of the awareness, that the silicon devices are proofed and the underlying technology operates correctly under the specified terms.

The main advantage of FPGAs is their reconfigurability. The demand for flexibility through reconfigurability will rise according to ITRS [ITR07] from 28% of all functionalities in 2007 until to an estimated 68% in the year 2022. Note that ITRS also takes into account software running on a microprocessor which can be updated. Furthermore, many FPGA devices support dynamic partial reconfiguration, which means that during runtime, the design or a part of it can be reconfigured. With this advantage, we can envisage new designs with new and improved possibilities and properties, like an adaptive design, which can adapt itself to a new operation environment. Unfortunately, dynamic reconfiguration is currently used rarely due to the lack of improved design tools which increases the development costs for dynamic reconfiguration. But now, improved design tools for partial reconfiguration are starting to become available, like the ReCoBus-Builder [KBT08, KHT08] or Xilinx Planahead [DSG05]. Nevertheless, dynamic reconfiguration for industrial designs is in its infancy, and it will take several years to use all the great features of FPGAs.

In the last years, the main application area of FPGAs were in small volume embedded systems and rapid prototyping platforms, where ASIC designs can be implemented and verified before the expensive masks are produced. Nevertheless, the FPGA usage in higher volume market rises, mainly due to lower FPGA price, higher logic density and lower power consumption. Furthermore, due to shorter time-to-market cycles and rising ASIC costs, FPGAs are breaking more and more into traditional ASIC domains. On the other hand, FPGAs are becoming competitors in the (reconfigurable) DSP domain with multi-core and coarse-grain reconfigurable architectures, as well as from graphic processing units (GPU) where DSP algorithms are adapted to run on these architectures. Nevertheless, these architectures suffer from the lack of flexibility and today, only FPGA technology is flexible enough to implement a heterogeneous reconfigurable system-on-a-chip.

1.3 ASICs

Besides the advantages and the success of FPGAs, there still exists a huge market for traditional *ASICs* (Application Specific Integrated Circuit). ASICs are designed for high volume productions, where small cost-per-unit is important, as well as in low power and high performance applications and designs with a high logic density.

The implementation of a core on an ASIC instead of an FPGA (both 90 *nm* technology) may require 40 times less area, may speed up the critical path by a factor between 3 and 4, and may reduce the power by a factor of about 12 [KR06]. Here,

we see that the big advantage of ASICs over FPGAs is the higher logic density, which results in significantly lower production cost per unit. The disadvantages of ASICs are the higher development and the higher initial production costs (e.g., masks, package design, test development [Kot06]). Therefore, the decision for using ASICs or FPGAs due to minimization of the total costs is highly dependent on the production volume. Figure 1.4 shows a comparison of the total costs between ASICs and FPGAs in different technology generations over the production volume. The ASIC graphs start with higher costs due to the high initial production costs, but with a lower slope due to cheap production costs per unit. The initial cost of ASICs increases from technology generation to generation, mainly because of the increasing chip and technology complexity and logic density. FPGA designs have lower initial costs, but higher costs per unit. In summary, the total costs of a design using FPGA technology is lower until reaching a certain production volume point. However, according to Xilinx [RBD+01] this point is shifting for each technology generation in the direction of higher volumes.

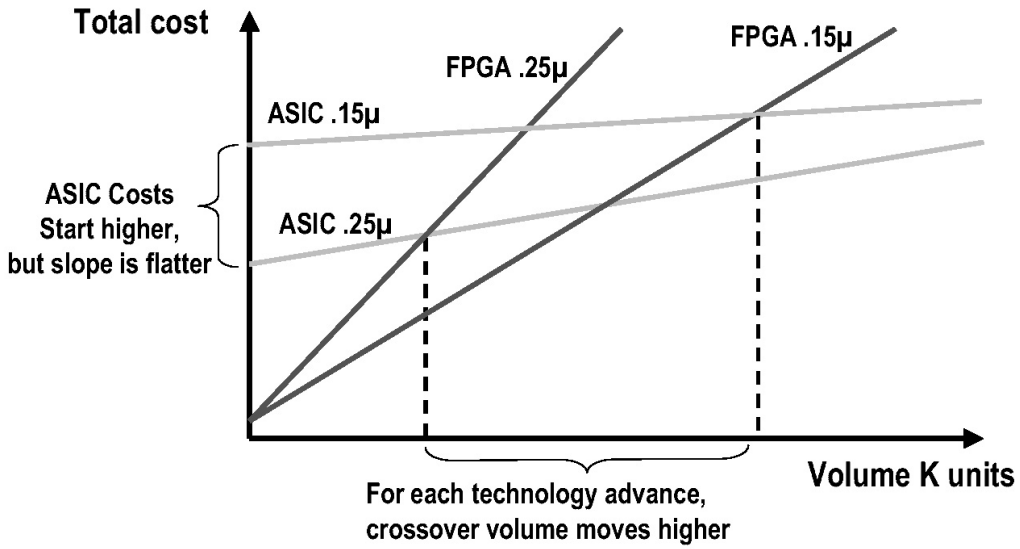


Figure 1.4: This figure from [RBD+01] shows a comparison of the total costs of FPGAs and ASICs in different technology generations over the production volume. With every new technology generation, the break even point between the total costs of FPGAs and ASICs designs is shifted more and more to the ASIC side. As on implication, one may expect the market for FPGAs to grow.

Nevertheless, besides the total costs discussion, there exist many design solutions, especially in the area of embedded systems, which can only be implemented using

ASIC technology. Examples include very low power designs and high performance designs.

Before summarizing the major contributions of the thesis with respect to the above topic, a set of definitions is in order.

2 Definitions

In this section, we introduce necessary definitions of terms with respect to security and reliability of embedded systems that will be throughout this thesis. First, definitions in the field of dependability and the difference between defects, faults, and errors are outlined. After the categorization of faults and errors, definitions stemming from the area of security attacks are presented. Finally, different types of overhead, which are indispensable for additional security and reliability functions, are described.

2.1 Dependability and its Attributes

The dependability of a system is defined by the IFIP 10.4 Working Group on Dependable Computing and Fault Tolerance as: *"... the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers ..."* [IFI90]. According to Laprie and others [ALR01], the concept of dependability consists of three parts: the threats to, the attributes of, and the means by which dependability is attained (see Figure 2.1).

The attributes of dependability are a way to assess the trustworthiness of a system. The attributes are: *availability*, *reliability*, *safety*, *confidentially*, *integrity*, and *maintainability*.

2.1.1 Availability

The availability is considered as the readiness for correct service [ALR01]. This means that the availability is a degree of the possibility to start a new function or task of the system. Usually, the availability is given in the percentage of time that a system is able of serving its intended function and can be calculated using the following formula:

$$Availability = \frac{Total\ Elapsed\ Time - Down\ Time}{Total\ Elapsed\ Time} \tag{2.1}$$

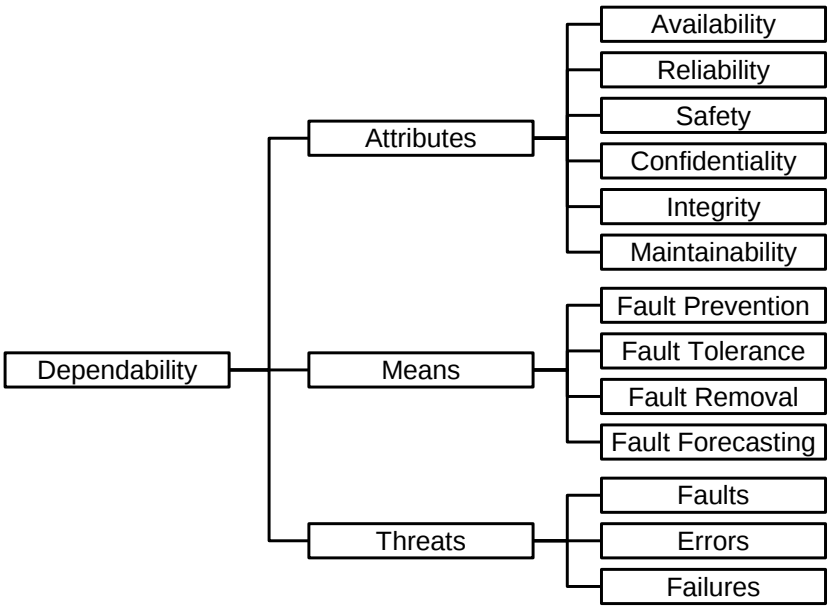


Figure 2.1: The relationship of dependability between attributes, threats and means [ALR01].

Availability is also often measured in *"nines"*. Two nines means an availability of 99%, three nines means 99.9% and so on. Table 2.1 shows the maximal downtime within a year for different availability values.

Availability	Percentage	8-hour day	24-hour day
Two nines	99%	29.22 hours	87.66 hours
Three nines	99.9%	2.922 hours	8.766 hours
Four nines	99.99%	17.53 mins	52.60 mins
Five nines	99.999%	1.753 mins	5.260 mins
Six nines	99.9999%	10.52 secs	31.56 secs

Table 2.1: The maximal annual downtime of a system for different values of availability, running either 8 hours or 24 hours per day [Rag06].

2.1.2 Reliability

Reliability is defined as the ability of a system or component to perform its required functions under well-defined conditions for a specified time period [Ger91]. Laprie and others transcribe the reliability with the continuity of correct service [ALR01]. Important parameters of reliability are the failure rate and its inversion, the *MTTF*

(mean time to failure). Other parameters, like the *MTBF* (mean time between failures) include the time which is necessary to repair the system. The MTBF is the sum of MTTF and the *MTTR* (mean time to repair).

2.1.3 Safety

Safety is the attribute of a safe system. This means that the system cannot lead to catastrophic consequences for the users or the environment. Safety is relative, the elimination of all possible risks is usually impossible. Furthermore, the safety of a system cannot be measured directly. It is rather a subjective confidence of the system. Whereas availability and reliability avoid all failures, safety avoids only the catastrophic failures, which are only a small subset.

2.1.4 Confidentially

The confidentially of a system describes the absence of unauthorized disclosure of information. The International Organization of Standardization (ISO) defines the confidentially as *"ensuring that information is accessible only to those authorized to have access"* [ISO05]. In many embedded systems (e.g., cryptographic systems), it is very important to secure the information (e.g., the secure key) stored inside the system against unauthorized access. But also the prevention of unlicensed usage of software programs or hardware cores are topics of confidentially. Confidentially is, like safety, subjective and cannot be measured directly.

2.1.5 Integrity

Integrity is the absence of improper system state alternation. This alternation can be an unauthorized access to alter system information inside the system, which are necessary for the correctness of the system. Furthermore, the system state alternation can also be a damage or modification of the system. System integrity assures that no part of the system (software or hardware) can be altered without the necessary privileges. Also, the IP core verification to ensure the correct creator and the absence of unauthorized supplementary changes can elevate the integrity of a system. Integrity is the precondition for availability, reliability and safety [ALR01].

2.1.6 Maintainability

Maintainability is the ability to undergo repairs and modifications. This can be done to repair errors, meet new requirements, make further maintenance easier, or to cope with a changed requirement or environment. A system with a high maintainability may have a good documentation, a modular structure, is parameterizable, uses assertions and implements built-in self tests.

2.1.7 Security

Security is defined as a combination of the attributes (1) *confidentially* (the prevention of the unauthorized disclosure of information), (2) *integrity* (the prevention of the unauthorized amendment or deletion of information), and (3) *availability* (the prevention of the unauthorized withholding of information) [ITS91]. An alternative definition for security could be the absence of unauthorized access to the system state [ALR01]. The prevention or detection of the usage of unlicensed software or IP cores can also be seen as a security aspect (confidentially) as well as the prevention of the unauthorized alteration of software or IP cores (integrity). Like safety, security shall prevent only a class of failures which are caused by unauthorized access or unauthorized handling of information.

2.2 Fault, Error, Failure

Faults, *errors*, and *failures* are the threats which affect the dependability (see Figure 2.1).

2.2.1 Failure

A system is typically composed of different components. Each component can be further subdivided into other components. All of these system components may have internal states. If a system delivers its intended function, then the system is working correctly. The intended function of a system can be described as an input/output or interface specification which defines the behavior of the system on the system boundaries with its users or other systems.

The system interface specification may not be complete. For example, it is specified that an event occurs on the output of the system, but the time of this event to occur is not exactly specified. So, the system behavior can vary without violating the specification. If the specification is violated, the system fails. A failure is an event which occurs when the system deviates from its interface specification (see Figure 2.2).

2.2.2 Errors

If the internal state of a component deviates from the specification (the specification of the states of the component), the component is erroneous and an error occurs. An error is an unintended internal state whereas a failure is an unintended interface behavior of the system. An error may lead to a failure. But it is also possible that an error occurs and does not lead to a system failure, because of the component is currently not used or the error is detected and corrected fast enough. Errors can be

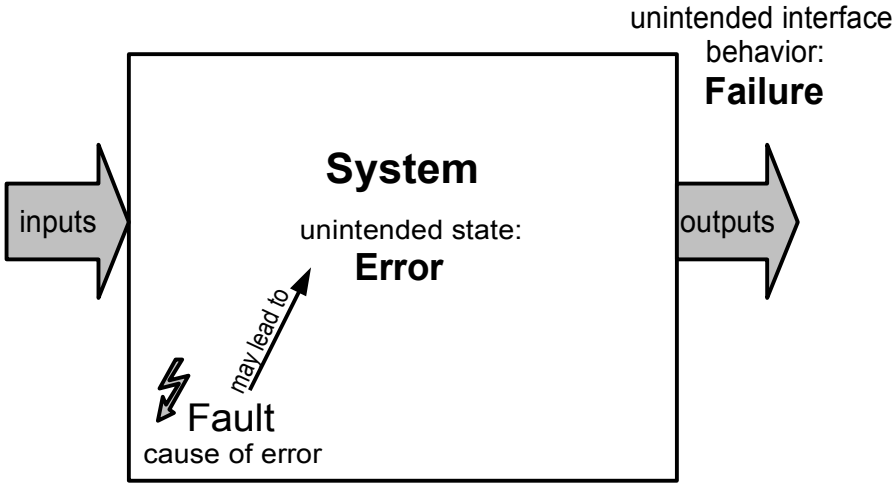


Figure 2.2: Faults may lead to an error, which may also lead to a system failure.

transient or permanent. Transient errors caused by transient faults usually occur in systems without feedback. In systems with feedback, an error might be permanent by affecting all following states. In this case, the error only disappears by a reset or by shut down of the system.

2.2.3 Faults

A fault is defined as a defect that has the potential to cause an error [Jal94]. All errors are caused by faults, but a fault may not lead to an error. In the latter case, the fault is masked out and has no impact on the system.

For example, consider the control path of a processor core. A fault like a single event transient fault, caused by an alpha particle impact, occurs on one signal of the program counter between two pipeline stages. If the time of occurrence is near the rising active clock edge, an error may occur. Otherwise, if the time of occurrence is far away form the rising edge of the clock, the fault does not lead to an error. The erroneous program counter value can now lead to a system failure, if the wrong subroutine is executed and the interface behavior differs from the specification. Otherwise, if an error detection technique, like a control flow checker, as introduced later in Chapter 4.7.3, is used, the error can be detected after the fault appearance, and the error may be corrected by a re-execution of the corresponding instruction. But, this additional re-execution needs several clock cycles to restore the error free state. For real-time systems with very critical timing requirements, the possible output events might be too late and the system thus might still fail.

2.3 Means to Attain Dependability

Means are ways to increase the dependability of a system. There exist four means, namely *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*.

2.3.1 Fault Prevention

Fault prevention deals with the question "How the occurrence or introduction of faults can be prevented?". Design fault might be prevented with quality control techniques during the development and manufacturing of the software and hardware of a system. Fault prevention is further closely related to maintainability. Transient faults, like single event effects, might be reduced by shielding, radiation hardening, or larger structure sizes. Attacks might be prevented by security measures, like firewalls or user authentication. To prevent the usage of unlicensed programs or IP cores, the code (source, binary, or netlist code) could be delivered encrypted and only the authorized customer has the right cryptographic key to decrypt the code. To prevent the impart of the key, techniques like dongles or an authentication with MAC-Address can be used.

2.3.2 Fault Tolerance

A fault-tolerant system does not fail, even if an erroneous state is reached. Fault tolerance enables a system to continue operation in the case of one or more errors. This is usually implemented by error detection and system recovery to an error-free state. In a fault tolerant system, errors may occur, but they must be handled correctly to prevent a system failure.

The first step towards a fault tolerant system is *error detection*. Error detection can be subdivided into two classes: *concurrent error detection* and *preemptive error detection* [ALR01]. Concurrent error detection takes place at runtime during the service delivery, whereas preemptive error detection runs in phases where the service delivery is suspended. Examples for concurrent error detection are error codes (e.g. parity or CRC), control flow checking, or razor flip-flops [ABMF04].

Also, redundancy belongs to this class of error detection. One may distinguish three types of redundancy: *hardware*, *time* and *information* redundancy. To detect errors with hardware redundancy, we need at least two units where both results are finally compared. If they divert, an error occurred. On time redundancy, the system executes the same inputs twice, and both results are compared after the second execution. Information redundancy uses additional information to detect errors (e.g., parity bits).

BISTs (Built In Self Tests) or start-up checks belong to the preemptive error detection class.

The next step is the recovery from the erroneous state. Recovery consists of two steps, namely *error handling* and *fault handling*. Error handling is usually accomplished by *rollback* or *rollforward*. Rollback is done by using error-free states which are stored on certain checkpoints to restore the state of the system to an older error-free state. Rollback is attended by delaying the operation. This might be a problem in case of real-time applications. Rollforward uses a new error-free state to recover the system.

If the cause of the error is a permanent or periodic temporal fault, we need fault handling to prevent the system from running into the same error state repeatedly. This is usually done by *fault diagnosis*, *fault isolation*, *system reconfiguration* and *system reinitialization*. For example, in case at permanent errors in memory structures, the faulty memory column is identified and this column is switched to a reserved spare column. After the switch over, the column content must be reinitialized.

It is important to notice that fault tolerance is a recursive concept. The techniques and methods which provide fault tolerance should obviously themselves be resistant against faults. This can, for example, be done by means of replication.

2.3.3 Fault Removal

During the development phase and during the operational runtime, fault removal might be performed. At the development phase, fault removal consists of the following steps: *verification*, *diagnostics*, and *correction* [ALR01]. This is usually done by debugging and/or simulation of software and hardware. For the verification of fault tolerant systems, fault injection can be used.

Fault removal during the operational runtime is usually done by *maintenance*. Here, faults can be removed by software updates or by the replacement of faulty system parts.

2.3.4 Fault Forecasting

Fault forecasting predicts feasible faults to prevent or avoid the fault or decrease the effect of the fault. This can be done by performing an evaluation of the system behavior with respect to fault occurrence and effects. Modeling and simulation of the system and faults are a common way to achieve this evaluation.

2.4 Security Flaws and Attacks

Faults affecting the security of a system are also called *flaws* [LBMC94]. In this work, the term flaw is used as a synonym of a fault, which leads to the degeneration of the security of a system. A flaw is therefore a weakness of the system which could be exploited to alter the system state (error). A threat is a potential event which might

lead to this alternation and therefore to a security failure. The process of exploiting the flaw by a threat is called an *attack* (see Figure 2.3). A security failure occurs when a security goal is violated. The main security goals are the dependability attributes *integrity*, *availability*, and *confidentially*. The difference between a flaw and a threat is that a flaw is a system characteristic, whereas a threat is an external event.

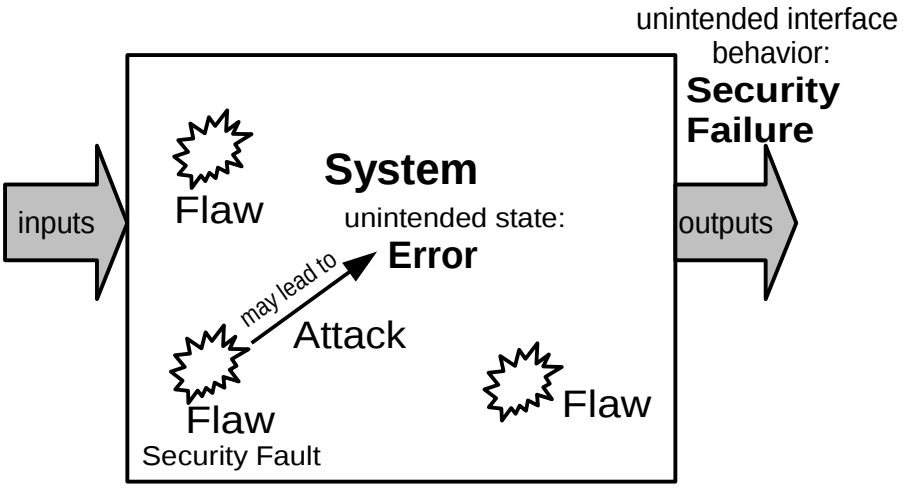


Figure 2.3: Flaws are security faults, which lead to errors if they are exploited by attacks. The state alternation in case of an attack may lead to a security failure.

A flaw can be *intentional* or *inadvertent*. Intentional flaws can further be *malicious* or *non-malicious*. An intentional malicious flaw is, for example, a trojan horse [And72]. An intentional non-malicious flaw could be a communication path in a computer system which is not intended as such by the system designer [LBMC94]. An inadvertent flaw could be, for example, a bug in a software program, which enables unauthorized persons with specific attacks to read protected data.

Like other faults, flaws can also be further categorized using the origin of the flaw and the persistence. The origin can be during the development (e.g., the developer implement a back door to the system), or during operation or maintenance. Usually, the flaws exist for a longer period of time (e.g., from the flaw arise until the flaw is disappeared by a security update). But also special flaws exists, which only appear on certain situations (e.g., the year 2000 problem; switching from year 1999 to 2000).

Attacks can be classified using the security goals or objective of the attack into *integrity*, *availability*, and *confidentially attacks*. Integrity attacks break into the system and change part or the whole system (software or hardware) or of the data. The goal of availability attacks is to make a part or the whole system unavailable for user

requests. Finally, the goal of confidentially attacks is to gather sensitive or protected data from the system.

Furthermore, if an attack is successful, new flaws can be generated as a result from the attack. For example, a flaw in software is exploited by a code injection attack (see Section 3) and the integrity of the system is injured by adding a malicious software routine. This software routine opens now intentional malicious flaws, which can be used by confidentially attacks to gather sensitive data.

To describe all attacks using this terminology is not easy. For example, a copyright infringement where someone unauthorized is copying an IP core. The result of the attack is a reversal of confidentially. Here, the sensitive data is the core itself. The erroneous state is the unauthorized copy of the IP core. But what is the flaw which makes the attack possible? Here, we must assume that the ability to easily copy an IP core is the flaw. This example teaches us that flaws exist even on reasonably secure systems and cannot be easily removed. On every system we must deal with flaws, which might affect the security as well as other faults which might affect the other areas of dependability.

2.5 Overhead

Methods for increasing security and reliability in embedded systems often have the drawback of additional overhead. To evaluate the additional costs of these methods, we can use the following criteria:

- *Area overhead* (hardware cost overhead),
- *Memory overhead*, and
- *Execution time overhead* (CPU time).

Analysis and quantification of the additional costs significantly depends on the given architecture and the implementation on a specific target technology.

2.5.1 Area Overhead

The straightforward method for measuring the area overhead of an additional core is to measure the chip area occupied by the core. Unfortunately, this method can only compare cores implemented in the same technology with exactly the same process (lateral dimensions). To become independent of this process, the *transistor count* may be used. However, information about the area of the signal routing is not included here. In most of the technologies and processes, signal routing requires little additional area because of the routing layers are above the transistors (in the third dimension). This also depends on the specific process.

The number of transistors, however, is a reasonable complexity indicator, only if the compared cores use the same technology (e.g., CMOS, bipolar). To compare the hardware costs of a core mapped onto different technologies, the *gate count* can be used. Here, the number of logical (two input) gates is used to describe the hardware costs. For comparing cores between ASIC and FPGA technologies, the count of *primitive components or operations*, like *n*-bit adders or multipliers, can be used.

Using primitive components or operations for the description of the overhead, one is independent of the underlying technology and process. Describing hardware cost in a higher hierarchy level, like primitive components or operations, however, is more inaccurate with respect to the real hardware costs than describing the overhead in chip area. The resulting chip area of the used primitive components depends highly on the technology and process and also on the knowledge of the chip designer or the quality of the tools.

2.5.2 Memory Overhead

The memory overhead for methods increasing the security and reliability can be measured by counting the additional *ram bits* used by the corresponding method. Memories embedded on the chip, so called internal memories, use hardware resources on the chip and so they contribute to the area overhead. Nevertheless, the content of memories can be relatively easily shifted to a cheaper external memory, for example an off-chip system DRAM. So, we decided to handle the memory overhead separately. It must be taken into account that internal memory has higher hardware costs at the same size, but a lower latency. External memory is usually cheaper, but it involves additional hardware costs on the chip, as for example a DRAM controller. If a DRAM with the corresponding controller already exists on the chip, these resources might be shared to reduce the hardware cost.

2.5.3 Execution Time Overhead

Some methods for increasing the security and reliability have additional latency. This means that the result of the protected core or software appears later on the outputs than on the unprotected one. For hardware cores, latency is usually counted in additional clock cycles. For software programs, latency can be expressed in additional instructions which must be executed by the processor or in additional execution time of the processor. For example, some existing methods for control flow checking [GRRV03] generate additional instructions that are inserted into the original program running on the processor which is monitored. This might cause a timing impact for the user program which impact can be measured by additional execution time of the processor. The execution time depends on the processor and the number of executed additional instructions.

Also, if no additional software is executed on the processor and the processor is enhanced with additional hardware, some methods can stall [ZT09] the processor pipeline, slow down the execution of the user program, or insert additional pipeline steps [Aus99] without executing additional instructions.

For processor architectures, the execution time overhead can be measured by counting the additional pipeline steps. If the processor architecture executes one instruction in one pipeline step (in the best case one clock cycle), the number of additional executed instructions are also given in the number of additional pipeline steps.

2.6 IP Cores and Design Flow

The reuse of IP cores is an important step to decrease the *productivity gap*, which emerges from the rapid increase of the chip complexity and the slower growth of the design productivity. Today, there is a huge market and repertoire of IP cores which can be seen in special aggregation web sites, for example [Reu] and [Est], which administrate IP core catalogs.

The delivery format of IP cores is closely related to the *design flow*. The design flow consists of different *design flow* or *abstraction levels* which transfer the description of the core from the level where the core is specified into the final implementation. The design flow is dependent from the target technology. The FPGA and the ASIC design flow look similar, however, there exist differences at some steps.

Figure 2.4 shows a general design flow for electronic designs with FPGA and ASIC target technologies. This design flow view can be embedded into a higher system model view for *hardware/software co-design*, for example the *double roof model* introduced by Teich [TH07]. The depicted design flow implements the logic synthesis and the following steps in the double roof model. Furthermore, the different abstraction levels are derived from the *Y-diagram*, introduced by Gaijski [GDWL92].

The different abstraction levels are the *register-transfer level*, the *logic level*, as well as the *device level*. Designs specified at the register-transfer level (RTL) are usually described in *Hardware Description Languages* (HDLs) like VHDL or Verilog, whereas designs at the logic level are usually represented in *netlists*, for example, *Electronic Design Interchange Format* (EDIF) netlists. At the device level, FPGA designs are implemented into bitfiles, while ASIC designs are usually represented by their chip layout description. The transformation of an HDL-model into an netlist-model is called logic *synthesis*, whereas the transformation of a netlist-model into a target depended circuit is called *implementation*. The implementation consists of the aggregation of the different netlist cores and subsequent *place and route* step. The *technology mapping* can be done in the synthesis or in the implementation step, or in both. For example, the Xilinx FPGA design flow maps the logic to device dependent *primitive cells* (LUT2, FF, etc.) in the synthesis step, whereas the mapping

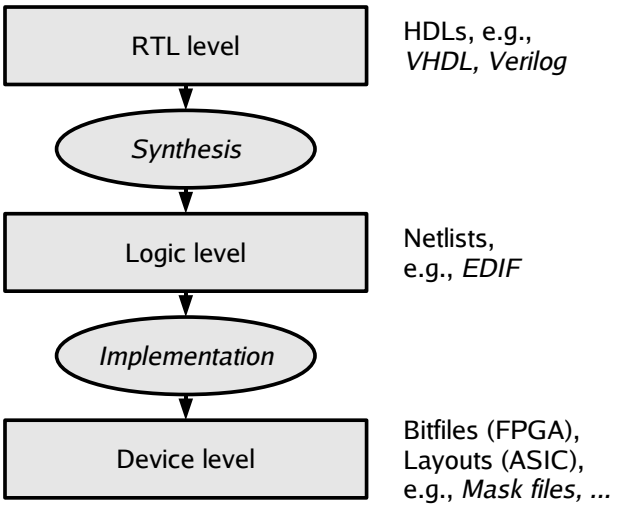


Figure 2.4: A general design flow for FPGA and ASIC designs with the synthesis and implementation steps and the different abstraction levels.

of these primitive cells to slices and *configurable logic blocks* (CLBs) is done in the implementation step [Xilb].

IP cores can be delivered at all different abstraction levels in the corresponding format: on the RTL as *VHDL* or *Verilog* code, on logic level as *EDIF* netlist, or on the device level as *mask files* for the ASIC flow or as FPGA depended (partial) *bitfiles*.

IP cores can be further categorized into *soft* and *hard cores*. Hard cores are ready to use and are offered into a target depended layout or bitfile. All IP cores which are delivered into an HDL or netlist format belongs to the soft cores. These cores need further transformations of the design flow to be usable. The advantages of soft cores are their flexibility for different target technologies and their can be parameterizable. However, the timing and the area overhead are less predictable compared to hard cores due the impact of the needed design tools. *Analog* or *mixed signal* IP cores are usually delivered as hard cores.

3 Attacks on Embedded Systems

There exist two ways for categorization of attacks. The first way is to categorize attacks by the violated security goals. The other way is to describe how the attack is realized and which way the attacker chose to compromise the system [Rag06, RRKH04].

Using the first categorization schema, the main security goals are *integrity*, *availability*, and *confidentially* (see Figure 3.1 above, and Section 2.4). Attacks which compromise integrity can be further subdivided into *manipulation of data*, *manipulation of software* or *IP cores*, as well as *forging of authorship*. Attacks which may paralyze a system compromise the availability. Attacks to compromise the confidentially of a system can be subdivided into *gathering of sensitive data* like passwords, keys, program code, or IP cores, and *getting access control* to a system. Additionally, *copyright infringement* compromises the confidentiality of the author of the core.

The means used to launch the attacks or the ways how the attack is realized can be categorized into *invasive* and *non-invasive attacks* (see Figure 3.1 below). Both groups can further be subdivided into *logical* and *physical attacks* [RRKH04]. Physical attacks typically require relatively expensive equipment and infrastructure, especially physical invasive attacks. Whereas for logical attacks, usually only a computer or the embedded system is needed.

3.1 Code Injection Attacks

Code injection assaults are attacks where the code integrity of a system is compromised. This can be the integrity of software as well the integrity of executed bitfiles in a reconfigurable system, such as FPGAs. The goals of code injection attacks are manifolded. The demolished integrity of further program code or sensitive data, the paralysis of the system as well as getting access to sensitive data are in the foreground of the attacker.

Code injection attacks bring the system under the control of the attacker. Programs inserted by the attacker, may easily read or alter the sensitive data and forward the data to interfaces where the data can be collected.

To gain control over a system, the attacker must first insert a routine, which performs the intended behavior, into memory. This routine may, for example, read out

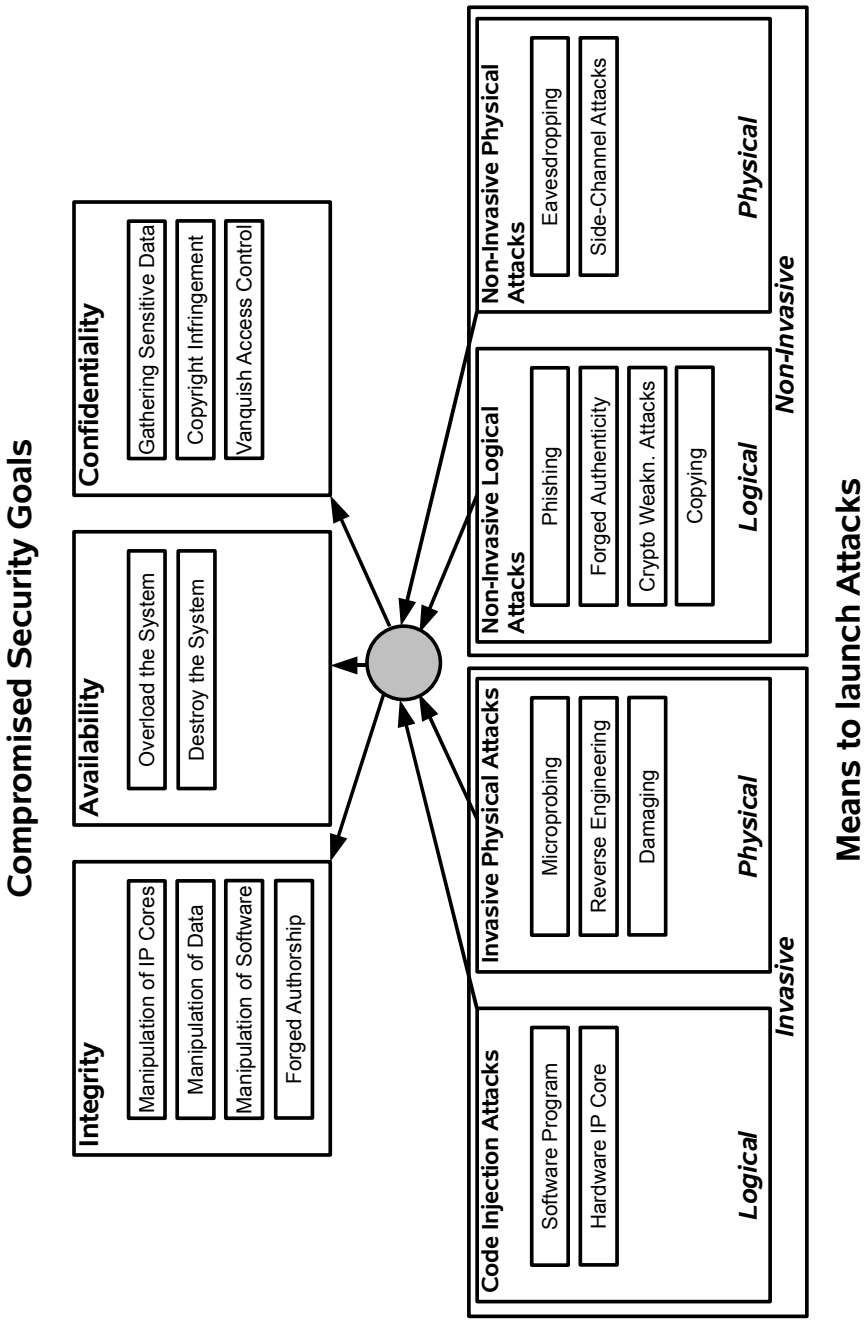


Figure 3.1: Attacks can be categorized with the compromised security goals or the attack goals (above) and with the means to launch the attack (below). The different means of attacks can invalidate different security goals.

secured data, deactivate security protection, open gateways for the attackers, or load another infiltrated code from the Internet. The malicious code can be inside the processed input data which is loaded into the memory by the processor. The second step is bringing the processor in a state to execute the inserted attacker's code. This can be done by manipulation of the program flow.

One way to achieve this is by utilizing *buffer overflows* for *smashing stacks*. Most programs are structured into subroutines with its own local variables. These variables and also the arguments and the return address are stored in memory segments called stacks. The return address is usually the first on the stack and the local variables are concatenated on the bottom. Normally, like in the C programming language, the content of array variables are written from bottom to the top, and if the range is not checked, the return address can be overwritten (see Figure 3.2). The attacker can manipulate the input data in a way that the return address is overwritten with the address of his malicious code. On the return, the malicious code is executed [Ale96, PB04]. Another possibility is to overwrite the frame pointer address instead of the return address [klo99].

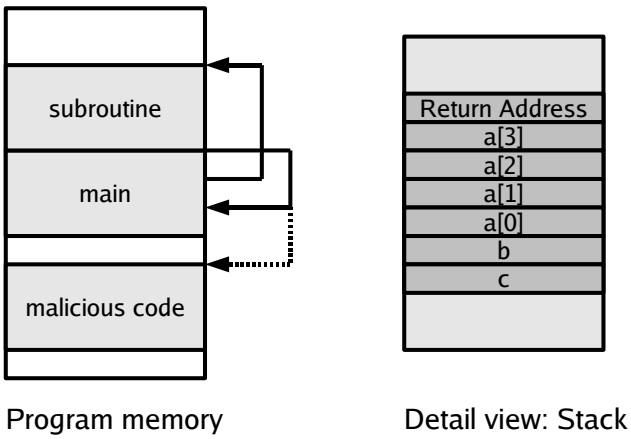


Figure 3.2: On the left side, a part of the program memory is shown. Normally, the subroutine is called and after its execution, the program counter jumps back to the main program after the call instruction. However, if the return address in the stack is overwritten by a buffer overflow of the vector `a[]` (see right side), the erroneous return destination may become the entry point of the malicious code (dashed line).

Heap-based buffer overflows are another class of code injection attacks. The memory heap is the dynamically allocated memory, in C managed by `malloc()` and `free()`, in C++ by `new()` and `delete()`. The heap consists of many memory

blocks which are usually chained together by a double linked list. These memory blocks are managed by the dynamic memory allocator, which can insert, unlink or merge blocks together. The information (pointer to the previous and next block) of the linked lists is stored in a header for each block.

A heap-based buffer overflow may overwrite this header information in a way that one pointer of the double linked list points to the stack segment before the return address [Rag06]. If this block is now freed by the dynamic memory allocator, the header information of the previous and next block are updated. Because one pointer points to the stack segment due to the attack, the stack is updated in a way that the return address is overwritten with the address of a heap block, which can now be executed after the control flow reaches a return [Rag06, PB04]. There exist many other different possibilities to utilize heap-based buffer overflows [Con99, Dob03].

Arc injection or *return-into-libc* is an attack where a system function is exploited to spawn a new process which performs the attacker's desired operations. The name arc injection came from the inserting a new arc (control flow transfer) into the control flow graph (CFG) of a program. In the standard C library (*libc* on UNIX-based systems), there exists a function called `system()`. This function validates a process call given as argument and after successful validation starts its execution as a new thread. The memory location of this standard function is usually known, and therefore also the starting address to bypass the validation of the argument. The return pointer of the stack can now be manipulated by using a stack-based buffer overflow to jump to the desired destination in the system function to execute a malicious process. The name of the malicious process can be transferred to the system function by using registers [PB04]. This attack is useful if the architecture or operating system prevents the stack or heap memory area from execution.

Shacham generalized the *return-into-libc* attacks to show that it is possible to do malicious computation without injecting malicious code [Sha07]. The idea is that due to shared libraries, e.g., libc, many analyzable and attackers known instruction snippets are in the memory. Shacham proposes that an attacker can build an arbitrary program from these snippets which can do arbitrary computation. This can be done by analyzing, for example, the *libc* library for code snippets which end with a *return* instruction. Moreover, Shacham shows that for the x86 architecture, it is possible to use only parts of instructions. The return instruction of the x86 architecture is a one byte instruction encoded with `0xc3`. However, other instructions which are longer consist also of this byte value. By starting the sequence in the middle of an instruction, the original instruction alignment is bypassed which enables the attacker the usage of additional new instruction sequences. From these building block, the attacker can build a program by chaining these snippets together by overwriting the register which stores the return address. This so-called *return-oriented programming* has been successfully transferred to other processor architectures, e.g., SPARC. In [BRSS08], a compiler is introduced which is able to construct return-oriented exploits

from a general propose language. In summary, Shacham shows that preventing the injection of code is not sufficient for preventing malicious computation.

Pointer subterfuge is an attack technique where pointer values are changed. There exist four varieties: *function pointer clobbering*, *data pointer manipulation*, *exception handler hijacking* and *virtual pointer smashing* [PB04].

Function pointer clobbering modifies a function pointer so that the pointer directs to the malicious code. When the control flow reaches the modified function call, the attacker's function is called and his code is executed.

Data pointer modifications can be used for arbitrary memory writes. This technique can be combined with other code injection attacks to launch complex attacks.

Exception handler hijacking modifies the thread environment block (in MS Windows) that points to the list of registered exception handler functions. Because of the fact that the list is stored on the stack, the entries can be easily manipulated to utilize stack based buffer overflows. This technique can be put to work to transfer the control flow to a malicious function. Within Linux, function pointers in the *fnlist* can be replaced to have a similar effect.

Virtual pointer smashing replaces the virtual function table used in the C++ implementation of virtual functions. The virtual function table is used in C++ at runtime to implement dynamic dispatch. Every C++ object has a virtual pointer, which points to the appropriate virtual function table. By modifying the virtual pointer to direct to an attacker's virtual function table, malicious functions can be called when the next virtual function is invoked.

3.2 Invasive Physical Attacks

Invasive physical attacks physically tamper the embedded system with special equipment. Trivial physical attacks only compromise the availability of the system or damage a part or the whole system by physical destruction. Also, switching off the power supply voltage or cutting wires belongs to these trivial attacks.

Other invasive physical attacks aim to read out confidential data or the implementation of IP cores as well as the manipulation of the circuit or data to get access to sensitive data. These attacks have in common that expensive special equipment is used. The realization of these attacks requires days or weeks in specialized laboratories. The first step is the *de-packing* of the circuit chips. This is usually done with special acids [KK99, Hag].

After removing the packaging, the layout of the circuit can be discovered with optical microscopes and cameras. By removing each metal layer the complete layout of the chip can be captured in a map [KK99]. The gathered informations of the layer reconstruction can be used for *reverse engineering* the circuit, which gives competitors the possibility to optimize their product or to obtain more information about the implementation to launch other attacks.

Further information can be collected by *micro-probing* the circuit. This can be done by manual micro-probing, where metal probes have electrical contact to signal wires on chip. This is usually done on a *micro-probing workstation* with an optical microscope [KK99].

Due to the decreased lateral structure dimensions in todays circuit technologies, manual micro-probing is nearly impossible. But there exist advanced technologies, like *ion* or *electron beams*, as well as *infrared laser* which make micro-probing also possible in todays chip manufacturing technologies. With a *focused ion beam* (FIB) the chip structure can be scanned in a very high resolution. The beam hits the chip surface where electrons and ions are sputtered off and can be detected by a *mass spectrometer* [DMW98]. With increased beam intensity, the beam can also remove chip material with high resolution (see Figure 3.3). This can be used to cut signal wires or drill holes to reach signal wires in underlying layers. These holes can be filled with platinum to bring the signal to the surface, where it can be easily micro-probed. With an *electron beam tester*, the activity on the signal wires can be recorded, if the clock frequency is drastically reduced (under 100 kHz). Finally, with the *infrared laser*, the chip can be scanned from rear, because the silicon substrate is transparent in the infrared wavelength range. With the photon current, internal states of transistors or activity on signal wires can be read out [AK96, Ajl95].

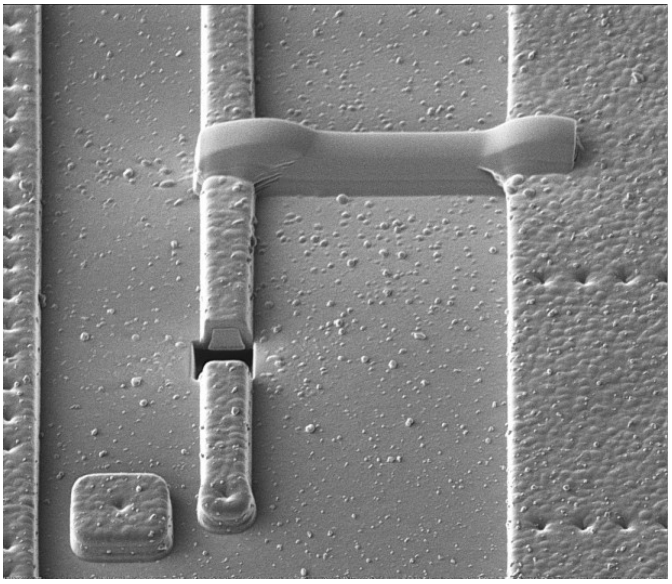


Figure 3.3: A secondary electron image recorded with a *focused ion beam* (FIB). The FIB previously interrupts a signal wire [Fra].

These advanced technologies can be used to launch a variety of attacks. In focus are smart cards with implemented cryptographic algorithms. Most of the time, it is the attackers goal to read a secret key. One example is to read out the memory content

using bus probing. The problem here is the generation of the successive addresses to get a linear memory trace. The attacker can bypass the software by destroying and deactivating the transistor gates which are responsible for branches and jumps with an FIB. The result is a program counter with can only linearly count up, which fits perfectly for this attack [KK99]. Other attacks are reading ROM, reviving and using test modes, ROM overwriting by using a laser cutter, EEPROM overwriting, key retrieval using gate destruction, memory remanence, or probing single bus bits, as well as EEPROM alternation [KK99, Hag].

3.3 Non-Invasive Logical Attacks

To the non-invasive logical attacks belong the following attacks: *phishing*, *authenticity forging*, *attacking cryptographic weaknesses*, and *copying*. The goal of phishing is to gather sensitive information, like passwords, credit card numbers, or whole identities. By means of social engineering, such as fake web sites, emails, or instant massages to imitate a trustworthy person. The victim gives sensitive data away, believing that the attacker is not a harmful person. Phishing belongs to the authenticity forging attacks. Other authenticity forging attacks are DNS or certificate spoofing (manipulation). The difference to phishing is that systems and not persons are cheated.

Cryptographic attacks exploit weaknesses of cryptographic algorithms, e.g., *symmetric ciphers*, *asymmetric ciphers*, or *hashing algorithms* as well as *protocol stacks*. The goals of these attacks are access to sensitive data or to break into a system. More about cryptographic attacks can be found in [FS03] or [RRKH04].

Finally, copying attacks are attacks were sensitive data, like health data, personal data and works, which are protected by copyright, such as music, texts, programs, or IP cores, are copied without authorization. These attacks, especially the gathering of sensitive data and copyright infringement, target the security goal confidentially.

3.4 Non-Invasive Physical Attacks

Eavesdropping and *side-channel attacks* belong to the class of non-invasive physical attacks which normally do not impair the system. Eavesdropping is the interception of conversations or data transmissions by unintended recipients. The attacker can gather sensitive information which is transmitted using electric media, e.g., email, instant messenger, or telephone. Sometimes a combination of eavesdropping and cryptographic weakness attacks are used to monitor sensitive data. For example, sniffing passwords in a *WEP* (Wired Equivalent Privacy) encrypted *WLAN* (Wireless Local Area Network).

Information of cryptographic operations in embedded systems can be gathered by side-channel attacks. Usually, the goal is to get the secret key or information about

the implementation of the cryptographic algorithm. Cryptographic embedded systems are particularly vulnerable to these attacks, because the attacker has full control over the power and clock supply lines. The different side-channel attacks are *timing analysis*, *power analysis*, *electromagnetic analysis*, *fault analysis*, and *glitch analysis* [Hag, KLMR04, RRKH04].

Timing analysis attacks are based on the correlation of output data timing behavior and internal data values. Kocher [Koc96] showed that it is possible to determine secret keys by analyzing small variations in the execution time of cryptographic computations. Different key bit values cause different execution time, which makes a read out and reconstruction of the key possible.

Power analysis attacks are based on the fact that different instructions cause variations in the activities on the signal lines, which result in different power consumption. The power consumption can be easily measured by observing the current or the voltage on a shunt resistor. With *simple power analysis* (SPA) [KJJ99], the measured power consumption is directly interpreted to the different operations in a cryptographic algorithm. With this technique, program parts in a microprocessor, for example DES rounds or RSA operations, can be identified. Because of the execution of these program parts depend on a key bit, the key bits can be read out. *Differential power analysis* (DPA) [KJJ99] is an enhanced method which uses statistical analysis, error correction and correlation techniques to extract exact information about the secret key.

Similar to power analysis techniques, information about the key, data, or the cryptographic algorithm or implementation can be extracted by electromagnetic radiation analysis [AARR03].

During different fault analysis (DFA) attacks, the system is exposed to harsh environment conditions, like heat, vibrations, pressure, or radiation, to enforce faults which result in an erroneous output. Comparing this output to the correct one, the analyst gains insight into the implementation of the cryptographic algorithms as well as the secret key. With DFA attacks, it was possible to extract the key from a DES implementation [BS97] as well as public key algorithm implementations [BDH$^+$98, BS97]. The last one shows that a single bit fault can cause fatal leakage of information which can be used to extract the key. Pellegrini and others show that using fault analysis, where the supply voltage of an processor is lowered, it is possible to reconstruct several bits from a secret key of the RSA cryptographic algorithm [PBA10]. For this attack, they used the RSA implementation of the common OpenSSL cryptographic library and a SPARC Leon3 core, implemented on a Xilinx Virtex-II Pro FPGA. The supply voltage of the FPGA is lowered till sporadic bit errors occur on the calculation of the signature using the FPGA's hardcore multiplier.

Glitch attacks also belong to the class of DFA attacks. Here, additional glitches are inserted into the clock signal to prevent the registering of signal values on the critical path. The simplest attack is to increase the clock frequency. One goal of glitch attacks can be the prevention of branches in a microprocessor program, because of the

calculation and registering of the new branch target address is a long combinatorial path on many processor implementations [KK99].

4 Defenses Against Code Injection Attacks

In this section, we show measures against different *code injection attacks*, as introduced in Section 3.1. A good overview of defenses against code injection attacks is further given in [Rag06] and [Erl07]. The related work in this section is divided into six groups: Methods using an *additional return stack, software encryption, safe languages, code analyzer, anomaly detection*, as well as *compiler, library* and *operation system support*. *Control flow checking* methods, which combine security and reliability issues are discussed in Section 4.7.

4.1 Methods using an Additional Return Stack

Almost all code injection attacks manipulate the *memory-based return stack*. The return stack can be protected by an additional *hardware-based return stack*. Hardware-based return stacks are usually used for *indirect branch prediction* [KE91]. Xu and others propose a *secure return address stack* (SRAS) which redundantly stores a copy of the memory-based stack [XKPI02]. If the return address from the SRAS differs with the processor-calculated address from the memory return stack, an exception is raised which is handled by the operation system to determine whether it stems from a misprediction or from a code injection attack.

Lee and others propose a similar SRAS approach [LKMS04, MKSL03]. Additionally, scenarios are considered when the control flow of a return from subroutine does not comes back to the point the function was called from. Lee suggests either to prevent these situations or to introduce additional instructions which manipulate the SARS to manually resolve such situations.

Ozdoganoglu and others [OVB$^+$06] present another SRAS method called *SmashGuard*. In some situations, the behavior of the correct control flow differs from the *last-in first-out* (LIFO) return stack scheme. In these situations which are often referred to as *setjmp* or *longjmp* calls, the control flow mostly returns to a previous call which is deeper in the stack. Ozdoganoglu resolves these situations by searching the target address of the currently executed return instruction in the complete stack.

Furthermore, Xu and others propose methods to divide the stack into a *control* and a *data stack* in [XKPI02]. Inside the control stack, the return addresses and stack pointers and inside the data stack variables, e.g., buffers are stored. This approach effectively solves the problem of buffer overflows. To achieve the stack split, Xu presents two different techniques, one modifies the compiler and the other is a hardware technique which modifies the processor.

4.2 Methods using Address Obfuscation and Software Encryption

Bhatkar and others [BDS03] and Xu and others [XKI03] propose methods for *address obfuscation*. To exploit buffer overflows and achieve the execution of malicious code, the attacker must know the *memory layout*. Due to address obfuscation, the achievement of such information about the memory structure is enormously complicated for the attacker. In these methods, the program code is modified so that each time the code is executed, the virtual addresses of the code and data are randomized. These approaches randomize the base address of the stack and heap, the starting address of the dynamic linked library, as well as the location of static data. Also, the order of local and static variables as well as the functions are permuted. For objects which cannot be rearranged, Bhatkar inserts random gaps by padding, e.g., in stack frames.

Shao and others proposed hardware assisted protection against *function pointer* and *buffer smashing attacks* [SZHS03, SXZ+04]. The function pointers are XORed with a randomly assigned key for each process which is hard to be reconstructed by the attacker. This is a countermeasure for *function pointer clobbering* (see Section 3.1). Furthermore, Shao introduces a hardware-based *boundary checking method* to avoid stack smashing. On each memory write, it is checked if the write destination is outside the current stack frame and if so, an exception is raised.

If the software is loaded encrypted into the memory and decrypted in the fetch stage, code injection attacks are impossible, because the attacker needs to inject his code encrypted. The key for de- and encryption is different for each process, hence it is impossible for the attacker to encrypt his code properly. Injection of unencrypted code produces data garbage after decryption and results in a crash of the process. Barrantes and others propose a method which uses an *x86 processor* emulator for simulate the decryption in the fetch stage [BAFS05]. The process is encrypted at load time, whereas Kc and others present an approach where the executable is stored encrypted on the hard disk [KKP03]. The proper key is stored in the executable header which is loaded into a special register for decryption. However, the key is also easily extractable for an attacker which lowers the effectiveness of this approach.

4.3 Safe Languages

Many attacks can be launched due to the inherent security flaws which exist in the C and C++ language. Programming in C or C++ allows a lot of programming close to the hardware and the memory layout makes these languages very flexible. However, the programmer must consider many facts if he would like to produce invulnerable code. Safe languages, like Java, are capable of some implementation vulnerabilities which are discussed in Section 3.1. Nevertheless, C or C++ are preferred languages for low-level or even for high-level programming, especially in embedded systems which makes a safe implementation of these languages reasonable.

Cyclone is a C dialect which statically analyses given C code at compile-time and inserts dynamic checks at places where it cannot ensure that the code is safe [JMG+02, GHJM05]. Cyclone is designed to avoid buffer overflows, format string attacks, and memory management errors. However, the C syntax and semantics as well as the capability of low-level programming are preserved. Insecure constructs are refused to compile until more information is provided to make these constructs secure. However, Cyclone programs need existing libraries, like the GNU C library *libc* which usually compiled with a standard C compiler [Rag06]. To secure library functions, the libraries should also be compiled with Cyclone.

Another approach is *CCured* which is a source to source translator for C [NCH+05]. The used techniques are similar to Cyclone, which includes static analysis and dynamic checks on these points where static analyses are not possible. CCured uses pointer and type analysis to make casts secure. However, these techniques make CCured programs incompatible with existing libraries. This obstacle can be solved by introducing *library wrappers*.

4.4 Code Analyzers

Code analyzers can be categorized into two groups: *static code analyzers* and *dynamic code analyzers*. Static code analyzer approaches check either the source code or the compiled object code for vulnerabilities without executing the program. It is impossible to detect buffer overflows statically. Therefore these tools use heuristics whose detection rates are never complete. The term *analyzer* corresponds to a wide area of automatic tools ranging from only considering the behavior of simple code statements to consider the complete source code. Some static code analyzer approaches need *annotations* to the source code whereas other approaches need no annotations.

For example, an annotated static code analyzer is *Splint* [EL02]. It is a lightweight tool which uses annotations to check properties of objects, e.g., the range of a variable. Dor and others introduce the *C String Static Verifier* (CSSV) which is able to detect string manipulation errors [DRS03]. This tool also uses annotations that have

pre-, post-, and side-effect conditions. Furthermore, it analyzes pointer interaction and performs integer analysis. A non-annotated static code analyzer for detection of buffer overflows is described by Wagner and others in [WFBA00]. The analysis is done by formulating buffer overflows as an *integer range problem*. Another non-annotated analyzer approach is *PREfix* [BPS00] and its extension *PREfast* [LBD+04]. These methods build an execution model of the analyzed code which includes all possible execution paths of the program.

Other static approaches use *lexical analysis* which can be implemented as an *editor extension*. The written source code is compared to database entries of vulnerable code snippets. If such an entry is found, the tool might examine them further and report the security impact. Approaches using such lexical analysis are *ITS4* [VBKM00] and *Flawfinder* [Whe].

Dynamic code analyzers add further information to the source code and perform test runs in order to detect vulnerabilities. However, not all vulnerabilities might be detected, because the used input stimulus for the test runs might not cover all situations. *Purify* [HJ92] is a tool which tests software to detect memory errors like uninitialized memory access, buffer overflows, or improper freeing of memory as well as memory leakages. The tool is commercially available from IBM known as *Rational Purify* [IBM]. Haugh and Bishop introduce a dynamic buffer overflow detection tool for C programs, called *STOBO* [HB03]. This tool instruments program code in order to keep track with memory buffers and checks function arguments. If a buffer overflow occurs in a test run, a warning is printed. Ghosh and O'Conner present the *Fault Injection Security Tool* (FIST) in [GO98]. This tool injects malicious strings in buffers and observes the application response to detect vulnerabilities.

4.5 Anomaly Detection

Anomaly detection refers to methods which compare the actual application behavior to a specified application profile. Any deviation from the profile will raise an exception which triggers further measures. The application profile can be user-specified or learned from past executions of the program. A disadvantage of these methods is the high false-positive rates due to the identification of any unusual behavior as an attack.

Hofmeyr and others and Forrest and others propose a method for monitoring system calls for UNIX processes [HFS98, FHSL96]. If the system call pattern deviates from the previous recorded pattern, subsequent actions like program terminations can be taken. A similar approach is presented from Wagner and Dean [WD01]. The occurrence of system calls is also monitored and checked with a *system call model*. The model is built statically from the *control flow graph* of a program, where the control flow graph is transformed into a *system call graph*, which models the sequence of the occurrence of system calls. Sekar and others also use a system call model for anomaly detection [SBDB01]. The model, however, is generated dynamically with

system call recording in a learning phase. Furthermore, techniques using sliding windows which analyze the system calls inside the window are presented by Forrest and others [FHSL96] and Wagner and others [WD01]. Forrest uses a dynamic learning phase whereas Wagner uses static information derived from the control flow graph.

Feng and others use, besides the system call information, the return address from the stack for anomaly detection [FKF+03]. Like the other methods, the checks are done on system calls. During a learning phase, so-called *virtual paths* are recorded. A virtual path can be built with all return addresses, gathered from the stack, on a system call. These return addresses correspond to all unreturned functions. During the execution of the detection phase, the virtual path is checked on every system call to detect anomaly behavior.

Zhang and others present a hardware approach for detecting anomalies in the program behavior [ZZPL04]. In this approach, the detection is done on the *control flow instruction level* which has a finer granularity than the other system call-based approaches. Jump and branch information, like target addresses or favored conditional branch decisions, are stored additionally in the system memory. Fast memory access is assured through common cache structures of the processor. This method has some similarities with our method which will be introduced in Section 4.7.3. However, this approach does not store control flow graphs, rather each branch or jump is separately looked up using a *context addressable memory* (CAM). Hereby, a hash of the branch or jump address is calculated which acts as index for the branch table. Although this approach needs more memory and has a higher latency as our approach, there is no need for synchronization with the control flow of the executed program. The aim of the method is to recognize attacks by detecting anomalous behavior. Therefore during a learning phase, the decisions of conditional branches are recorded and stored in the branch table. If the recorded decisions differ from the control flow behavior in the detection phase, a warning signal will be risen, whereas if the control flow diverges from the stored jump and branch information, a threat is signaled. The approach was extended with an anomalous path detection which compares sequences of branch decisions of the executed program with the decisions recorded in the learning phase [ZZPL05]. In the second approach, *general indirect jumps* (non returns from subroutine) are considered as well.

4.6 Compiler, Library, and Operating System Support

In this section we discuss countermeasures for code injection attacks through enhancement of compilers, libraries, or the operation system.

4.6.1 Compiler Support

Compilers are the most convenient place to insert countermeasures for code injection attacks without changing the programming language. Most attacks exploit buffer overflows to overwrite stack based content. Therefore, many approaches propose *stack frame protection* measures. In the stack, mainly the return address or frame pointers are in focus of attackers. Other items which can be protected with security enhanced compiler support are pointers in the program code. Buffer overflows occur, if there is more data written to the buffer, than its capacity can hold (see Section 3.1). Therefore, *boundary check methods* are in focus of this section.

There exist many different methods to protect the return address inside the stack, for example *StackGuard* [CPM+98, CBD+99], *Stack Shield* [Ven00], or *Return Address Defender* (RAD) [CH01]. StackGuard places a so-called *canary word*[1] between the return address and the local variables inside the stack. Before executing the return instruction, the canary word is checked and verified whether it is intact. By exploiting buffer overflows for return address alteration, the canary word is also overwritten which can be detected. Stack Shield uses a redundant return address which is copied in the data segment in the beginning of the function. Before leaving the function with the return jump, the return address is compared to the copy. If the addresses differ, the program will be terminated. A similar technique is used by RAD. However, the redundant copy of the return address is stored in an array in the data segment which is called *return address repository* (RAR). The RAR is further protected by so-called *mine zones* or *read only techniques*. Mine zones are the read only array boundaries, which protect the RAR from buffer overflow attacks. All these return address protection methods can be applied as a compiler patch for the *gcc compiler*. However, attacks described in [BK00] and [Ric02] are able to cancel the Stack Shield and StackGuard protection. Foremost, StackGuard is vulnerable if the attacker uses a pointer which directly points to the return address which allows the return address alteration without destroying the canary word.

Cowan and others introduce a compiler extension for *pointer protection*, known as *PointGuard* [CBJW03]. The technique protects pointers through encrypting them while they are in the memory. Additional en- and decryption operations are inserted in pointer read and write sequences at compile-time. For example, by accessing a pointer, the pointer is decrypted to a processor register which is safe against malicious overwriting. If a pointer is altered by overwriting the memory during a buffer overflow attack, the decrypted result points to a different location which prevents the access of malicious code.

[1] The term *canary word* corresponds to the miner's canary which was used in coal mines as an early warning system. If there were toxic gases in the mine, the birds died before the miners were affected. Canaries sing a lot, which made them very suitable for a visual and audible warning system. The last canaries in mines were phased out in 1986 in the UK [BBC05].

Lhee and others propose a compiler extension which inserts additional buffer size checks to prevent buffer overflows at runtime [LC02]. The buffer size information is read out of a compilation with debugging information of the program. Using this information, additional checks are automatically inserted into the source code.

Erlingsson and others propose a fine-grained software-based memory access control technique called XFI [EAV+06, ABEL09]. This technique enriches the program code to grant access to an arbitrary number of memory regions. Furthermore, the entry and exit point of a program can be controlled using XFI. Budiu and others propose additional instructions to extend the instruction set architecture (ISA) for XFI hardware support [BEA06].

Jones and Kelly propose a method to identify *out-of-bound pointers* [JK97]. Every result of a pointer arithmetic must reference the same object as the original pointer. If not, the pointer is out-of-bounds. Such pointers can be identified dynamically by additional instructions which are included at compile-time and a new object table which is maintained during the execution. If a pointer is out-of-bounds, this pointer value is set to '-2'. The problem of this approach is that out-of-bound accesses are not allowed in ANSI C, however, such pointers are used in many programs. Therefore, Ruwase and Lam extend this approach with an *out-of-bound object* and call this approach *C Range Error Detector* (CRED) [RL04]. If a pointer becomes out-of-bounds, it is redirected to a special out-of-bound object which keeps the original pointer value and the referenced data. This approach prevents buffer overflows, because all data written over the bounds of the buffer are automatically redirected to other memory locations managed by the out-of-bound object.

4.6.2 Library Support

Many buffer overflows are caused by mishandling vulnerable *standard C library functions*. Particularly, string handling functions are vulnerable for buffer overflow attacks. Therefore, the obvious solution is to design safer libraries. Safe string function replacements are `strlcpy()` and `strlcat()` [MdR99] and *SafeStr* [MV05] which are immune to buffer overflows and format string vulnerabilities. *FormatGuard* is a patch for the glibc library to protect the `printf()` function from format string vulnerabilities [CBB+01].

Baratloo and others introduce two methods against buffer overflows which are completely transparent: *libsafe* and *libverify* [BST00]. Both approaches are implemented as dynamic link libraries under the Linux operating system. The library *libsafe* intercepts all calls to vulnerable functions of the glibc library and substitutes these calls with alternative functions which are not vulnerable to buffer overflow or format string attacks. The library *libverify* uses binary re-writing of the process memory to verify critical elements of the stack frame before they are used. The verification and protection against buffer overflows is similar to the StackGuard [CPM+98] approach, however the implementations differ. Whereas StackGuard is applied during

the compilation, *libverify* embeds the verification code at the start of the process. The advantage is that the code does not have to be recompiled which makes this approach completely transparent to the user.

Robertson and others [RKMV03] and Krennmair [Kre03] propose countermeasures for *heap-based attacks*, described in Section 3.1. The allocation and deallocation routines of the standard C library are modified to protect the header of the heap segment. Robertson includes a padding mechanism and a checksum in the header on frame allocation and verifies these information, if the segment should be freed. Krennmairs technique, called *ContraPolice*, protects the heap pointer in the header of each heap segment by randomly generating canaries like the StackGuard approach for stack-based headers.

4.6.3 Operation System Support

Finally, the operating system can be enhanced to protect programs from code injection attacks. *Non-executable stack* prevents the execution of malicious code, injected into the stack. However, this approach prevents some allowed situations where code is executed in the stack. Examples are *functional programming languages* which generate code during runtime in the stack, *function trampolines* for nested functions used by the *gcc compiler*, or *stack-based signal handling* which is used by Linux. A patch for a non-executable stack for the Linux operating system was provided in [Des97] which also handles the above mentioned executions by disabling the protection in case of these situations. However, this approach is defeated by Wojtczuk [Woj98].

Lately, processor vendors have introduced hardware support to prevent the execution of code from the stack. With a new flag, the so called *NX* (No eXecute) bit, memory regions can be declared page-wise as non-executable areas which are excluded from execution by the hardware. Non-executable stack approaches for the Linux operating system, like *PaX* [PAX03] or *Exec Shield* [vdV04] are able to use this NX bit, or emulate it on processors which have no NX bit support. The technique can be combined with write protection to achieve that no memory location in the process can be marked as writable ('W') and executable ('X'). This so called *W⊕X protection* prevents attackers from injecting malicious code with subsequent execution. Nevertheless, Shacham demonstrated that it is not necessary to inject code in order to do malicious computations [Sha07] (see also Section 3.1).

StackGhost is an operation system-based approach to protect the stack frame for systems running on the *SPARC* architecture [FS01]. This method utilizes special SPARC features like the *windowed register file* and provides a redundant copy of the return address. StackGhost is available as a patch for the *OpenBSD* kernel.

4.7 Control Flow Checking

Control flow checking (CFC) denotes the task of checking the control flow of a program according to a given specification. The specification is derived mostly statically at compile-time from the program. Control flow checking combines reliability and security issues and is a countermeasure against *single event effect*, *degeneration faults*, *code injection* and *invasive physical attacks*.

Related work on control flow checking can be divided into completely software-based approaches and approaches using an additional hardware *checker unit* or a *watchdog processor*. Usually in these approaches, the program code is first structured into basic blocks[2]. Other approaches achieve control flow checking by redundantly decoding the control flow instructions in an additional checker unit (e.g., [RLC+07]).

4.7.1 Software-Based Methods

Software-based control flow checking techniques belong to the so-called *software implemented hardware fault tolerance* (SIHFT) methods. A famous software-based CFC technique is called *Control Flow Checking using Assertions* (CCA) [KNKA96, MN98]. After the creation of the *control flow graph* (CFG, see also Section 4.7.3), special control instructions are inserted into the program code at the beginning and the end of a basic block (see Figure 4.1). The approach introduces two identifiers which are set and checked with these instructions. At the entrance of a basic block, a *basic block identifier* is assigned to a variable. Moreover, corresponding to the control flow graph, a special *control flow identifier* is checked and subsequently set for the next basic block. At the end of a basic block, both identifiers are verified, so erroneous jumps or branches from or in the middle of a basic block are detected. By checking the control flow identifier, the correct processing order of the basic blocks is ensured. The advantage is that no hardware modules are required, however this approach has impact on the performance of the program code and the erroneous jumps can only be checked at the transitions of the basic blocks.

The approach is enhanced for real-time distributed systems to achieve a lower performance overhead and faster error detection in [ANKA99]. The additional instructions are inserted at an intermediate level of the compiler which makes this technique, called *Enhanced Control Flow Checking with Assertions* (ECCA), language independent.

Another software-based control flow checking approach called *Block Signature Self Checking* (BSSC) is introduced by Miremadi and others [MKGT92]. The code is also structured into basic blocks and additional instructions are added at the entry

[2] A basic block is a sequence of code which is executed successively without any jumps or branches except, possibly, at the end. The basic block can only be left at the end of a block and can only be entered at the beginning. Only the last instruction can be a jump or branch and only the first instruction can be a jump or branch destination (see Section 4.7.3).

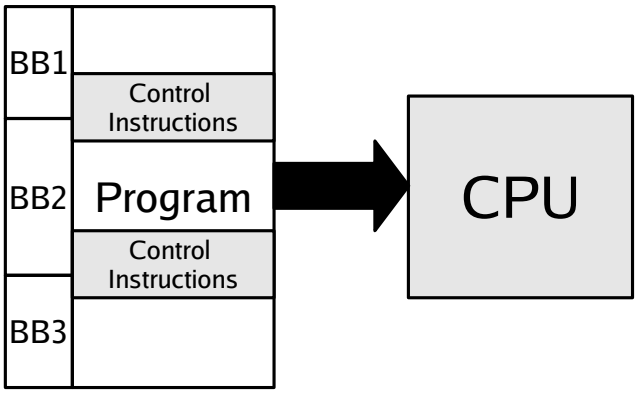


Figure 4.1: Control instructions are usually inserted before and after a basic block (BB1-BB3) in software-based control flow checking approaches. The additional control flow instructions check the executed control flow according to the control flow graph.

and at the end of each block. Checking is done by storing a signature, e.g., the current address, into a variable at the basic block entrance. Before leaving the basic block, this signature is verified. The method can verify the subsequent linear execution of the basic block. However, the processing of the correct basic block order cannot be verified.

Oh and others introduce a technique called *Control Flow Checking by Software Signatures* (CFCSS) [OSM02]. A unique signature is assigned to each basic block and the signature is embedded with the signature difference to the predecessor block in the code. During the execution, a runtime signature is calculated and stored in a general purpose register. The signature of the last block and the stored signature difference are used to calculate and verify the current runtime signature at each basic block entrance. In other words, this approach checks if the correct predecessor of the current basic block, according to the CFG, was processed. However, if a basic block has more than two predecessors, the method is not applicable. In this case, an additional runtime variable is introduced to resolve the problem. Borin and others propose error classification for control flow checking and analyze the error coverage of the most existing software-based approaches [BWWA06]. Furthermore, they introduce two methods which are enhancements to the original CFCSS method. The first method is called *Edge Control Flow Checking* (EdgCF) and the second is called *Region Based Control Flow* (RCF) technique.

Other similar approaches are SWIFT [RCV+05] and YACCA [GRRV03, GRRV05]. All these method insert control instructions at the basic block borders into the program code, as depicted in Figure 4.1.

Bagchi and others introduce the *Preemptive Control Signature* (PECOS) checking, which is able to prevent jumps or branches in case of an error [BLW+01, BKIL03]. The program code is equipped with additional checker instructions before each control flow instruction. The runtime target address is determined and verified with the valid target addresses, extracted from the compiled code. The list of valid target addresses is stored inside the code, whereas the runtime target address is determined by loading the control flow instruction into a register and decode it by software. If the runtime target address is not in the list with valid addresses, an exception is triggered which prevents the execution of the erroneous jump or branch. The drawback of this approach is that only the integrity of the control flow instruction in the memory is checked. Transient faults, such as single event effects in the control path cannot be detected by this method.

Abadi and others introduce a software-based CFC technique for security issues called *Control Flow Integrity* (CFI) [ABEL05, ABEL09]. This method focuses on indirect calls and returns. The destination of indirect calls and returns are determined at compile-time, and each of these jump destination are labeled with a unique identifier in the code. Instructions to check the identifier of the destination are added to the program code before an indirect jump. Only if the identifier is correct, the jump is executed. Budiu and others present an *instruction set architecture* (ISA) extension which introduces new instructions for CFI hardware support [BEA06].

4.7.2 Methods using Watchdog Processors

A *watchdog processor* is a simple coprocessor which is able to monitor the behavior of a main processor in order to detect errors [Lu82, MM88]. The predecessor of the watchdog processor is the *watchdog timer* [CPW74, OCK+75]. A watchdog timer is reset by the program running on the main processor at certain intervals. If the monitored program hangs, the timer is not reset anymore. Therefore, the timer produces an overflow which generates an interrupt. Inside the *interrupt service routine*, countermeasures can be started, e.g., the erroneous program can be terminated.

A watchdog processor is initialized with information about the main processor or the process which should be monitored. At runtime, the watchdog processor concurrently collects information about the execution of the program in the main processor and compares the gathered with initial information to detect errors. The information may include the *memory access behavior*, the *control flow*, the *control signals*, or the *reasonability of results*. Mahmood and McCluskey give a survey over error detection with watchdog processors in [MM88]. Traditionally, a watchdog processor is coupled with the main processor via a system bus. However, other approaches exist where the watchdog processor is directly attached by dedicated signals.

The advantages of watchdog processors are the lower overhead than the duplication of the main processor, the possibility of concurrently checking the execution which results in none or only little performance degeneration and the detection of common

or related design errors of the program or the processor due to the diversity of the processors architectures.

Watchdog processors, when used for control flow checking, have a *watchdog program* which is derived from the control flow of the checked program. The control flow of a program can be represented in a graph whose nodes represent sequences of code, e.g., basic blocks or whole functions, and the edges between the nodes represent the control flow. An identifier, often called signature which is known by the watchdog program is attached to each node. The signatures can be assigned at random or they can be derived form the instructions inside a node. Techniques using the arbitrarily assigned signatures are called *assigned-signature control flow checking* and techniques using the derived signature are called *derived-signature control flow checking* [MM88]. The different watchdog processor approaches can be further categorized by the storage of the watchdog signatures. Therefore, the methods can be divided into two groups, called *Embedded Signature Monitoring* (ESM) and *Autonomous Signature Monitoring* (ASM). ESM methods embed the watchdog signatures into the code of the checked program. To verify a signature, the corresponding signature must first be transferred to the watchdog or to the main processor, depending on the comparison location. The watchdog processors for the ASM methods have their own memory to store the signatures. Therefore, the watchdog must be initialized with all watchdog signatures before the program execution. In summary, there exist four categories for control flow checking with watchdog processors (see Table 4.1).

	signature storage location	
	ESM	ASM
Assigned-Signature CFC	SEIS [PMHH93]	SIC [Lu82], ESIC [MH91]
Derived-Signature CFC	PSA [Nam82], SIS [SS87]	Cerberus-16 [Nam83], RMP [ES84]

Table 4.1: Four different categories for control flow checking using watchdog processors with some example references. The methods are categorized by different watchdog signature storage locations (embedded into the program: ESM; in additional memories for the watchdog processor: ASM) and the different type of signatures (derived, assigned) according to [MHPS96].

Watchdog-processor-based CFC approaches can be further categorized according to their error detection capability. The first category checks that the nodes are processed in an allowed sequence whereas approaches of the second category verify the

instruction sequence inside a node. The third category includes schemes which do both [MM88].

Assigned-Signature Control Flow Checking

During the execution, the arbitrarily assigned signatures used for assigned-signature CFC are transferred to the watchdog processor for verification. Usually, the transferred signatures are compared by the watchdog processor to the watchdog signatures, stored in a separate watchdog memory (ASM method). The advantages of these methods are the ease of implementation and the possibility to perform runtime checks asynchronously. However, the drawbacks are the performance impact, due to the program-based transfer of the signatures to the watchdog with additional control flow instructions, and the low error coverage since only the sequence of the signatures is checked.

The first known method is introduced by Yau and Chen [YC80] which assigns prime numbers to loop-free intervals which are checked at runtime. Lu proposes a method called *Structural Integrity Checking* (SIC) [Lu82]. The method assigns labels to high-level control flow structures which are verified by the watchdog processor. The approach is enhanced by Majzik and Hohl which is called *Extended Structural Integrity Checking* (ESIC) [MH91].

An embedded signature monitoring approach for assigned-signature CFC is introduced by Pataricza and others, called *Signature Encoded Instruction Stream* (SEIS) [PMHH93, MHPS96]. In this approach, each basic block is assigned a unique signature which further encodes the successor basic block. The signatures are transferred to the watchdog processor during the execution which verifies the control flow of the program only using the information encoded in the signatures. Therefore, the watchdog processor needs no signature storage memory and initialization phase.

Derived-Signature Control Flow Checking

Derived-signature CFC uses a signature calculated from the properties of the executed instructions inside a node. To check all instructions, a signature, e.g., an XOR, hash or CRC value, of all instructions of a basic block is calculated offline (at compile-time). At runtime, a checker unit calculates the signature of the executed instruction in a basic block. When leaving a basic block, both signatures can be compared and errors inside the basic block can be detected. The derived-signature CFC methods can also be categorized by the storage of the precalculated (golden) signature in ESM and ASM methods.

Embedded Signature Monitoring *Derived-signature ESM methods* store the offline calculated signature (golden signature) in the program code with additionally

inserted instructions at the end or the beginning of each basic block. During runtime, the calculated signatures from the watchdog processor are compared to these embedded signatures. The advantage of these methods is that all instructions can be checked and a new program already contains the corresponding signature (see Figure 4.2). The disadvantages are the performance impact and that a fault can only be detected at the end of a basic block which may be too late. Also, a single event upset during the execution of the additionally inserted instructions can lead to a false detection or spoofing of an error.

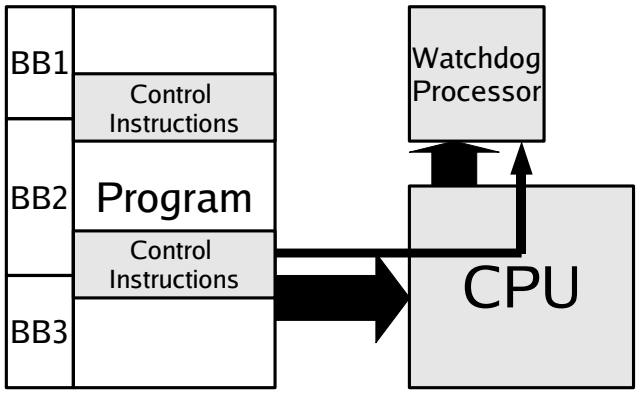


Figure 4.2: In the *derived-signature ESM CFC* approaches, a signature is calculated from the executed instructions by a watchdog processor. The golden signatures are calculated at compile-time and embedded with control instructions in the code. The additional control instructions read out the runtime calculated signature from the watchdog processor and compare it to the golden signature.

According to [MM88], the first derived-signature CFC method is introduced by Namjoo [Nam82] and is called *Basic Path Signature Analysis* (Basic PSA). The signatures are calculated by XORing over all instructions inside each basic block. After the calculation of the signatures at compile-time, the signatures are stored at the beginning of each basic block. The watchdog processor monitors the instruction stream and identifies the loading of the signature from the instruction memory. During the execution of the basic block, the watchdog processor calculates the runtime signature by XORing the processed instructions and, at the end of a basic block, the signatures are compared. A very similar technique is proposed by Sidhar and Thatte in [ST82].

Other approaches use *linear feedback shift registers* (LFSR) [DS90, DS91] or *checksums* [SM90] as signatures or try to lower the number of used signatures by using larger blocks which include multiple branches [SS87, WS90]. Gaisler enhanced his *ERC32* processor with an ESM CFC technique where the signatures are embed-

ded into NOP instructions [Gai94]. Meixner and others store the signatures for the *Argus-1 checker* into unused instruction bits of the SPARC ISA to reduce the performance and memory overhead of their ESM method [MBS07, MBS08]. If insufficient unused bits are available, they also embed the signature into NOP instructions.

Upadhyaya and Ramamurth propose a derived-signature CFC technique using *m*-of-*n* codes [UR94]. An *m*-of-*n* code is an *n*-bit code whose bit values have *m* ones. At compile-time, the signature of a basic block is calculated, for example, by XORing the instructions. If the intermediate result is an *m*-of-*n* code, then this instruction is tagged. At runtime, the watchdog calculates the signature, recognizes the tagged instructions and verifies on the tagged instructions if the runtime signature is an *m*-of-*n* code. At the basic block borders, an additional byte is inserted which adjusts the current signature to an *m*-of-*n* code in order to force a check. The advantage is that the signature must not be stored in the program code. However, one additional byte per branch is necessary to force a check in order to restart the runtime signature calculation at a new basic block. A similar approach is presented by Ohlsson and Rimen called *Implicit Signature Checking* (ISC) [OR95]. The implicit signatures are the current start addresses of the basic blocks. This can be achieved by using additional justified signatures embedded into the code.

Autonomous Signature Monitoring The golden (compile-time calculated) signatures of *derived-signature ASM methods* are stored in a separate memory for the watchdog processor. The comparison between the golden and the runtime calculated signature is implemented in hardware (see Figure 4.3). If the control flow graph is mapped into the instruction memory of the watchdog processor, the jumps and branch destinations can also be checked. The advantages are that the program code does not need to be altered and that there is no performance impact. Also, all instructions can be monitored. The disadvantages are that extra memory is required for the checker unit and the synchronization between the CPU and the checker unit is difficult. Therefore, interrupts, multi threading, and indirect jumps cannot be covered completely.

One of the first approaches using the ASM scheme is the *Cerberus-16* watchdog processor [MM88, Nam83]. The control flow graph and the corresponding signatures are mapped in the microinstructions which are stored into the watchdog processor memory. The Cerberus-16 processor only has control flow instructions with encoded signatures and instructions for the communication with the main processor. The processing of the control flow of the main and the watchdog processor are completely synchronized. The approach is extended by Michel and others by a *branch address hashing* (BAH) technique, now called *Watchdog Direct Processing* (WDP) which reduces the memory overhead for the watchdog processor memory [MLS91].

An asynchronous ASM approach is presented by Eifert and Shen [ES84, ST87] which is called *Roving Monitor Processor* (RMP). At compile-time, the control flow

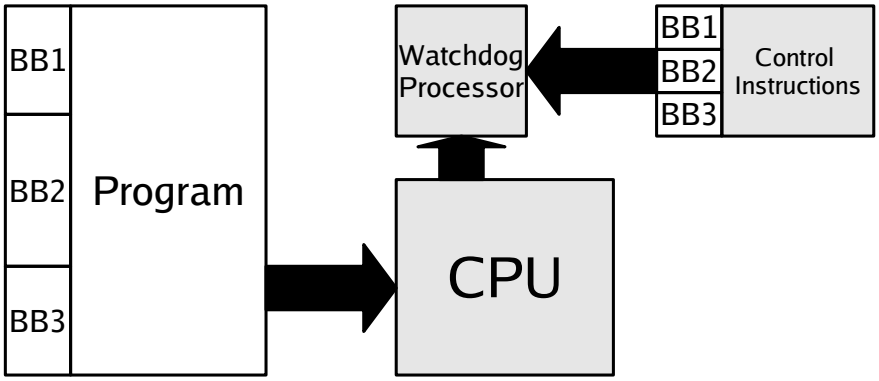


Figure 4.3: In the derived-signature ASM approach, the watchdog processor has a separate memory for storing the control instructions. The execution must be synchronized between the CPU and the watchdog processor.

of the program is extracted and the signatures (CRC values) are calculated. During the execution, the runtime signatures are calculated with an additional signature generation unit and, at the end of a block, the calculated signature is sent to the watchdog processor. The watchdog compares the received signature to the signatures achieved from the control flow graph and stores then in the watchdog memory. At branches, the received signature is compared with the two successor signatures of the current node in order to determine the next signature. This approach can also be used to check multi-processor systems where each processor has its own signature generation unit and sends the signature via a signature queue to the watchdog processor which is responsible for the whole system. The approach is extended by Madeira and Silva who introduce the *Online Signature Learning and Checking* (OSLC) technique. In this approach, the golden signature is generated during a learning phase [MS91]. The learned signatures are stored in the watchdog memory of an RMP-like watchdog processor.

Arora and others describe an ASM approach for security applications in [ARRJ06]. This hardware approach consists of three parts: the *Inter-Procedural CFC*, the *Intra-Procedural CFC*, and the *Instruction Stream Integrity Checker*. The Inter-Procedural CFC verifies the function calls and returns by implementing the function call graph in hardware using *content addressable memories* (CAMs) and an FSM. The Intra-Procedural CFC checks the basic blocks by a compile-time generated control flow graph, implemented in checker memories. Finally, the Instruction Stream Integrity Checker is similar to those of other ASM methods, however, they use hash functions to generate and verify the signatures.

Our new control flow checking approach introduced in Section 4.7.3 belongs to the class of *derived-signature ASM methods*. We propose a term *checker unit* for the

watchdog processor, because our unit is very simple with only few hardware overhead. However, like in the Cerberus-16 or the WDP approach, the control flow graph is mapped into microinstructions which are stored in a separate memory. Further advantages of our control flow checker are the fast error detection due to the tight integration into the processor, the error recovery possibility, and the expandability with modules which support more control flow instructions or detect more errors. Moreover, like the other ASM-methods we have no performance impact on the error-free case and the program must not be altered.

4.7.3 New Control Flow Checking

In this section, new methods for control flow checking in embedded processors and IP cores are presented. First, a classification of control flow instructions in embedded RISC processors is given. Subsequently, two different methodological concepts for control flow checking of direct jumps and branches are introduced and possibilities to check indirect jumps are discussed. Then, methods for repairing a corrupted program path are proposed. Finally, methods for checking *Finite State Machines* (FSMs) in general IP cores are analyzed.

Branches and Jumps

Control flow instructions (CFI) can be categorized into conditional branches and unconditional jumps. Conditional branches depend on the result of a logical or an arithmetic operation. On most processor architectures, the arithmetic operation affects a register, called *integer condition codes* (icc). This register consists of flags which describe properties of the result, for example whether the result is greater than zero, or negative. Conditional branches evaluate this register for the decision to take or not take the branch. The way of evaluation (e.g., branch if the zero flag is set) is statically coded in the instruction itself, whereas the evaluation of the condition is performed at runtime.

Both groups of control flow instructions can be further subdivided into *direct* (static) and *indirect* (dynamic) jumps or branches. The destination of direct branches or jumps is fixed at compile-time and is encoded into the jump or branch instruction in an absolute or relative address. For indirect jumps or branches, the destination address is determined during program execution. The destination address is given in absolute or relative manner by either a register value or as the result of an operation with registers or the result of an operation with a register and a constant value which is encoded into the instruction.

In summary, four types of control flow instructions exist:

- *(Unconditional) direct jumps* (e.g., `call, goto`),
- *(Conditional) direct branches* (e.g., `if .. then .. else`),

- *(Unconditional) indirect jumps* (e.g., `return` from subroutine), and
- *(Conditional) indirect branches*[3].

Furthermore, the class of unconditional indirect jumps can be subdivided into *returns from subroutine*, *register indirect calls* and *other jumps*. A return from subroutine is an example of an indirect jump, because the program counter jumps to the address where the routine is called from, and this address is only known at runtime. Register indirect calls are calls where the address of the called subroutine is determined at runtime. This usually happen in C++ if a *virtual function* is called.

Finally, jumps which are not triggered by an instruction can occur such as *interrupts* and *traps*. The destinations of interrupts are typically given by the start address of the main interrupt service routine, and so, interrupts belong to the class of direct jumps. Traps occur on exception conditions (like divide by zero). Here, the program redirects to the address of an exception handler, and so, traps can be treated as direct jumps, too.

Table 4.2 presents an analysis of the quantity of these different types of branches and jumps in the code on the *SPARC V8* [SPA] architecture for the *SPEC CINT2000 benchmark* [SPE] for a given list of programs. As can be seen, indirect calls and jumps occur relatively rarely as opposed to direct branches and jumps.

SPEC	all	direct		indirect		
program	instructions	branches	jumps	returns	calls	other jumps
gzip	19979	1426	599	111	4	0
gcc	566280	54791	22446	2236	140	273
vpr	51771	2764	2012	269	2	7
mcf	3881	288	82	26	0	0
crafty	82891	4814	4074	108	0	13
parser	36862	3189	1701	320	0	2
gap	236181	18733	4158	828	1262	5
vortex	174567	12537	8491	913	15	21
bzip2	12162	748	380	73	0	0
twolf	102899	5701	2060	189	0	2

Table 4.2: Accumulated number of all and different kinds of control flow instructions of benchmarks of the SPEC CINT2000 test suite [SPE] when compiled to the SPARC V8 [SPA] architecture.

[3]Note that conditional indirect branches are not supported by any instruction set architecture that we know of.

Methods for Checking Direct Jumps/Branches

In SoCs, a CPU often executes only a few specified programs over its lifetime. This holds true particularly for embedded applications where the system is often only programmed once, and the code is never changed during the lifetime of the product, except for the update of the SoC with a new firmware and software. Furthermore, it is well known that in many computational intensive problems, most of the execution time is spent in only few subroutines. So, it is beneficial to analyze these subroutines for branches and jumps statically.

If we assume that only direct jumps and branches exist in a given code segment, we will show that we are able to verify the control flow of this code and guarantee the correct execution of each direct control flow instruction as well as the (successively) linear execution of all the other instructions (the program counter value is incremented by one word address after each instruction). To verify the correct execution of control flow instructions, we need to check whether the address of the control flow instruction and the target address are correct. The program counter value before and after the execution of a control flow instruction can be compared to these addresses. If there is a mismatch, an error signal is raised. To check a non control flow instruction, the program counter before and after the execution of the instruction can be compared. If the second one is not an increment of the first one, the error signal is also raised.

In the following, we propose two alternative methods to obtain the correct addresses of control flow instructions of a given machine program and the corresponding targets. The first method is called *basic block* or *control flow method* (CF). The second method is called *control flow instruction method* (CFI).

Control Flow Method First, a given compiled machine code is separated into a set of *basic blocks* (BB). A basic block is a sequence of code which is executed successively without any jumps or branches except, possibly, at the end. The basic block can only be left at the end of a block and can only be entered at the beginning. Only the last instruction can be a jump or branch and only the first instruction can be a jump or branch destination. The following instructions define the beginning of a basic block [TH07]:

- the first instruction in a program or segment,
- the instruction following a control flow instruction,
- the instruction which is a destination of a control flow instruction.

From this information, the control flow graph $CFG(BB,T)$ is built: Each node $BB_i \in BB$ of the control flow graph represents a basic block. The nodes are sorted with increasing start address of the corresponding basic block in ascending order. Each edge $t_j \in T$ represents a transition of the control flow from one basic block to

another. If the last instruction of a basic block BB_i is a direct branch instruction, the basic block has two successors. One is the basic block next in the list BB_{i+1} (if the branch is not taken), and to a basic block where the first instruction is the branch destination (if the branch is taken). Jumps have only one successor, and if the last instruction is not a control flow instruction, the successor basic block is always the next basic block BB_{i+1}. An example program and the corresponding CFG are shown in Figure 4.4 which is separated into basic blocks.

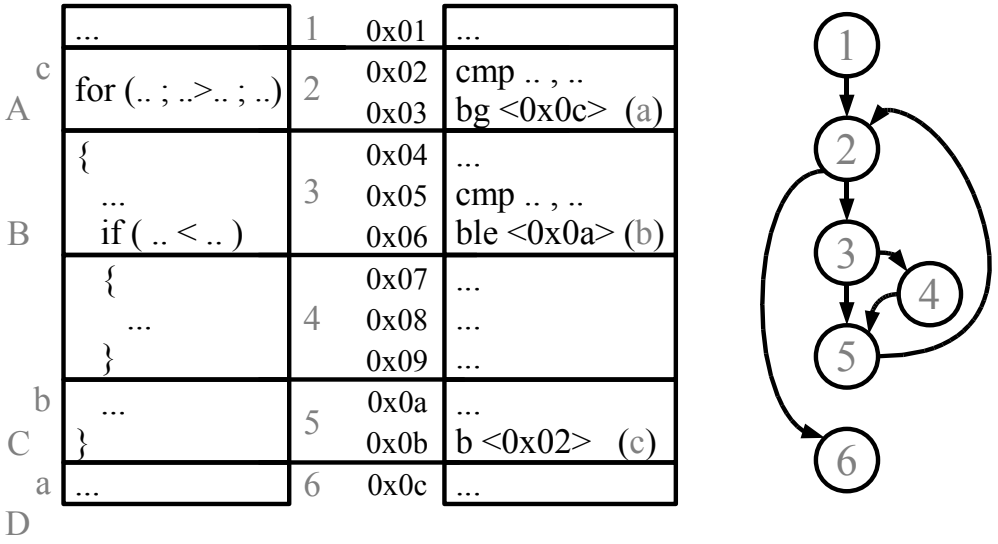


Figure 4.4: An example program code is given on the left hand side together with the corresponding assembler code. The CFIs are denoted A to C, and the CFI destination addresses a to c. D denotes the end of the program or segment to be checked. Furthermore, the code is divided into basic blocks BB_i, $i = 1,\ldots,6$. On the right hand side, the corresponding control flow graph (CFG) is shown.

With the given CFG, we have all information to check a sequence of program counter values for correctness leading to the specification of a proper control flow checker unit as follows: The information of the CFG can be either used to directly define a *finite state machine* (FSM) to check the correctness of a sequence of control flow instructions. Alternatively, an implementation using micro-instructions of a micro-programmed circuit can be deducted from the CFG.

For an implementation of the checker unit as a micro-programmed circuit, the information of the CFG can be stored inside memories. For each basic block, we need to store the start and the end address and also the indices of the successor basic blocks. The start address of each basic block is the end address of the previous basic block incremented by one. To minimize the memory overhead, we can store only the

end address and a global start address. Also, we only need to store one successor of a basic block for branches because if the branch is not taken, the basic block with the next index (BB_{i+1}) is always executed.

With these memory overhead improvements, we need three memory items for each basic block inside the memory:

- One for the basic block end address ($addr$),
- one for the index of the branch taken successor basic block (suc), and
- a flag ($flag$) which identifies the type of the last instruction of the basic block.

The flag is needed to choose the right transition to the next basic block (see Figure 4.5). Note that if the last instruction of a basic block is not a CFI, the successor basic block index (suc) is not needed.

index	*addr*	*suc*	*flag*
1	0x01	-	N
2	0x03	6	B
3	0x06	5	B
4	0x09	-	N
5	0x0b	2	J
6	0x0c	-	N

Figure 4.5: Three memory areas are necessary to store required information for each CFG. In the first column ($addr$), the address of the last instruction of the basic block is stored. The successor basic block for a taken branch is stored in the second column (suc). In the third column ($flag$), a flag is stored which identifies the type of the last instruction of a basic block. An N denotes a non control flow instruction, whereas a B denotes a branch. This example memory stores the values for the example program in Figure 4.4.

The control flow checking algorithm is depicted in C language in Listing 4.1. For checking the control flow, we need the current program counter (PC) and the following program counter (nPC). The algorithm, implemented as a C function, returns 0 if the control flow for the program counter and its successor is correct, and -1 if the control flow differs from its specification. Further, the index i of the current basic block and the three memories ($addr$, suc, and $flag$) are needed. The function $addr(i)$ returns the entry with index i of the memory $addr$.

Listing 4.1 Control flow (CF) checking algorithm

```
1  int check_cf( PC, nPC) {
2    static int i;                          // index i
3    if (addr(i) == PC) {                   // PC is BB end
4      if (flag(i) == 'J') {                // uncond jump
5        if ((addr(suc(i)-1)+1) == nPC) {   // correct?
6          i = suc(i);
7          return 0;
8        } else return -1;
9      } else if (flag(i) == 'B') {         // cond branch
10       if (((addr(suc(i)-1)+1) == nPC) { // branch taken
11           i = suc(i);
12           return 0;
13       } else if (PC +1 == nPC)) {        // branch not taken
14           i++;
15           return 0;
16       } else return -1;
17     } else {                             // non CFI
18       if (PC +1 == nPC) {                // correct?
19          i++;
20          return 0;
21       } else return -1;
22     }
23   } else {                               // non BB end
24     if (PC +1 == nPC) return 0;          // correct?
25     else return -1;
26   }
27 }
```

By looking up the basic block end address in the *addr* memory ($addr(i)$), we know when the basic block end is reached (Line 3). If the basic block end is not reached, the address of the next program counter must be the current address incremented by one (Line 23). If not, an error occurs. If the basic block end is reached, we must distinguish between the different types of the last instruction inside the basic block (Line 4, 9, and 17). If this is an unconditional jump, like a *call*, only the jump target must be checked for correctness. The corresponding target address is the start address of the successor basic block, given by its index. To get this address, the end address of the basic block with the previous index is fetched and the address is incremented ($BB_{i-1}+1$ or Line 5). Furthermore, the current index i must be updated to the successor basic block index (Line 6). If the last basic block instruction is a conditional *branch*, two possible successor basic blocks exist. If the branch is taken (Line 10), the handling is the same as for an unconditional jump. If the branch is not taken (Line 13), the next program counter value should be the current value incremented by one ($nPC == PC+1$). Hence, the next instruction belongs to the successive basic block and also the basic block index i must be updated (Line 14).

Finally, if the last instruction of a basic block is not a CFI, the checking behavior is the same as on conditional branches, where the branch is not taken (Line 18).

One very similar approach of a CF method is described in [ARRJ06]. Here, the CFG is implemented in hardware by a finite state machine and a lookup table for resolving the control flow instruction addresses and indices (in memory). The disadvantage of this approach is that the checker unit must be synthesized new for each program. Here, in our memory-based approach, only the contents of the memories need to be reconfigured in order to check a new program.

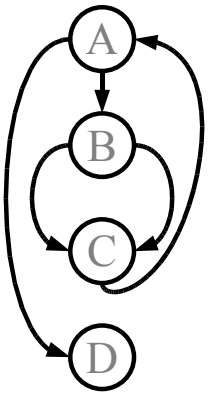
Control Flow Instruction Method In contrast to the control flow (CF) method, the control flow instruction (CFI) method is based on storing control flow instructions instead of basic blocks. In case of direct branches and jumps, the start and target address are known at compile-time. So, it is possible to extract this information from the binary or the disassembled program code by decoding the instructions. The control flow instructions are then sorted by increasing addresses in ascending address order.

Then, the *control flow instruction graph* $CFIG(CFI, T)$ is built: Here, each control flow instruction in the code which should be checked represents a node ($CFI_i \in CFI$). Directed edges $t_j \in T$ of the CFIG denote transitions to the next following control flow instruction in the given code.

Like in a CFG, each node can have a maximum of two successors: two for a branch instruction and one in case of a jump instruction. For a branch instruction CFI_i, one successor is CFI_{i+1} (branch is not taken). The other successor of a direct branch and jump instruction is CFI_n which is the next control flow instruction in the program code after the branch destination (branch is taken). The CFIG of the example program code form Figure 4.4 is shown on the left side of Figure 4.6. Note that D is not a CFI, rather it refers to the end of the checking segment or function.

Like in the CF method, the information of the CFIG can be used as a specification of the correct branching behavior inside a control flow checker unit and implemented either directly by an FSM or by micro-instructions of a micro-programmed circuit. In case of a micro-programmed circuit implementation, we store for each CFI the start and the target address in memory ($addr$ and $target$ in Figure 4.6). Also, the index of the successor CFI must be stored inside this memory (suc in Figure 4.6). For direct branches, we store the successor CFI for taken branches. If the branch is not taken, the successor CFI is CFI_{i+1}. Finally, we need a flag ($flag$) to distinguish between the different CFI types.

A proper control flow instruction checking algorithm is shown in Listing 4.2. Like the CF algorithm, the inputs are the current program counter *PC* and the next program counter *nPC* and the output is a 0 in case of a correct control flow, or a -1 in case of an error. The checking algorithm needs the four memory columns, introduced in



index	*addr*	*target*	*suc*	*flag*
A	0x03	0x0c	D	B
B	0x06	0x0a	C	B
C	0x0b	0x02	A	J
D	0x0c	-	-	-

Figure 4.6: For the example program code in Figure 4.4, the corresponding CFIG is shown on the left hand side. The nodes correspond to the control flow instructions, whereas the edges denote transitions. On the right hand side, the four memory areas are shown which are necessary to store the CFIG. In the *addr* memory, the address of the CFI is stored and the corresponding target address is stored in the *target* column. In the third column (*suc*), the successor CFI index is stored. Finally, the kind of instruction is stored in the last column ($flag$).

Figure 4.6 and the index i, which denotes the next CFI from the current program flow position.

The algorithm is quite similar to the CF method, with the difference of accessing the jump or branch targets and the missing check of basic block ends with a non CFI. In Line 3, we check if the current executed instruction is a CFI. If it is not, the linear successive program flow is checked (Line 18). If the current program counter references to a CFI, we must also distinguish between the different types of CFIs (Line 4 and 9). If the CFI is an unconditional jump, the next program counter should be the value stored in the target memory column (*target*, Line 5). Also, we must update the index i to the index of the successive CFI ($suc(i)$, Line 6). If the current CFI is a conditional branch, we must check if the branch is taken or not (Line 10 and 13). If the branch is taken, the same checking strategy as in the case of unconditional jumps is used. If the branch is not taken, the next program counter must be the current one, incremented by one (Line 14). In both cases, the index i must be recently updated.

Memory Overhead Discussion In the following, the different memory overheads of both methods shall be compared. The example program in Figure 4.4 has 6 nodes in the CFG and 4 nodes in the CFIG. For the CF method, we need to store only one address for a CFG node (basic block end address *addr*) and the index of the

Listing 4.2 Control flow instruction (CFI) checking algorithm

```
1  int check_cfi(PC, nPC) {
2    static int i;                          // index i
3    if (addr(i) == PC) {                   // PC is CFI
4      if (flag(i) == 'J') {                // uncond jump
5        if (target(i) == nPC) {            // correct?
6          i = suc(i);
7          return 0;
8        } else return -1;
9      } else {                             // cond branch
10       if (target(i)) == nPC) {           // branch taken
11           i = suc(i);
12           return 0;
13       } else if (PC +1 == nPC)) {        // branch not taken
14           i++;
15           return 0;
16       } else return -1;
17     }
18   } else {                               // non CFI
19     if (PC +1 == nPC) return 0;          // correct?
20     else return -1;
21   }
22 }
```

successor block. In the CFI method, we need to store two addresses (control flow instruction address *addr* and target address *target*) and the index of the successor block for each CFIG node. Both methods also need bits to store the flags for distinguishing the different CFI or basic block types. Usually, the index needs less bits than the addresses of instructions, so the CF method uses less memory than the CFI method for this example.

For measuring the memory overhead for standard user programs, we use the programs from the SPEC CINT2000 [SPE] benchmark in the following (see Section 4.7.3). Table 4.3 shows the memory overhead caused to implement the CF and CFI method for the SPEC CINT2000 benchmark when compiled to the 32-bit SPARC V8 [SPA] architecture. The smallest possible index bit width is chosen for the given program to calculate the memory overhead in bits.

Also, the memory overhead of the checking methods are compared with the memory usage of the test programs. The number of instructions of the test programs are presented in Table 4.2. On the SPARC V8 architecture, each instruction needs 32-bit of memory space. The additional memory overhead of the checker methods are shown in absolute values and in percentage of the memory usage of the test program in Table 4.3.

The results in Table 4.3 show that the CF method usually produces a lower memory overhead than the CFI method and in a range of typically less than 20%. Note that

SPEC prog.	CF method				CFI method			
	# BB	Ind. w. [bits]	Overhead [bits]	[%]	# CFI	Ind. w. [bits]	Overhead [bits]	[%]
gzip	2615	12	109830	17.2	2140	12	154080	24.1
gcc	90819	17	4268493	23.6	79886	17	6151222	33.9
vpr	6029	13	259247	15.6	5054	13	368942	22.3
mcf	514	10	20560	16.6	398	9	27324	22.0
crafty	10205	14	449020	16.9	9009	14	666666	25.1
parser	6384	13	274512	23.3	5212	13	380476	32.3
gap	30484	15	1371780	18.1	24986	15	1873950	24.8
vortex	24978	15	1124010	20.1	21977	15	1648275	29.5
bzip2	1502	11	61582	15.8	1201	11	85271	21.9
twolf	9827	14	432388	13.1	7952	13	580496	17.6

Table 4.3: Required memory overhead of the programs of the SPEC CINT2000 benchmark in bits for the CF and CFI method. Also, the number of basic blocks and control flow instructions, and the corresponding index width is shown. The memory overhead is shown in absolute values and in percentage of the memory usage of the corresponding test program.

the shown overhead is for checking the whole program, with all subroutines which is not always the best way. By restricting the checking to only few subroutines which are executed very often and should have a high reliability and security, the memory overhead can be significantly reduced.

Instruction Integrity Checker The *instruction integrity checker* (IIC) is an extension to the control flow method to check all types of instructions, not only control flow instructions. A *CRC* (cyclic redundancy check) or hash value may be calculated offline for all instructions inside a basic block (at compile-time) and online inside the checker unit [MLS91]. The offline calculated CRCs are stored inside an additional memory which extends the other checker memories (see Figure 4.5). For each basic block, we store the end address *addr*, the index of the next basic block *suc* (for a jump or a taken branch), the flags $flag$ and additionally the CRC or hash value in the memory *iic* (instruction integrity check).

The checker unit calculates a CRC from the instructions during the execution of a basic block. At the last instruction of the basic block, the calculated CRC can be compared with the offline calculated CRC, stored inside the *iic* memory. If the CRCs are not equal, one or more bits are false in the instruction stream. This error can be signaled to the operating system by an interrupt or the system might be rebooted by

a hardware reset. A re-execution or correction is not or hardly possible, because we are able to detect an error inside a basic block only at the end of the block.

The instruction integrity checking is not applicable for the CFI method, because there may exist more than one path to a CFI node, whereas in the CF method a basic block is always traversed on the same path. As an example, consider the *if clause* in the program in Figure 4.4. In the CF method, if the branch (if) is not taken, only basic block 5 is traversed. If the branch is taken, basic blocks 4 and 5 is transversed, but basic block 5 is executed on the same way as if the branch was not taken. Unlike in the CFI method, if the branch is taken or not, always the CFI *C* is the successor. However, through the way to *C* the program flow takes different paths, depending on whether the branch is taken or not.

Walking through different paths to a node results in different CRC or hash values. With the IIC method, we are only able to store one CRC or hash value in the additional memory column for one node. Surely, we could extend the memory to store a value for each possible path, but this would result in a huge memory overhead, because we must reserve memory space for each node. This shows that the instruction integrity checker is not practical to the CFI method.

Conclusions Both introduced methods can only check direct branches and jumps, where start and destination addresses can be extracted from the compiled code.

The advantage of the CF method is that in most cases, fewer additional memory resources are needed than for the CFI method. The disadvantage of the CF method is that memory handling is more difficult. On many processor architectures, the fastest execution of one instruction is one clock cycle. Consider Algorithm 4.1, where we need access to the *addr* memory for each control flow instruction twice, once for the end address of the basic block (Line 3) and once for the start address of the successor basic block (Line 5 and 10). To achieve this in a single clock cycle, we need a dual-port memory which is more expensive than single port memories. Furthermore, for the second access to the memory, we need first the successor index from the *suc* memory. To do both memory accesses in one clock cycle is nearly impossible on high-clocked processors. Furthermore, the access to the *suc* memory cannot be scheduled one clock cycle before, because if the current basic block consists only of one instruction, and the previous basic block ends with a branch instruction, the current index *i* depends on the result of the executed branch (taken or not). This shows us that we need at least two clock cycles to check a transition in CFG. To ensure that on a basic block end the correct start and destination addresses are available, we might pre-read both values. This can also be done with a single ported *addr* memory. On the first clock cycle, the basic block end address is read from the *addr* memory and the successor basic block index is read from the *suc* memory. On the second clock cycle, the target address is read from the *addr*. But this pre-read can only be done if the basic block consists of more than one instruction. If a basic block consists

of only one instruction, we must stall the processor pipeline to verify the control flow instruction to prevent possible erroneous behavior. Fortunately, basic blocks with only one instruction are very rare.

The CFI method, on the other hand, requires only one memory access for each memory. In Line 3 of the Algorithm 4.2, we access the *addr* memory to get the next CFI address. In the same clock cycle we can access the *target* memory to fetch the correct destination address (Line 5 and 10). With the CFI method, it is possible to check a transition in the CFI graph with at least one clock cycle. Therefore, the CFI method has no execution time overhead at all.

The advantages of the CFI method are that the checker unit is very simple and uses only few logic resources. Also, we have no performance impact, because the correct control flow instruction address and target address may be loaded from the memory in a single clock cycle. The disadvantages are that usually more memory resources are needed as for the CF method and that we are not able to check the integrity of non control flow instructions.

Finally, both introduced concepts for control flow checking have the big advantage over [ARRJ06] in being reprogrammable. Thus, only the memory of the control flow checker unit needs to be reprogrammed so to check a different program. No adjustments of the hardware are thus necessary. Moreover, we have no performance impact for verify the control flow like the software-based methods.

Methods for Checking Indirect Jumps/Branches

Checking *indirect jumps* or *branches* is more difficult than direct branches or jumps, because the jump destination cannot be determined from the compiled program code. In fact, according to the instruction specification of indirect jumps, all possible targets which are inside the reachable area of the jump, are allowed. From the hardware side, also a falsified indirect CFI which jumps to a wrong address is in accordance to the processor specification. Almost all code injection attacks target this behavior by manipulating indirect jump targets (the *return stack*). However, from the software or logical side, there are certainly some restrictions of indirect jump targets: Compilers use indirect jumps in a stylized manner which can be analyzed [LB94]. Almost all indirect jumps which are compiled from a modern program language, like C, C++, or Java, are returns from subroutine, or either belong to a `switch case` clause, which is implemented using a jump table, or are indirect calls which are mainly used in object-oriented languages, like C++ or Java. Indirect jumps which appear in hand-written code are nearly impossible to analyze. Fortunately, hand-written assembler code is used more and more rarely today.

The results reported in Table 4.2 show that returns from subroutine are clearly the main usage of indirect jumps. Upon a call, the address of the back-jump is stored inside a register or a memory stack, and on a return from subroutine, a back-jump to this address is initiated.

Indirect jumps are also used for jump tables to efficiently implement `switch case` clauses. Here, the alternative `case` targets are assembled in a jump table which is addressed by the previously calculated operator. Furthermore, the targets may be direct jumps which lead the control flow to the desired code segment (see Figure 4.7). Another way to use a jump table is to call different functions, depending on an input. Here, the alternative function pointers are stored inside a jump table, whereas the index of the table is calculated with the input value. The address of the desired function is fetched from the table and is called with an indirect jump. Note that jump tables are not often used by compilers. Usually, `switch case` clauses are translated to an `if .. else if` tree. But, depending on the compiler and optimization parameters, indirect jumps might nevertheless occur. Indirect jumps which result from jump table implementations are listed in Table 4.2 under the category "other jumps".

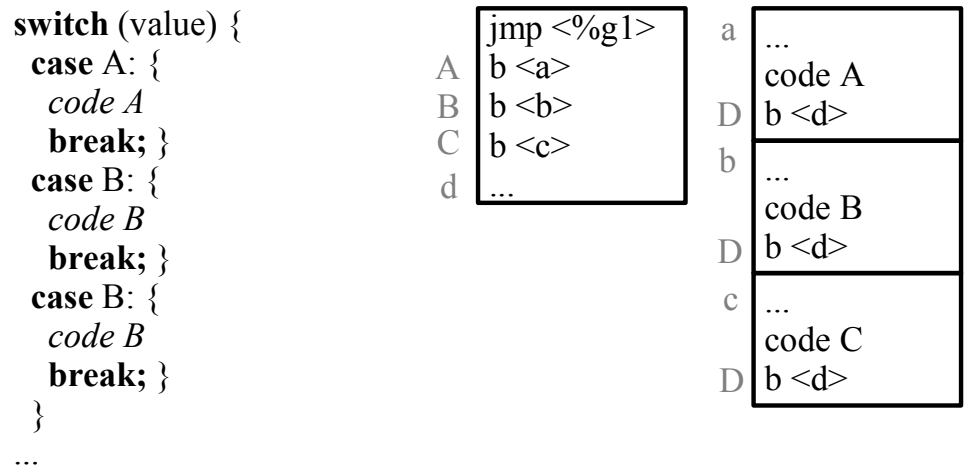


Figure 4.7: An pseudo C example code of a `switch case` clause is shown on the left side. On the right side, a possible implementation in assembler with a jump table and an indirect jump is depicted. The upper case letters (A-D) are direct jump instructions and the corresponding targets are depicted in lower case letters (a-d).

Finally, the indirect jumps are also used as *indirect calls* (see Table 4.2). During indirect calls, the address of the target function is loaded inside a register and with an indirect jump the function will be called. This occurs mainly in object-oriented programming languages that support function pointers and virtual functions. However, functions called by a jump table also use indirect calls.

The methods for checking indirect jumps that will be described in the following can be categorized into methods using information which are gathered by analysis or simulation at compile-time and methods which are only using runtime information.

Methods Using Compile-time Information If we are able to analyze the targets of indirect jumps at compile-time, we can extend our hardware checking units to support multiple jump destinations for monitoring program code that includes indirect jumps. Cifuentes and Emmerik [CE99] present a method to identify indirect branch targets, if the indirect jump is used within a jump table. Furthermore, simulations with different input stimuli may also be helpful to identify indirect jump targets. However, to get the possible jump targets by simulation requires a high effort.

Another approach is to convert all indirect jumps into direct jumps and branches. Bergstra and Middelburg [BM07] present a method to convert most indirect jumps in a compiled program, including jump tables and returns, into direct jumps and branches. The length of program code could be extremely increased and the performance could be reduced by this method.

Methods Using Runtime Information

Methods using runtime information do not need information from the compiled code. Here, we monitor the control flow at runtime to decide if the execution of the indirect jump is correct or not.

Most indirect jumps are returns from subroutine (see Table 4.2). By executing the return from subroutine instruction, the program counter jumps to the next address after the instruction, were the subroutine was called. The return address is typically stored in a register inside the CPU so the return instruction is a special indirect jump. Returns can be verified by implementing an additional *hardware stack* [KE91]. On a call (direct or indirect), the return address is stored in the stack and when the return instruction is executed, the back-jump can be verified.

Furthermore, indirect branch prediction units can be used to evaluate an indirect jump address. Branch prediction is used in pipelined processors to avoid pipeline stalls on branches. A prediction is made if a branch will be taken or not and the next instructions will be fetched according to the prediction. If the prediction was right, no stall occurs, if not, the pipeline must be stalled and the right instructions must be fetched.

Indirect branch prediction units predict destinations of indirect jumps. The predictions are made based on the jump behavior in the past [CHP97, SFF+02, JMKP07]. The result of an indirect branch predict unit might be used to evaluate how reliable the jump destination is. If the prediction is correct, then the probability that this jump is correct is high, but if the prediction is incorrect, the jump destination could be false. A non-predicted indirect jump target has a lower trustworthiness. With this method, no exact proposition can be made, but, for example, a higher level autonomic operating system can evaluate this jump confidentially to increase the reliability of the whole system.

Methods for Handling a Corrupt Control Flow

In the sections above, methods for autonomous monitoring the control flow were described. However, what can we do if an error is detected? There are three opportunities:

- The faulty instruction can be re-executed.
- The CPU can be transferred into a secure state.
- The CPU can continue executing the code at a lower reliability level.

If an error in the control flow occurs, the faulty instruction might be re-executed as follows: The error should be detected fast enough to ensure that the state of the CPU is not altered by the erroneous instruction execution. To guarantee this, a possible checker unit must monitor the program counter in the first pipeline stage of a given RISC CPU. Unfortunately, in most architectures, the jump or branch instructions need more than one cycle to execute. So, until the error is detected, some other instructions after the jump might be executed already. After error detection, the program counter is reset to a value previous to the error by looping back the program counter value from a subsequent pipeline step or by a calculated value from the checker unit. The details of the re-execution process depend highly on the processor architecture and design.

For example, the SPARC V8 architecture allows to execute one instruction after a branch instruction or two instructions after a jump instruction before the branch or jump is performed (see SPARC Architecture Manual [SPA]). If an error is detected and the jump or branch instruction must be re-executed, also these following instructions must be re-executed. It must also be ensured that these instructions cannot alter the state (e.g., register content or memory operations) of the CPU before re-execution. If the retry also fails, the instruction cache can be invalidated to ensure that on the next re-execution, the instructions are transferred again from the memory. If a predefined number of retries fail, the checker unit can lead the CPU into a secure state. Also, the number of retries can be reported to the operating system to show how reliable the CPU is.

Another possibility to react in the case of an error is to transfer the CPU into a secure state. This state can be the reset state or any other state until the program was executed correctly. After reaching this state, the operating system can initialize the CPU with correct data and the CPU can start to execute from this clean state. The invalidation of the data resulting from the erroneous task can be also done by the operating system.

However, the CPU might continue executing the code at lower reliability level with deactivated checker if the task has a low reliability requirement or is further checked by another process.

In all cases, the operating system should be informed about the error and update the internal reliability state of the CPU. If many errors occur, the CPU should only be allowed to execute tasks with low reliability requirements or unimportant tasks, or should finally be excluded by the dispatcher and shut down.

Additional Reading

In [ZT08a, ZT09, Zie10, SBE+07b, SBE+07a, MWB+10], more information about this control flow checker unit can be found.

5 IP Protection

Intellectual property (IP) denotes the absolute right on an *intangible asset*, like music, literature, artistic works, discoveries, inventions, words, phrases, symbols, designs, software, or IP cores. The owner of the IP can license his work to other people or companies. IPs are protected by law with patents, copyrights, trademarks, and industrial design rights.

Drimer defines the following protection or defense categories against IP theft or fraud [Dri09]: *social, reactive*, and *active protections*.

Social protection means that IP works are protected by laws, non-disclosure agreements, copyrights, trademarks, patents, contracts, and so on. The deterrents are conviction by a court of law and the loss of a good reputation. However, these deterrents are only effective if the misconduct can be proven and the appropriate laws exist. Furthermore, the laws must be enforced which is handled differently from country to country.

Reactive protection means that the theft or fraud cannot be prevented, however, it can be detected and delivers evidence of the misconduct. Some reactive protection mechanisms deliver only suspicious facts which, however, may be enough to trigger further investigations. Furthermore, the persistence of reactive protection mechanisms might deter would-be attackers.

Active protection means that physical or cryptographic mechanisms prevent the theft or fraudulent usage of the protected work. This category has the highest deterrent degree. However, these mechanisms can be broken by attacks. Often the attack can be proven if the misconduct is detected.

In this work, we concentrate on the protection of the IP of hardware cores. These so called *IP cores* are distributed like software and can easily be copied. Some core suppliers encrypt their cores and deliver special development tools which can handle encrypted cores. The disadvantage is that common tools cannot handle encrypted cores and that the shipped tools can be cracked so that unlicensed cores can be processed. Another approach is to hide a signature in the core, a so-called *watermark*, which can be used as a reactive proof of the original ownership. There exist many concepts and approaches on the issue of integrating a watermark into a core.

In general, hiding a unique signature into user data, such as pictures, video, audio, text, program code, or IP cores is called *watermarking*. Embedding a watermark

into multimedia data is achieved by altering the data slightly at points where human sense organs have lower perception sensitivity. For example, one can remove frequencies which cannot be perceived by the human ear by coding an audio sequence into an MP3 file. Now, it is possible to hide a signature into these frequencies without decreasing quality of the coded audio sequence [BTH96].

One problem of watermarking is that for verification, the existence and the characteristic of a watermark must be disclosed, which enables possible attackers to remove the watermark. To overcome this obstacle, Adelsbach and others [ARS04] and Li and others [LC06] presented so-called *zero-knowledge watermark* schemes which enable the detection of the watermark without disclosing relevant information.

The watermarking of IP cores is different from multimedia watermarking, because the user data, which represents the circuit, must not be altered since functional correctness must be preserved. A *fingerprint* denotes a watermark which is varied for individual copies of a core. This technique can be used to identify individual authorized users. In case of an unauthorized copy, the user, the copied source belongs to, can be detected and the copyright infringement may be reconstructed. Watermarking procedures can be categorized into two groups of methods: *additive methods* and *constraint-based methods*.

In additive methods, the signature is added to the functional core, for example, by using unused lookup-tables in an FPGA [LMSP98]. The constraint-based methods were originally introduced by [KLMS+01] and restrict the solution space of an optimization algorithm by setting additional constraints which are used to encode the signature.

A survey and analysis of watermarking techniques in the context of IP cores is provided by Abdel-Hamid and others [AHTA04]. Further, we refer to our own survey of watermarking techniques for FPGA designs [ZT05]. A survey of security topics for FPGAs is given by Drimer [Dri09] who also maintains the *FPGA design security bibliography* website: `http://www.cl.cam.ac.uk/~sd410/fpgasec/`.

In order to compare different watermarking strategies, some criteria are defined in the following [HP99]:

Functional correctness: This is the most important criteria. If the watermark process destroys the functional correctness, it is useless to distribute the core.

Resource overhead: Many watermarking techniques need some extra resources. Some to generate and store the watermark itself, some because of the degradation of the optimization results from the design tools. The ratio between the original and the watermarked core's resource demand is defined as the resource overhead.

Transparency: The watermark procedure should be transparent to the design tools. It should be easy to integrate the watermarking step into the design flow, without altering common design tools.

Verifiability: The watermark should be embedded in such a way that the authorship can be verified easily. It should be possible to read out the watermark only with the given product and without any further information from the design flow which must be requested from a company suspected of IP fraud.

Difficulty of removal: The watermark should be resistant against removal. The effort to remove the watermark should be greater than the effort needed to develop a new core, or the removal of the watermark should cause corruptness of the functionality of the core. Watermarks which are embedded into the function of the core are in general more robust against removal than additive watermarks.

Strong proof of authorship: The watermark should identify the author with a strong proof. It should be impossible that other persons can claim the ownership of the core. The watermarking procedure must be resistant against tampering.

In this section, we first discuss IP protection methods using core encryption. After that, related work using additive and constraint based watermarking methods is presented.

5.1 Encryption of IP Cores

The goals of active IP protection for cores are, first, that the core cannot be used without a proper license and, second, that the core is protected from unauthorized modifications. The cores can be delivered in encrypted form and are decrypted by design tools. Other approaches for FPGAs use an encrypted configuration bitfile which is decrypted on the FPGA.

5.1.1 Encrypted HDL or Netlist Cores

One solution is to deliver encrypted IP cores to the customers and integrate de- and encryption functions into the EDA tools. The customer buys the encrypted core and obtains the appropriate key from the IP core developer or vendor. This technique is applicable for IP cores of all *abstraction* or *technology levels* (RTL – HDL cores, logic level – netlist cores, device level – bitfile/layout cores). However, if, e.g., an HDL core should be protected at all abstraction levels, the synthesis tool must produce an encrypted netlist. This must be done for all steps: decryption of the core, processing, and encryption. It is important that the customer only has access to the

encrypted data, which means that the EDA tool routines must be protected against read out attacks.

The problem is that no consistent industrial standard exists which handles encrypted IP cores [Dau06]. This complicates the interoperability of IP cores and EDA tools.

Today, *symmetric* and *asymmetric cryptographic approaches* are used. Using symmetric cryptographic approaches, the en- and decryption is done with the same key. The advantage of this approach is the reduced computational complexity compared to asymmetric approaches. One problem is the secure distribution and communication of the key. Furthermore, EDA tools must deal with different keys for different IP vendors, and if one key is cracked, usually all IP cores of the corresponding vendor have lost their protection. Nevertheless, this approach is used, for example, by Xilinx to encrypt some of their parameterizable HDL IP cores, e.g., the Microblaze processor softcore [Xild].

Methods using asymmetric cryptography are also known as *public key cryptography* which need two keys, the *private* and the *public key*. The private key is for decryption inside the EDA tools, where as the encryption key is publicly available and is used by the IP core vendor. The EDA vendor creates the key pairs and embeds the private key in his tools. The IP core developer can now use the public key for the encryption. The advantage is that the private decryption key may not be transferred over untrusted communication channels and is only known by the EDA vendor. The disadvantage is that asymmetric approaches have a high computational complexity which results in long runtime for decryption up to several hours for IP cores [Dau06]. Another drawback is that the IP vendor must create a separate version for each EDA tool, which is encrypted with the corresponding public key of the EDA vendor.

Dauman, Vice President of the *Synopsys' Synplicity Business Group*, introduced a hybrid approach [Dau06]. The IP core is encrypted with a symmetric cryptographic method, like *Triple-DES*, or *AES* using a key which is generated by the IP vendor. This key, now referenced as the *data key*, is encrypted with an asymmetric cryptographic method, like *RSA* [RSA78], with the public key of the EDA vendor. This approach is similar to the *PGP* approach [Zim95] for cryptographic privacy and authentication of messages. The decryption is done with the (decrypted) data key and the cryptographic method which is specified by the IP vendor. Inside the EDA tools, there exist different symmetric cryptographic routines for the decryption of the core. The advantage is that the decryption with a symmetric algorithm is very fast and the computational complex asymmetric method is only used for the data key which is very small compared to the whole IP core. Synplicity suggested this approach as future industry standard and includes this method called *ReadyIP* into the product *Synplify Premier* [Syn].

In 2007, a industry-wide panel discussion [Wil07] provided some insight into the perception of encrypted IP cores of the EDA industry. The conclusion was that the current social-based protection works well for large cooperations. A better solution

is desirable but not necessarily urgent. However, they express their reservation to small companies or startups which are not known in the community and might not be willing to sell IP cores to these companies.

Barrick argued against the usage of encrypted netlist cores due to their hidden costs [Bar]. The disadvantages are the fixed constraints, the prevention to reuse parts of the logic for other cores, slower simulation speed or inaccurate behavioral models, restriction of the choice of EDA tools, and fewer debugging possibilities. However, sometimes encryption can be worth due to reduced acquisition costs.

5.1.2 Encrypted FPGA Configurations

Another kind of IP protection is the encryption of the FPGA *configuration* or *bitfile*. The bitfile is stored in a *non-volatile memory*, e.g., a PROM, and transferred to the FPGA encrypted. Inside the FPGA during the configuration, the bitfile is decrypted. This approach prevents copy attacks for bitfile designs and protects the bitfile from reverse engineering.

The first suggestion of this method was in 1995 by a patent from Austin [Aus95]. The first FPGA devices which offered configuration encryption was the Actel's 60RS family. However, all FPGAs had the same permanent key, which prevents no copy attacks. Furthermore, the key was also stored in the software. Consequently, it was easy for attackers to extract the key from the software. Xilinx introduced configuration decryption with a Triple-DES hardcore for Virtex-II devices in the year 2000. The user defined key can be stored and updated into an FPGA internal battery-backed SRAM. Today, bitfile encryption is supported by many high-end FPGA families. Some FPGA devices, such as the Altera Stratix II/III, can be configured to always perform decryption. This prevents the configuration with bitfiles which are not encrypted with the proper key.

There exist two different key storing techniques: *volatile* and *non-volatile*. Volatile key storing uses low power SRAMs which are powered by an external battery. Attackers must keep powering the key storage during the attack, which is more complicated. On an attacker's error, the key is cleared and the bitfile cannot be loaded. The disadvantage is the increased printed circuit board space and costs for the external battery. Non-volatile key storage uses fuses, flash, or EEPROMs. The problem is that these technologies must be combined with the latest CMOS technology on the same chip, which affords in a non-standard manufacturing step. The results are increasing costs and more complex verification strategies.

An important aspect of methods using encrypted FPGA configuration bitfiles is the *key management* which includes the generation and the distribution of keys. Kean suggests a method where the FPGA can encrypt and decrypt bitfiles with hardware cores and a permanent embedded key [Kea01]. The FPGA is able to encrypt the bitfile on the first programming and store this encrypted bitfile in a non-volatile memory.

Upon every FPGA configuration during the power-up cycle, the bitfile is loaded and decrypted in the FPGA. The advantage is that the key never leaves the FPGA.

Bosset and others [BGB06] propose a method for using partial reconfiguration for en- and decryption of FPGA bitfiles by user-defined soft cores. At power-up, the decryption core is initially loaded form the PROM which decrypts the bitfile with the user logic. Soudan and others [SAH] propose a method for the encryption of partially reconfigurable bitfiles using device-specific keys.

5.2 Additive Watermarking of IP Cores

Additive methods are watermarking procedures, where a signature is added to the core. This means that the watermark is not embedded into the function of the core. Nevertheless, the watermark can be masked, so it appears to be part of the functional part. Additive watermarks can be embedded into HDL, netlist, bitfile or layout cores.

5.2.1 HDL Cores

Additive watermarking for HDL cores seems to be very complicated, because of the human-readable structure of the HDL code. Hiding a watermark there is very difficult, because on the one hand, an attacker may easily detect the watermark, and on the other hand, subsequently used design tools might remove the watermark during circuit optimization. However, it is not impossible to include an additive HDL component into the core, which may not removed by the design tools.

Castillo and others hide a signature into unused space of dedicated lookup table based memory [CPG+06]. To extract the signature, an additional logic monitors the input stream for a special *signature extraction sequence*. If this sequence is detected, the signature is sent to the outputs of the core. This approach was later generalized for other memory structures in [CPG+08]. The drawback is that distribution as an HDL core is not possible, because the signature extracting logic is easy to detect and to remove.

Oliveira presents a general method for watermarking *finite state machines* (FSMs) in a way that on occurrence of a certain input sequence, a specific property exhibits [Oli01]. The certain input sequence corresponds to the signature which is previously processed by cryptographic functions. A similar approach is presented by Torunoglu and others in [TC00] which explores unused transitions.

Fan provides a method where the watermark or signature is sent as a preamble of the output of the test mode [FT03]. Some ASIC circuits provide a special test mode which stimulates the core with special input patterns. To analyze the correctness of the core, the output of these input patterns are measured and compared to the correct patterns. The idea is to send the watermark sequence over the output port before the test sequence starts.

The disadvantage of these approaches is the usage of ports for signature verification. This works only if the ports are reachable. If the core is embedded into other cores, the ports of the watermarked core can be altered which falsifies or prevents the detection of the signature in the output stream. This applies also to the signature extraction sequence in the input stream.

5.2.2 Netlist Cores

To the best of our knowledge, there exist no publications on the use of additive watermarking at the level of netlist cores. In [ZT05] we presented the first two examples of how additional watermarking for netlist cores can look like. The first idea is to apply redundant logic in some paths of the core according to a signature. To verify the watermark, one can optimize the core so that the redundant logic is removed, show the differences and reconstruct the signature.

The second idea is to add false paths in the design which do not affect the following logic. The weakness of both ideas is that the design tools applied in subsequent steps use transformations which may destroy the watermark. Therefore, these ideas are not applicable.

Identification of Netlist Cores by Analysis of LUT Contents

In this approach, we do not add any signature or watermark. The core itself remains unchanged, so the functional correctness is given and no additional resources are used. We compare the content of the used lookup tables from the registered core I_{L_1} with the used lookup tables in an FPGA design I_B from the product of the accused company. If a high percentage of identical content is detected, the probability that the registered core is used is very high.

The synthesis tool maps the combinatorial logic of an FPGA core to lookup tables and writes these values into a netlist. After the synthesis step, the content of the lookup tables of a core is known, so we can protect netlist cores which are delivered at the logic level. The protection of bitfile cores at the device level is also possible.

After the core I_{L_1} is purchased, the customer can combine this core with other cores: $I_B = \mathcal{T}_{L\to B}(I_{L_1} \circ I_{L_2} \circ I_{L_3} \circ \dots)$. In the following CLB mapping step, it is possible that lookup tables are merged across the core boundaries or are removed by an optimizing transformation. This happens when different cores share logic or when outputs of the core are not used. These lookup tables cannot be found in the FPGA bitfile I_B, but experimental results show that the percentage of these lookup tables compared to the number of all lookup tables in the core is typically low for the used mapping tool (Xilinx *map*).

If a company is accused of using unlicensed cores in a product, the bitfile of the used FPGA can be extracted. After reading out the content and the positions of the lookup tables from the bitfile and comparing them with the lookup table contents

from the original core, the ownership of the core can be proven by evaluating a detector function $\mathcal{D}(I_B, I_{L_1})$.

Identifying the Core After the extraction of the content of lookup tables from a bitfile, we can compare the obtained values with the information in the netlist. The extraction of all lookup table contents from a bitfile is done as described in [Zie10]: $\mathcal{L}_B(I_B) = \{x_{B_1}, x_{B_2}, \ldots, x_{B_q}\}$. The content of the lookup tables can easily be read out from a netlist file: $\mathcal{L}_L(I_{L_1}) = \{x_{L_1}, x_{L_2}, \ldots, x_{L_r}\}$. For example, in an EDIF netlist for Xilinx FPGA devices, the lookup table contents appear after the INIT property for the lookup table instances. Unfortunately, the mapping tools do not necessarily adopt these values. The mapping tool may merge lookup tables from different cores together, convert one, two or three input lookup tables to four input lookup tables and permute the inputs to achieve a better routing.

All lookup tables of an FPGA have n_l inputs. On most FPGA architectures, lookup tables have $n_l = 4$ inputs. In a core netlist, also lookup tables with less than n_l inputs may exist. These lookup tables must be mapped onto n_l input lookup tables. If one input is unused, only half of the memory is needed to store the function and the remaining space must be filled. In the case that a function uses less inputs than the underlying technology of the FPGA provides, it is desirable to turn the unused inputs into don't cares. Intuitively, this can be achieved rather easily by replicating the function table as it is demonstrated in Figure 5.1.

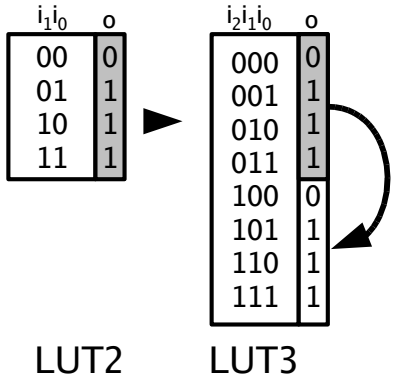


Figure 5.1: Converting a two input lookup table into a three input lookup table with unused input i_2.

The mapping tool can permute the inputs of the lookup tables, for example, to achieve a better routing. In most FPGA architectures, the routing resources for lookup table inputs are not equal, and so a permutation of the lookup table inputs

can lower the amount of used routing resources. Permutation of the inputs significantly alters the content of a lookup table. For n_l inputs, $n_l!$ permutations exist and thus up to $n_l!$ different lookup table values for one so-called *unique function*. To compare the contents of the lookup table from the netlist and the bitfile, it must be checked if one of these possible different lookup table values for one unique function is equal to the value of the lookup table in the bitfile. This is done by creating a table with all possible values of lookup tables for all unique functions (see Figure 5.2).

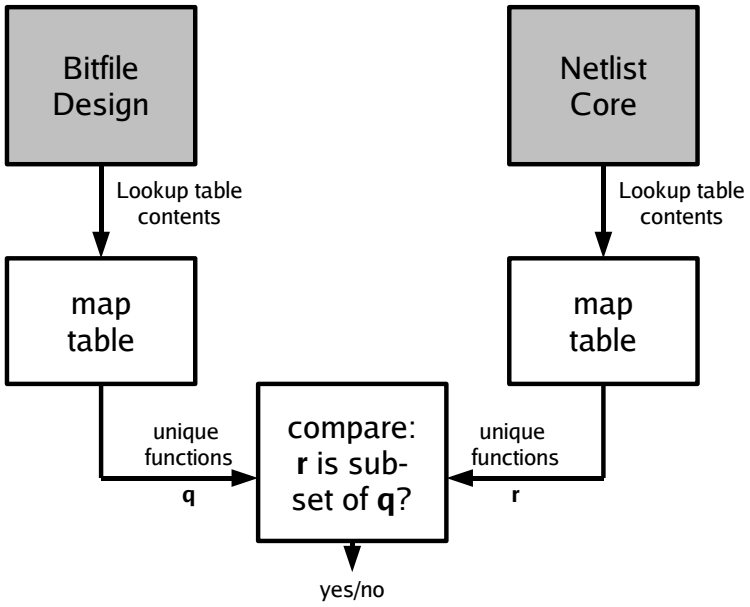


Figure 5.2: Before the lookup table contents of the bitfile and the netlist are compared, they are mapped into unique functions.

Summary We have presented a new method to identify IP cores in FPGA bitfiles. Possible transformations of the mapping tools and the effect of the robustness of the method were discussed. The experimental results show that it is possible to identify a core in the design with a high probability [ZAT06]. The identification process is based on two parameters, namely the number of found lookup tables of the core in the design and the mean distance to the core center. However, it must be taken into account that lookup tables of the core are removed by optimization tools, if parts of the core are not used because outputs are unused or constant values are applied to inputs. More information can be found in [ZAT06].

Watermarks in Functional LUTs for Netlist Cores

After watermarking bitfile cores, we now watermark netlist cores. Netlist IP cores consist of *primitive cells* (e.g., LUT4, DFF, XORCY) of a certain FPGA family which covers many different FPGA devices. For example, the whole Xilinx Virtex-4, or Altera Stratix-II family with all different FPGA sizes. This means, that one netlist core can be deployed for the whole family without changing the file. Once again, we are using the Virtex-II and II Pro family to demonstrate this approach. However, using other FPGA families should also be possible by adapting the methods to their primitive cells. Another big advantage from netlist cores over bitfile cores is that the bitfile creator (e.g., product developer) can combine different cores.

As mentioned before in Section 5.2.2, FPGAs usually consist of the same type of lookup tables with regard to the number of inputs. For example, the Xilinx Virtex-II uses lookup tables with four inputs whereas the Virtex-5 has lookup tables with six inputs. However, in common netlist cores many logical lookup tables exist, which have less inputs than the FPGA type. These lookup tables are mapped to the physical lookup tables of the FPGA. If the logical lookup table of the netlist cores has fewer inputs than the physical one, the memory space which cannot be addressed remains unused. We use this memory space to embed a watermark into functional lookup tables.

One problem of watermarking netlist cores is that the core further traverses the design flow which includes different optimization steps. Additive watermarking methods which use redundant structures or logic as watermark have the problem that the global optimization steps may detect and remove this redundancy. Todays design tools are very sophisticated to find redundant logic in a design. Even if a special redundant logic which can be used as watermark is not removed by today's tools, it is not granted that future versions or other tools may not detect and remove this logic. The challenge is to find an element or component which can be used as watermark and is not altered by design tools. For Xilinx FPGAs such elements are shift registers and memories which are implemented in lookup tables.

In some FPGA architectures (e.g., all Xilinx Virtex architectures), the lookup tables (LUTs) can also be used as a *shift register* or *distrubuted memory* [Xilf]. For example, a 4-input lookup table can be further used as a 16-bit shift register (see Figure 5.3). The content of such a shift register can be further addressed by the lookup table input ports. So, the shift register can also be used as a functional lookup table. If the lookup table is used as a LUT primitive cell, the content is interpreted as logic by the design tools and is in focus of optimization. However, if the same content is used as a shift register or memory primitive cell, the design tools do not touch the content. Using the unused memory space of functional lookup tables for watermarking without converting the lookup table either to a shift register or distributed memory turns out to be not applicable, because design flow tools identify the watermark as redundant and remove the content due to optimization. Converting the watermarked

functional lookup table into shift registers or memory cells, prevents the watermark from deletion due to optimization.

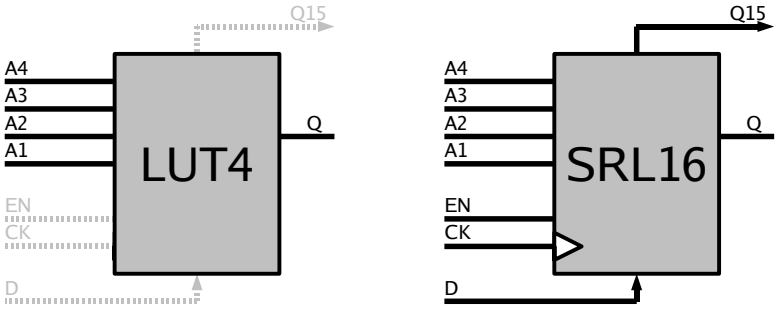


Figure 5.3: In the Xilinx Virtex architecture, a lookup table (LUT4) can also be configured as a 16-bit shift register (SRL16).

Embedding of the Watermark In this approach we are use Virtex-II Pro FPGAs and convert LUT1, LUT2, or LUT3 primitive cell which can be found in netlists of IP cores into the shift register primitive cell SRLC16E. Note that LUT1 has one input, LUT2 two and so on. LUT4 has four input and uses the whole lookup table memory for its function which make this type uninteresting for our approach. The content of the physical 4-input lookup table in an FPGA stores 16 bits. A LUT3 primitive cell uses only 8 bits, a LUT2 4 bits, and LUT1 only 2 bits out of the 16 bits. The Xilinx mapping tool *map* duplicates the used memory area to the unused area if not all inputs are needed (see Section 5.2.2). Therefore, to use the unused memory space for embedding a watermark, we must restrict the memory reachability of the function by clamping the unused inputs to constant values. In Figure 5.4, we demonstrate this idea for an AND-function, implemented by a LUT2. By clamping input A3 and A4 to zero, we can free 12 bits which can be used for carrying a watermark.

Another problem of watermarking netlist cores is that the published core is combined with other cores and undergoes further design flow steps, like the placement of the lookup table in the FPGA. Therefore, at the extraction of the watermark, we do not know the locations of the watermarks. To reduce the effort for identifying the watermarks after the extraction, we can cascade the watermarks over the *shift in* (D) and *shift out* (Q15) ports of the shift register cell. We assume, that design tools place these chains of watermarks close together which extremely simplifies the extraction of the watermarks. Furthermore, for rebuilding the watermark from the individual extracted watermarked lookup tables, the sequence is important. To bring the different watermarks, which have further different sizes according to the used original functional lookup table cell, into the right order, we concatenate the watermark bits

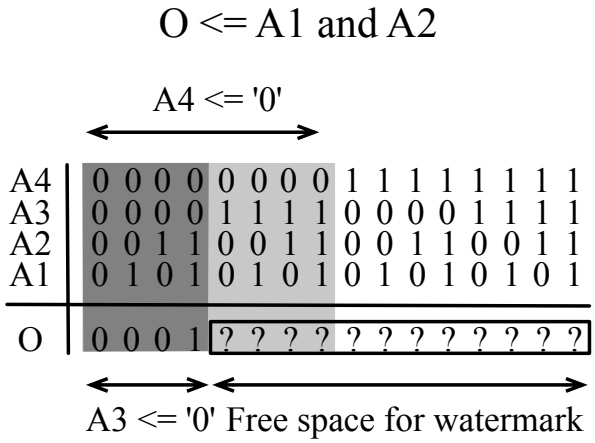


Figure 5.4: Example of implementing a two input AND gate using a four input lookup table. Addressable storage is restricted by connecting the unused inputs to zero [SZT08].

with a counter. Due to limited space, only few counter bits can be used, which results in a repetition of the counter values. Nevertheless, this method further simplifies the detection and extraction of the watermark during the verification process.

The first step of embedding a watermark is to extract all lookup tables from a given netlist core I_L: $\mathcal{L}_L(I_L) = \{lut_{L_1}, lut_{L_2}, \ldots, lut_{L_r}\}$, where L denotes the logic abstraction level used for netlist cores (see Figure 5.5). Each element lut_{L_i} denotes a lookup table primitive cell in the netlist (e.g. for Virtex-II devices, LUT1, LUT2, LUT3, or LUT4). A watermark generator $\mathcal{G}_L(\cdot,\cdot)$ must know the different lookup table cells with the functional content as well as the unique key K to generate the watermarks: $\mathcal{G}_L(K, \mathcal{L}_L(I_L)) = W_L$.

From the unique key K a secure pseudo random sequence is generated. Some or all of the extracted lookup table primitive cells are chosen to carry a watermark. Usually a core which is worth to be watermarked consists of many markable lookup tables. Transforming all of these lookup tables into shift registers restricts the optimization degree of the tools and results in non-optimal timing behavior. Therefore, only a small subset of all suitable lookup table are chosen. Note that the shift registers must never be shifted, because this alters the functional part of it. Nevertheless, we connect the clock input with the clock, but the shift enable input to ground. Now, the transformed shift registers are ordered and the first 4 bits of the free space are used for the counter value. The other bits are initialized according to the position with values from the pseudo random stream, generated from the key K. Note that the number of bits which can be used for the random stream depends on the original functional lookup table type.

The generated watermark W_L consists of the transformed shift registers: $W_L = \{srl_{L_1}, srl_{L_2}, \ldots, srl_{L_k}\}$ with $k \leq r$. The watermark embedder $\mathcal{E}_L$ inserts the water-

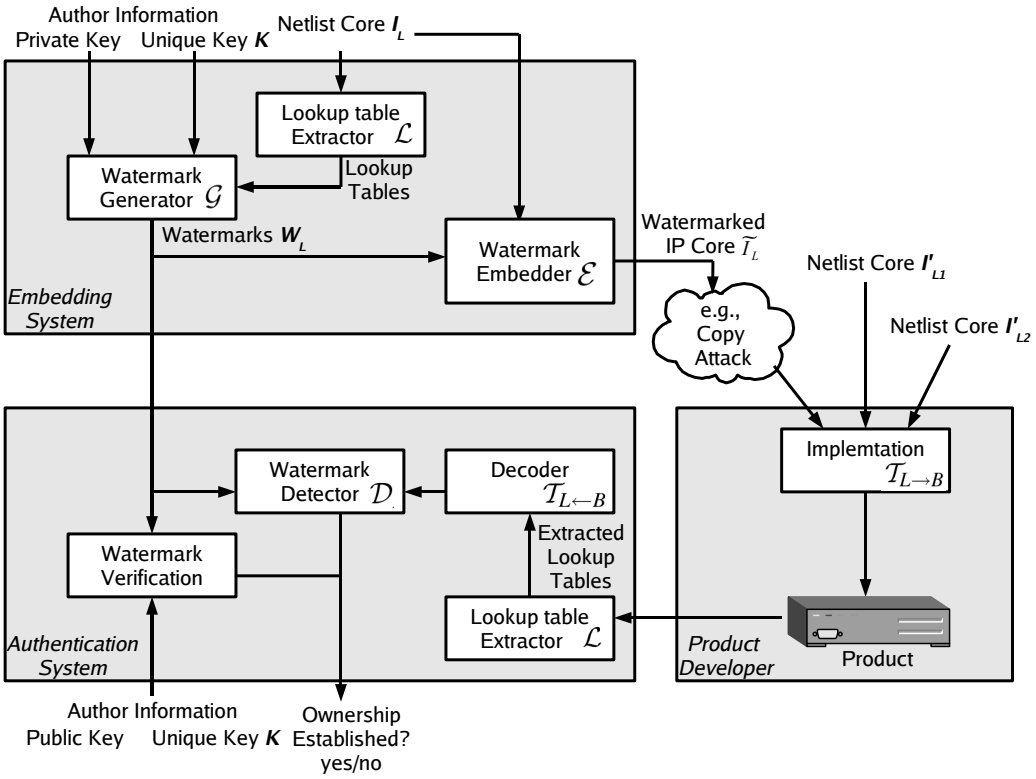


Figure 5.5: The watermarking netlist core system. In the embedding system the lookup tables are extracted from the netlist core and the watermark generator select suitable lookup table, transform it to shift register and add the watermark. The embedder insert the watermark. A product developer may obtain this watermarked netlist core an combine it with other cores into a product. The lookup tables from the product can be extracted and transformed so, that the detector can decide if the watermark is present or not.

marks into the netlist core I_L by replacing the corresponding original functional lookup tables with the shift registers: $\mathcal{E}_L(I_L, W_L) = \widetilde{I_L}$. The watermarked work $\widetilde{I_L}$ can now be published and sold.

Extraction of the Watermark The purchased core $\widetilde{I_L}$ can now be combined by a product developer with other purchased or self developed cores and implemented into an FPGA bitfile: $\widehat{I_B} = \mathcal{T}_{L\to B}(\widetilde{I_L} \circ I'_{L_1} \circ I'_{L_2} \circ \dots)$ (see Figure 5.5). An FPGA which is programmed with this bitfile $\widehat{I_B}$ may be part of a product. If the product developer is accused of using an unlicensed core, the product can be purchased and the bitfile can be read out, e.g., by wire tapping. The lookup table content and the content of the shift registers can be extracted from the bitfile: $\mathcal{L}_B(\widehat{I_B}) = \{\widehat{x}_{B_1}, \widehat{x}_{B_2}, \dots, \widehat{x}_{B_q}\}$.

The lookup table or shift register elements x_{B_i} belong to the device abstraction level B. The representation can differ from the representation of the same content in the logic abstraction level L. For example, in Xilinx Virtex-II FPGAs the encoding of the shift register differs from the encoding of lookup tables. For shift registers the bit order is reversed compared to the lookup table encodings. Therefore, the bitfile elements must be transferred to the logic level by the corresponding decoding. This can be done by the *reverse engineering operator*: $\mathcal{T}_{L\leftarrow B}(\mathcal{L}_B(\widehat{I_B})) = \{\widehat{x}_{L_1}, \widehat{x}_{L_2}, \dots, \widehat{x}_{L_q}\}$. Reverse engineering lookup table or shift register content is however very simple compared to reverse engineering the whole bitfile. Now, the lookup table or shift register content can be used for the watermark detector $\mathcal{D}_L$ which can decide if the watermark W_L is embedded in the work or not: $\mathcal{D}_L(W_L, \{\widehat{x}_{L_1}, \widehat{x}_{L_2}, \dots, \widehat{x}_{L_q}\}) = true/false$.

The detector $\mathcal{D}_L$ searches the content of the watermarked shift register W_L in the extracted lookup table contents from the bitfile. It might occur that certain watermarks will be found in more than one locations, because more of the same watermarks exist with an identical content, or a complete functional lookup table has, by chance, the value of a watermarked one. To simplify the extraction, the watermarks are chained together by the shift in and out ports. It is likely that these watermarks are placed close together. From the bitfile lookup table extraction $\mathcal{L}_B$, we also have the locations of the possible watermarks. Using these locations we can in most cases identify the right watermark, if duplicates exist. Note that this chaining approach is not mandatory, but elevates the robustness of the approach against ambiguity attacks.

After the detection of the watermark W_L inside the bitfile $\widehat{I_B}$, the watermark must be verified similar to the watermarking approach for bitfile cores proposed in Section 5.2.3.

Additional Reading More information about this method can be found in [SZT08].

Power Watermarking

This section introduces watermarking techniques, where a signature is verified over the *power consumption pattern* of an FPGA. These techniques may also be suitable for ASIC designs, however, we concentrate on FPGA designs and develop several enhancements which are exclusively related to the FPGA technology. The presented idea is new and differs from [KJJ99] and [AARR03] where the goal of using power analysis techniques is the detection of cryptographic keys and other security issues.

For power watermarking methods, the term *signature* refers to the part of the watermark which can be extracted and is needed for the detection and verification of the watermark. The signature is usually a bit sequence which is derived from the unique key for author and core identification.

First of all, a short introduction is given and the communication channel between the generation and the detection of the watermark is discussed. Next, the basis method is presented and afterwards, several enhanced methods which increase the robustness of decoding the watermark in case of external or internal disturbances are introduced. Finally, multiplex methods are discussed which enable the detection of more than one watermark if multiple watermarked cores are present in the design.

Verification over Power Consumption There is no way to measure the relative power consumption of an FPGA directly, only through measuring the relative supply voltage or current. We have decided to measure the voltage of the core as close as possible to the voltage supply pins such that the smoothing from the plane and block capacities are minimal and no shunt is required. Most FPGAs have *ball grid array* (BGA) packages and the majority of them have vias to the back of the PCB for the supply voltage pins. So, the voltage can be measured on the rear side of the PCB using an oscilloscope. The voltage can be sampled using a standard oscilloscope, and analyzed and decoded using a program developed to run on a PC. The decoded signature can be compared with the original signature and thus, the watermark can be verified. This method has the advantage of being non-destructive and requires no further information or aids than the given product (see Figure 5.6).

The consumed power of an FPGA can be divided into two parts, namely the static and the dynamic power. The static power consumption is caused by the leakage current from CMOS transistors and does not change over time if the temperature stays constant. The dynamic power consists of the power related to short circuit currents and the power required of reloading the capacities of transistors and wires. The short circuit current occurs when the *PMOS* and the *NMOS* transistors are both in conducting state for a short time during the switching activity. As shown in [SKB02], the main part of an FPGA's dynamic power results from capacity reloading. Both parts of the dynamic power consumption depend on the switching frequency [CSB92].

What happens to the core voltage, if many switching activities occur at the same time, at the rising edge of a clock signal? It is interesting to observe that the core

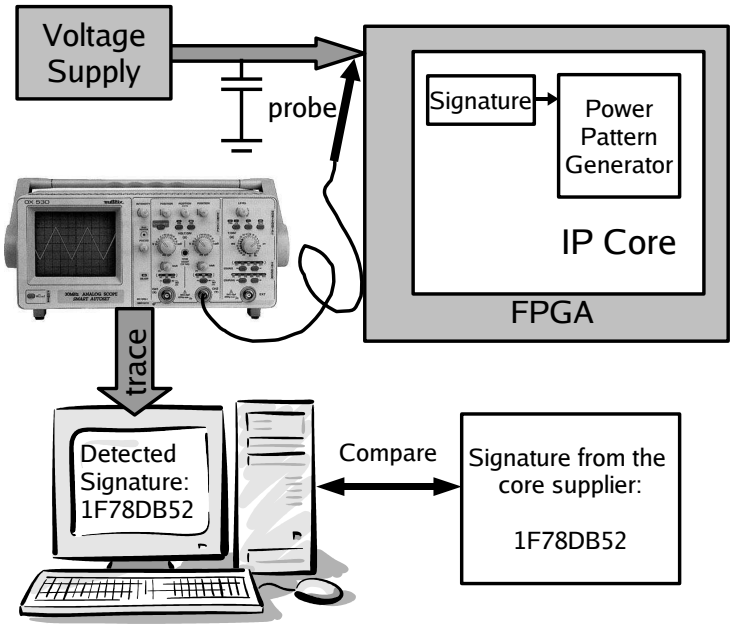


Figure 5.6: Watermark verification using power signature analysis: From a signature (watermark), a power pattern inside the core will be generated that can be probed at the voltage supply pins of the FPGA. From the trace, a detection algorithm verifies the existence of the watermark.

supply voltage drops and rises (see Figure 5.7). In the frequency domain, the clock frequency with harmonics and even integer divisions are present (see Figure 5.8). The real behavior of the core voltage depends on the individual FPGA, the individual printed circuit board and the individual voltage supply circuits.

In the following, we seek for techniques to encode a watermark such that the core voltage is subject to change once the watermark is processed within a core. In the first method, the frequency of the voltage drops shall be influenced, in the second, the amplitude of the voltage drops shall be manipulated.

In the first case, a watermark can be identified if we produce another frequency line in the spectrum of the core voltage which is not an integral multiple or a rational fraction of the clock frequency. For achieving this, we need a circuit that consumes a considerable amount of power and generates a signature-specific power pattern, and a clock which can be identified in the spectrum. The power consumer can be, for example, an additional shift register. If we would derive the clock source from the operational clock, we would not be able to distinguish the frequency line in the spectrum from operational logic. Another opportunity is to generate a clock using

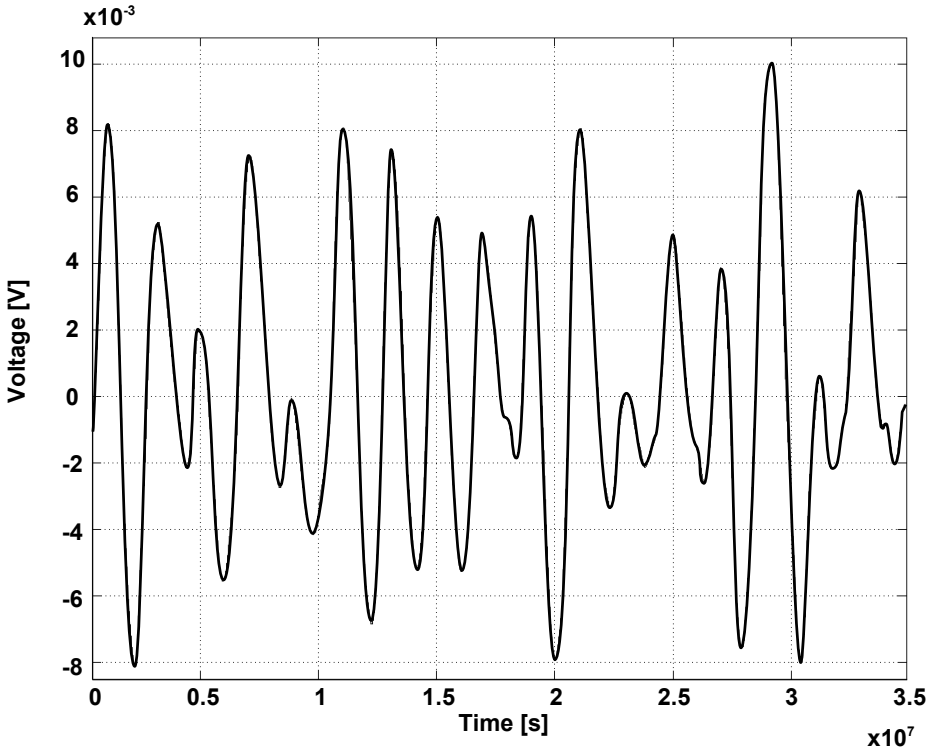


Figure 5.7: A measured voltage signal at the voltage supply pin of an FPGA. The core supply voltage drops and rises. Note that the DC component is filtered out.

combinatorial logic. This clock could be identified as a watermark, but the jitter of a combinatorial clock source might be very high, and no clean frequency line could be seen in the spectrum. This means that we need a higher additional power consumer to make the watermark readable. Another drawback is that we have only limited possibilities to encode a signature reliably in these frequency lines.

In the following approaches, we alter the amplitude of the interferences in the core voltage. The basic idea is to add a power pattern generator (e.g., a set of shift registers), and clock it either with the operational clock or an integer division thereof. Further, we control these power pattern generators according to the characteristics of the data sequence which should be sent, respectively detected. A logical '1' lets the power consumer operate one cycle (e.g., perform a shift), a '0' causes no operation. We detect higher amplitudes in the voltage profile over time corresponding to the ones and smaller amplitudes according to the zeros. Note that the amplitude for the no-operation state is not zero, because the operational logic and the clock tree is still active.

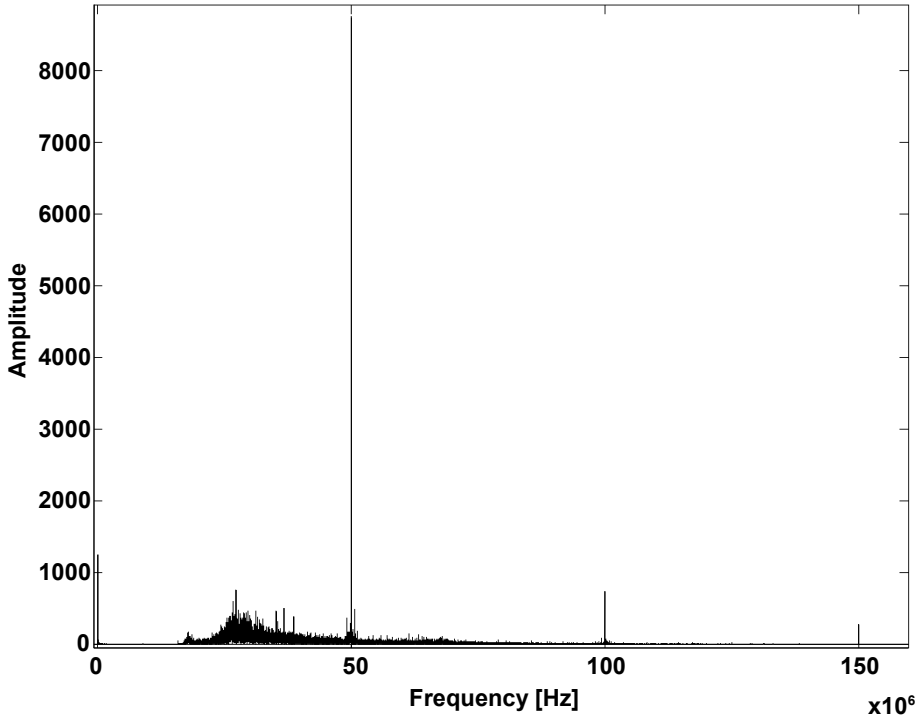


Figure 5.8: The spectrum of the measured signal in Figure 5.7. The clock frequency of 50 MHz and harmonics can be seen. Also, a peak at the half of the clock frequency is visible which is caused by switching activities from the logic.

The advantage of power watermarking methods is that the signature can easily be read out from a given device. Only the core voltage of the FPGA must be measured and recorded. No bitfile is required which needs to be reverse-engineered. Furthermore, these methods work also for encrypted bitfiles whereas methods where the signature is extracted from the bitfile fail. Moreover, we are able to sign netlist cores, because our watermarking algorithm does not need any placement information. So, also cores at this level can be protectedly watermarked.

Basic Method In this section, we describe the *basic method* for power watermarking of netlist cores. The concept, the embedding of the watermark, as well as the detection and verification procedure are described. The encoding and decoding for sending the signature through the FPGA power communication channel is relatively simple and straightforward in the basic method and will be refined later on with the enhanced methods. However, the basic concepts of embedding and the verification are very similar in all methods.

For power watermarking, two shift registers are used, a large one for causing a recognizable signature-dependent power consumption pattern, and a shift register storing the signature itself (see Figure 5.6 in Section 5.2.2). The signature shift register is clocked by the operational clock and the output bit enables the power pattern generator. If the output bit is a '1', the power pattern register will be shifted at the next rising edge of the operational clock. At a '0', no shift is done. Therefore, the channel encoding is $\mathcal{Z} = \{(\gamma, 1, 1), (\bar{\gamma}, 1, 1)\}$. To avoid interference from the operational logic in the measured voltage, the signature is only generated during the reset phase of the core.

As mentioned before in Section 5.2.2, a shift register can also be used as a lookup table and vice versa in many FPGA architectures (see Figure 5.3 in Section 5.2.2). A conversion of functional lookup tables into shift registers does not affect the functionality if the new inputs are set correctly. This allows us to use functional logic for implementing the power pattern generator. The core operates in two modes, the *functional mode* and the *reset mode*. In the functional mode, the shift is disabled and the shift register operates as a normal lookup table. In the reset mode, the content is shifted according to the signature bits and consumes power which can be measured outside of the FPGA. To prevent the loss of the content of the lookup table, the output of the shift register is fed back to the input, so the content is shifted circularly. When the core changes to the functional mode, the content must be shifted to the proper position to have a functional lookup table for the core.

The amplitude of the generated power signature depends on the number and content of the converted lookup tables. It will be assumed that the transitions between zeros and ones in the bit pattern of the lookup table contents are sufficient to produce a recognizable pattern on the supply voltage. Experimental results in [Bau08] show that, on average, 8 of maximal 16 transitions are generated in functional 4 input lookup tables of example cores if the content will be shifted.

To increase the robustness against removal and ambiguity attacks, the content of the power consumption shift register which is also part of the functional logic can be initialized shifted. Only during the reset state, when the signature is transmitted, the content of the functional lookup table can be positioned correctly. So, normal core operation cannot start before the signature was transmitted completely. The advantage is that the core is only able to work after sending the signature. Furthermore, to avoid a too short reset time in which the watermark cannot be detected exactly, the right functionality will only be established if the reset state is longer than a predefined time. This prevents the user from leaving out or shorten the reset state with the result that the signature cannot be detected properly.

The signature itself can be implemented as a part of the functional logic in the same way. Some lookup tables are connected together and the content, the function of the LUTs, represents the signature. Furthermore, techniques described in Section 5.2.2 can be used to combine an additional watermark and the functional part in a single lookup table if not all lookup table inputs are used for the function. For example,

LUT2 primitives in Xilinx Virtex-II devices can be used to carry an additional 12-bit watermark by restricting the reachability of the functional lookup table through clamping certain signals to constant values. Therefore, the final sending sequence consists of the functional part and the additional watermark. This principle makes it almost impossible for an attacker to change the content of the signature shift register. Altering the signature would also affect the functional core and thus result in a corrupt core.

The advantages of using the functional logic of the core also as a shift register are a reduced resource overhead for watermarking and the robustness of this method, because these shift registers are embedded in the functional design and it is hard, if not impossible, to remove the shift registers without destroying the functionality of the core. Furthermore, our watermarking procedure is difficult to be detected in a netlist file, because the main part of the required logic for signature creation depends on the functional logic for the proper core. Another benefit is that our watermark cannot be removed by an optimization step during the mapping into CLBs (Configurable Logic Blocks). Nevertheless, if an attacker had special knowledge of the watermarking method and of the EDIF netlist format, he may reverse-engineer the alteration of the embedding algorithm and remove or disable the sending method. This can be avoided by initializing the power pattern register with shifted lookup table contents (see above). If sending of the signature is prevented, the core will not function properly.

Embedding of the Watermark In this section, we describe the procedure of watermarking a core. The first step is to generate the watermark W_L for embedding at the logic abstraction layer L. As described in the last section, the watermark is a bit sequence, consisting either of random choice bits, of partly functional bits of lookup tables, or completely of functional bits. The watermark generation procedure depends on the sequence type.

If only random choice bits are used, the watermark generated needs only the unique key K which identifies the author of the core: $\mathcal{G}_L(K) = W_L$. The pseudo random output can be split into different shift registers: $W_L = \{w_{L_1}, w_{L_2}, \ldots, w_{L_m}\}$. The number of used shift registers m depends on the strength of the generated signature and the FPGA architecture. For example, a 128-bit signature can be stored in the Virtex-II architecture in $m = 8$ shift registers.

If the content of functional lookup tables should be used as signature, the first step is to extract all lookup tables form the netlist core: $\mathcal{L}_L(I_L) = \{lut_{L_1}, lut_{L_2}, \ldots, lut_{L_r}\}$. The watermark generator $\mathcal{G}_L$ searches for suitable functional lookup tables, transforms these into shift registers and either adds the watermark bits form the pseudo random sequence $\mathcal{G}_L(K, \mathcal{L}_L(I_L)) = W_L$, or only uses the lookup table content as signature: $\mathcal{G}_L(\mathcal{L}_L(I_L)) = W_L$.

The watermark embedder $\mathcal{E}_L(I_L, W_L) = \widetilde{I_L}$ consists of two steps. First, the core I_L must be embedded in a wrapper which contains the control logic for emitting the signature. This step is done at the register-transfer level before synthesis. The second step is at the logic level after the synthesis. A program converts suitable lookup tables (for example LUT4 for Virtex-II FPGAs) into shift registers for the generation of the power pattern and attaches the corresponding control signal from the control logic in the wrapper (see Figure 5.9).

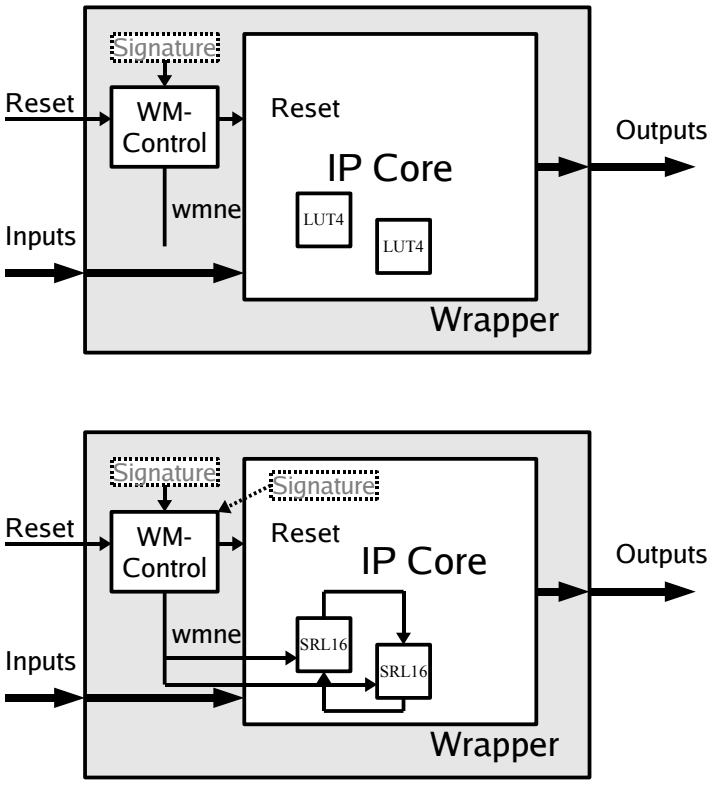


Figure 5.9: The core and the wrapper before (above) and after (below) the netlist alternation step. The signal "wmne" is an enable signal for shifting the power pattern generator shift register.

The wrapper contains the control logic for emitting the watermark and the shift register, holding the signature. If functional lookup tables are used for implementing the signature shift register, we add or convert this shift register in the second step so that the wrapper contains only the control logic. Some control signals have no sink yet, because the sink will be added in the second step (e.g., the power pattern generator shift register). So we must use synthesis constraints to prevent the synthesis

tool from optimizing these signals away. The ports of the wrapper are the same for the core, so we can easily integrate this wrapper into the hierarchy. The control logic shifts the signature shift register, while the core is in reset state. Also, the power pattern shift register is shifted corresponding to the output of the signature shift register. If the reset input of the wrapper gets inactive, the function of the core cannot start at the same cycle, because the positions of the content in the shift register are not in the correct state. The control logic shifts the register content into the correct position and leaves the reset state to start the normal operation mode.

The translation of lookup tables of the functional logic into shift registers is done at the logic level. At Xilinx Virtex-II FPGAs, the usage of a LUT4 as a 16-bit shift register (SRL16) is only possible if the LUT4 is not part of a multiplexer logic, because the additional shift logic and the multiplexer share common resources in a slice. Also, if the lookup table is a part of an adder, the mapping tool splits the lookup table and the carry chain. In these two cases, additional slices would be required, so we do not convert these lookup tables into shift registers.

The embedding procedure for Virtex-II netlist cores is done by a program which parses an EDIF netlist and writes back the modified EDIF netlist. First, the program reads all LUT4 instances and only select those that are not a "MUXF5", a "MUXCY" or an "XORCY". Then, the instances are converted to a shift register (SRL16), if required, initialized with the shifted value and connected to the clock and the watermark enable (wmne) signal according to Figure 5.9. Always two shift registers are connected together to rotate their contents. Finally, the modified netlist is created. The watermarked core $\widetilde{I_L}$ is now ready for purchase or publication.

Detection Algorithm A company may obtain an unlicensed version of the core $\widehat{I_L}$ and embeds this core in a product: $\widehat{I_P} = \mathcal{T}_{L\to B}(\widehat{I_L} \circ I'_{L_1} \circ I'_{L_2} \circ \dots)$. If the core developer has a suspicious fact, he can buy the product and verify that his signature is inside the core using a detection function $\mathcal{D}_P(\widehat{I_P}, W_L) = true/false$.

Detecting the basic power watermark, the measured voltage will be probed, digitized and decoded by a signature detection algorithm (see Figure 5.10). To decode the digitalized voltage signal, the sampling rate, the clock frequency of the shifted signature and the bit length of the signature is needed. The clock frequency can be extracted using the *Fast Fourier Transformation* (FFT) of the measured signal. Our detection algorithm consists of five steps: *down sampling*, *differential step*, *accumulation step*, *phase detection* and *quantization* (see Figure 5.10). After successful extraction, the decoded signature can be compared to the signature inside the watermark W_L to establish the ownership. Furthermore, the signature must be verified by cryptographic methods with the author's unique key K.

As mentioned before, the main characteristic caused by a switching event is the drop of the voltage followed by a subsequent overshoot. This results in extreme slopes. The basic method detection algorithm can find each rising edge as follows:

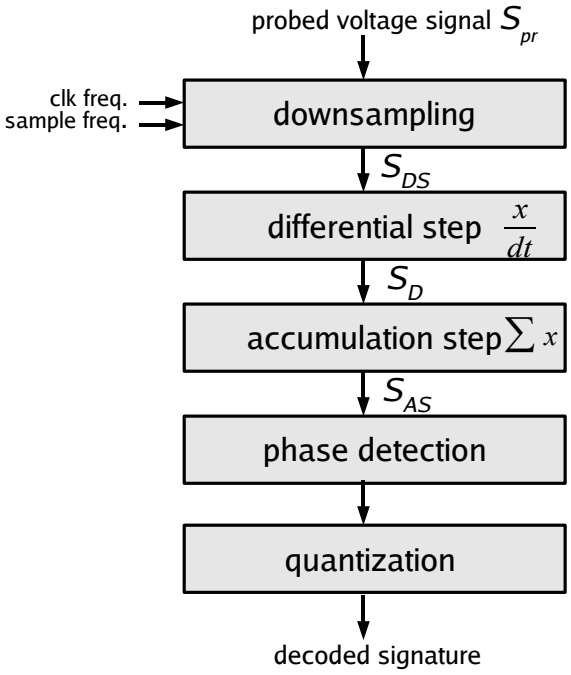


Figure 5.10: The five steps of the watermark detection algorithm: downsampling, differential and accumulation step, phase detection and finally quantization.

First, the measured signal will be sampled down from the recorded sample rate to the quadruple of the clock frequency, so each signature bit is represented by four samples. Then, the discrete derivative of the signal will be calculated. This transforms the rising edges of the switching events into peaks. The easiest way to calculate the discrete derivative at a discrete point in time $S_D[k]$ is to take the difference of two subsequent samples over time (see Figure 5.11).

$$S_D[k] = S_{DS}[k] - S_{DS}[k-1], \tag{5.1}$$

where S_{DS} is the down sampled probed voltage signal and k denotes the sample index.

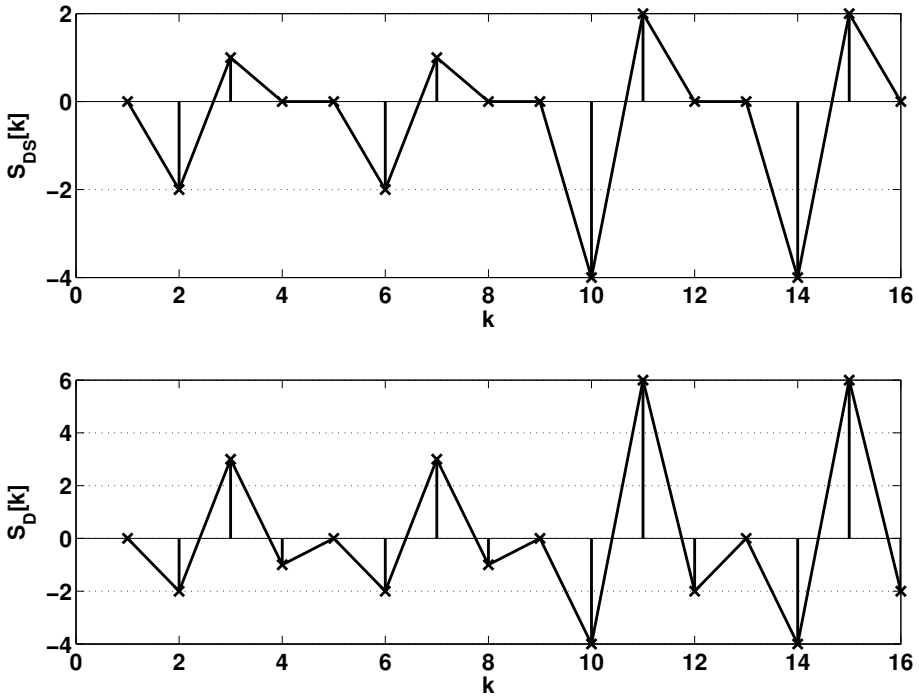


Figure 5.11: An example voltage signal which represents the signature "0011" (above). The example voltage signal after the differentiation step (below).

Since the signature is repeated many times during the reset state, the signal can be accumulated and averaged to reduce the noise level. To accumulate the coherent pattern, we need to know the bit length of the signature. If we record a longer signal sequence, we can accumulate more patterns and reduce noise as well as switching events which do not belong to the power consumption register of the watermarking algorithm. The disadvantage is that we would need a longer time for the reset phase.

After this third step, we have a signal in which each signature bit is represented by four samples. But only one sample carries the information of the rising edge. Since the measurement is not synchronized with the FPGA clock, the phase (position) of the relevant sample of a bit is unknown. We divide the signal into four new signals, where one signature bit is represented in one sample. The four signals have a phase shift of 90^o to each other. Let

$$S_{AS}[k], \qquad k = 0,1,..,4m-1 \tag{5.2}$$

denote the sampled voltage signal after the accumulation step where m is the length of the signature. Then, we obtain the four following phase shifted signals

$$S_0 = S_{AS}[4k], \qquad k = 0,1,..,m-1 \tag{5.3}$$
$$S_{90} = S_{AS}[4k+1], \qquad " \tag{5.4}$$
$$S_{180} = S_{AS}[4k+2], \qquad " \tag{5.5}$$
$$S_{270} = S_{AS}[4k+3], \qquad " \tag{5.6}$$

where S_{AS} is the accumulated signal and S_0, S_{90}, S_{180}, and S_{270} are the phase signals (see Figure 5.12).

We are able to extract the right phase of the signal if we calculate the mean value of each phase-shifted signal. The maximal mean value corresponds to the correct phase, because the switching event should cause the greatest rising edge in the signal.

Now, we have a signal in which each sample is represented by the accumulated switching activities of one bit of the signature. The decision if the sample corresponds to a signature bit '1' or '0' can be done by comparing the sample value with the mean value of the signal. If the sample value is higher than the mean value, the algorithm decides a '1', in the other case a '0'.

Robustness Analysis The most common attacks against watermarking are *removal, ambiguity, key copy*, and *copy attacks*. Once again, key copy attacks can be prevented by asymmetric cryptographic methods, and there is no protection against copy attacks.

Removal attacks most likely take place on the logic level instead of the device level where it is really hard to alter the design. The signature and power shift registers as well as the watermark sending control logic in the wrapper are mixed with functional elements in the netlist. Therefore, they are not easy to detect. Even if an attacker is able to identify the sending logic, a deactivation is useless if the content of the power shift register is only shifted into correct positions after sending the signature. By preventing the sending of the watermark, the core is unable to start. Another possibility is to alter the signature inside the shift register. The attacker may analyze the netlist to find the place were the signature is stored. This attack is only successful

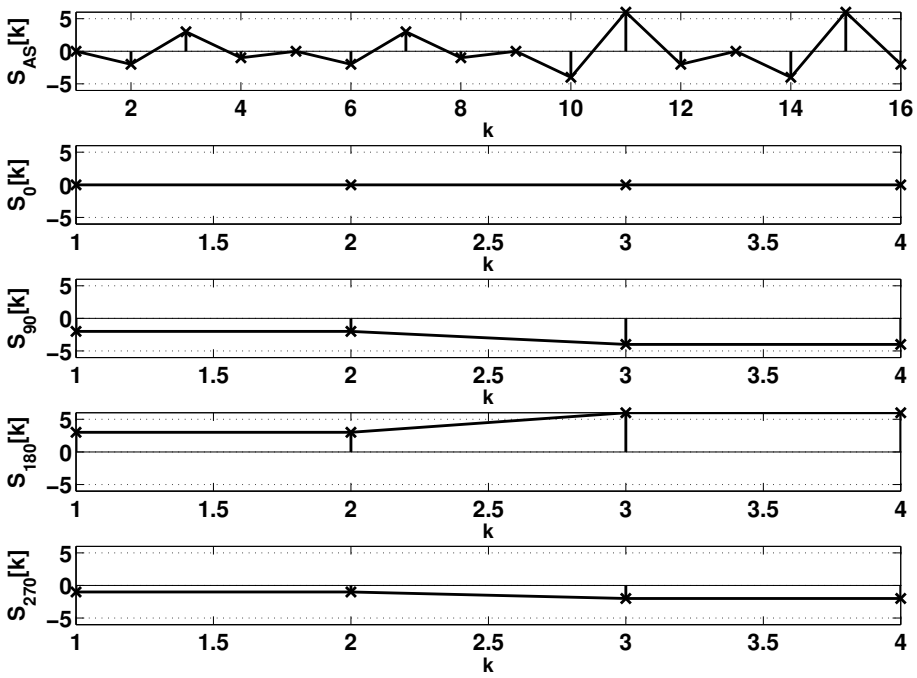


Figure 5.12: The example voltage signal after the accumulation step (above) and the four phase shifted signals (below). Here, S_{180} corresponds to the right phasing.

if there is no functional logic part mixed with the signature. By mixing the random bits with functional bits, it is hard to alter the signature without destroying the correct functionality of the core. Therefore, this watermark technique can be considered as resistant against removal attacks.

In case of *ambiguity attacks*, an attacker analyses the power consumption of the FPGA in order to find a fake watermark, or to implement a core whose power pattern disturbs the detection of the watermark. In order to trustfully fake watermarks inside the power consumption signal, the attacker must present the insertion and sending procedure which should be impossible without using an additional core. Another possibility for the attacker is to implement a *disturbance core* which needs a lot of power and makes the detection of the watermark impossible. In the next sections, *enhanced robustness encoding methods* are presented which increase the possibility to decode the signature, even if other cores are operating during the sending of the signature. Although a disturbance core might be successful, this core needs area and most notably power which increases the costs for the product. The presence of a disturbance core in a product is also suspicious and might lead to further investigation if a copyright infringement has occurred. Finally, the attacker may watermark another

core with his watermark and claim that all cores belong to him. This can be prevented by adding a hash value of the original core without the watermark to the signature like in the bitfile watermarking method for netlist cores.

Enhanced Robustness Encoding Method Experimental results have shown that the decoding of the signature with the basic method works well, but on some targets, problems occur in the decoding of signatures with long runs of '1' followed by many zeros, like "1111111100000000". The problem is intersymbol interference, because the transmitting slot for one symbol in the basic method might be smaller than the symbol length. For the first eight bits, we see a huge amplitude in Figure 5.13. Then, a phase in which the amplitude is faded out is observed. The phase can last many clock cycles and may lead to wrong detection results of the following bits.

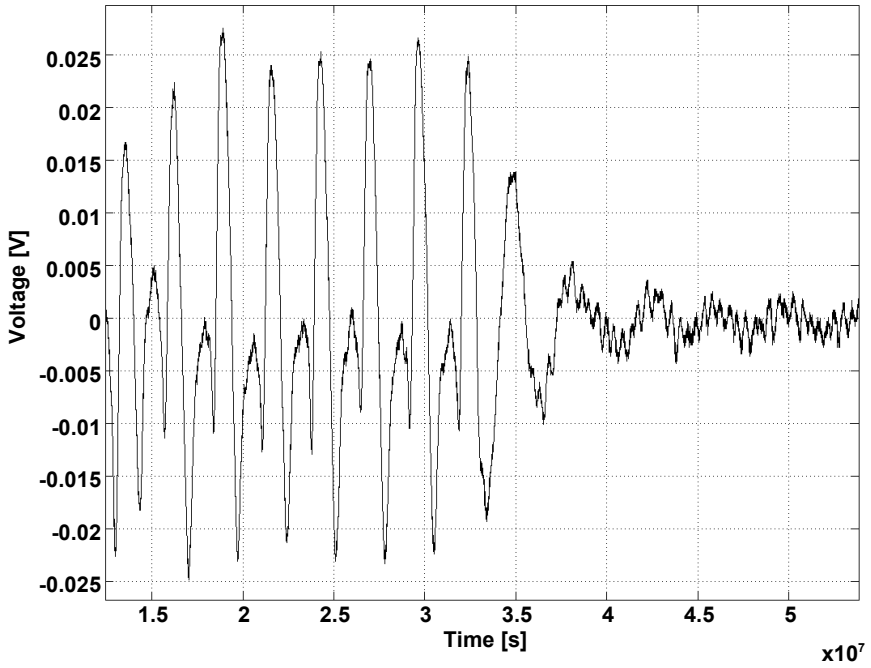


Figure 5.13: Measured voltage supply signal when sending "FFFF0000" with a large power pattern generator shift register.

This fading out amplitude belongs to an overlaid frequency which might be produced by a resonance circuit that consists of the capacitances and resistances from the power supply plane and its blocking capacitances. This behavior is dependent on the printed circuit board and the power supply circuit.

To avoid such a false detection, the transmission time of one symbol is extended by the time of the swing out of the printed circuit board by sending the same signature

bit multiple times: $\mathcal{Z} = \{(\gamma, 1, \omega), (\bar{\gamma}, 1, \omega)\}$. The repetition rate for each signature bit is ω clock cycles. If we connect two SRL16 together, one period for this shift register needs 32 clock cycles. If the reset phase ends and we have finished sending one bit, the content in the shift register which also represents a part of the logic of the core is in the correct position.

The detection algorithm differs for this method. First, the signal will be sampled down and the approximate derivation will be calculated as in the original method (see Section 5.2.2). Now, we average the signal to suppress the noise. Here, the length of one signature word is the length of the signature (m) multiplied by the number of times each bit is sent (ω).

$$S_D[k], \qquad k = 0, 1, .., K_{max} - 1 \tag{5.7}$$

$$n_s = \left\lfloor \frac{K_{max}}{4\omega \cdot m} \right\rfloor, \tag{5.8}$$

$$S = \frac{1}{n_s} \sum_{i=0}^{n_s - 1} D[4\omega \cdot m \cdot i, .., 4\omega \cdot n \cdot i + 4\omega \cdot m - 1], \tag{5.9}$$

Here, S_D is the voltage signal after the differential step with index k and n_s being the number of repetitions of the pattern in S_D.

The phase detection of the shift clock is the same as in the original method (see Section 5.2.2), but we also need the position p where a new signature bit starts. This is done in a loop to detect this position. In the beginning, we assume that the starting position is the beginning of our trace ($p = 0$). First, we accumulate ω successive values where ω is the repetition of one bit:

$$S_p[j] = \sum_{i=0}^{\omega - 1} S_\phi[i + p + \omega j], \qquad j = 0, 1, .., m - 1 \tag{5.10}$$

Here, S_ϕ denotes the signal after the phase detection step. Now, we subtract the mean value and generate the absolute value and calculate the sum of it.

$$F_p = \sum_{i=0}^{n-1} \left| S_p[i] - \frac{1}{n} \sum_{j=0}^{n-1} S_p[j] \right| \tag{5.11}$$

F_p identifies how good our signature bit starting position p fits the real position. Now, we shift our trace one value ($p = 1$) and calculate the fitting value again, and so on. This is done ω times. The starting position with the best fitting value will be used.

The decoding of the watermark signature is done like in the basic method (see Section 5.2.2) by comparing the sample values with the mean value of the samples.

BPSK Detection Method The enhanced robustness method introduced above works well, but if other cores with the same clock frequency have a very high toggle rate in the reset phase of the watermarked core, the quality of decoding may suffer. In the worst case, the decoding is not possible, because the watermarked signal is too weak in contrast to the interferences with the same frequency generated by the high toggle rate of the other cores.

To enhance the robustness of decoding our transmit signal in case of interferences with the same frequency, we combine a new sending scheme with a new detection algorithm. The basic idea is to shift the carrier frequency of our watermarking signal away from the clock frequency of the chip, where we expect most of the interferences to occur.

We introduce a new binary signal S_{BPSK} with the frequency f_{BPSK}, where the signature bits are modulated using *Binary Phase Shift Keying* (BPSK) modulation. Using BPSK modulation, each value of a signature bit $(0,1)$ is represented by a phase (usually $0°$ and $180°$). Practically, by sending a '0', the carrier signal is not altered, and is inverted by sending a '1' (see Figure 5.14).

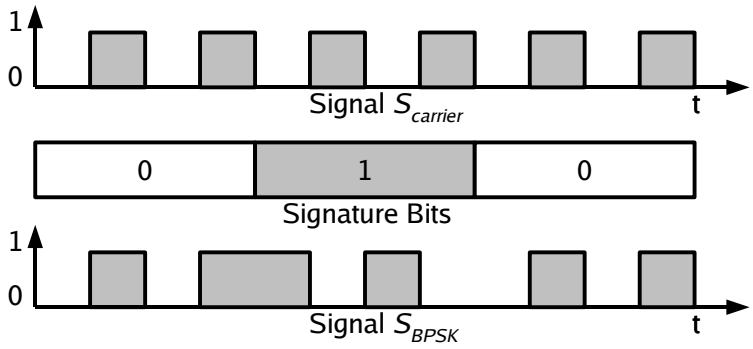


Figure 5.14: Shown is a carrier signal $S_{carrier}$ and the *BPSK* modulated signal S_{BPSK}. The signature bit value '0' is decoded with $0°$ and the value '1' is decoded with $180°$.

We generate the new frequency f_{BPSK} by an *on-off keying* (OOK) modulation, a binary *amplitude modulation* (AM) of the clock frequency f_{clk}. So, the frequency f_{BPSK} must be a rational fraction of the clock frequency f_{clk}. However, interferences from working cores have also an impact here, because these frequencies could also be produced by working cores with different toggle rates. The measurements suggest that frequencies $f = \frac{f_{clk}}{2^n}$ may have high interference from working cores due to whereas other frequencies have lower interference. The interferences decrease as well at lower derived frequencies. In the following, we choose $f_{BPSK} = \frac{f_{clk}}{10}$ as our carrier frequency.

To generate the new watermark signal, the power pattern generator is driven by the signal S_{BPSK} and performs the OOK modulation. The encoding scheme for the signal S_{BPSK} is: $\mathcal{Z} = \{(\gamma, 1, \omega), (\bar{\gamma}, 1, \omega)\}$, where ω is chosen 10 in our case. To send the signal S_{BPSK} for one period, we first send five ones (the power pattern shift register is shifted five times) and then five zeros (the power pattern shift register is not shifted) in case the signature bit is '1'. If the signature bit is '0', first five zeros and then five ones are sent (see Figure 5.15). For each signature bit, we repeat this period 32 times to ensure that the content of the power pattern shift registers which are also functional lookup tables are in the correct positions after sending one signature bit. Repetition allows to detect the signature with a higher probability. The decreased bit rate results in a smaller bandwidth for our watermarking signal. Using this method, we need more time to send the signature than the previously presented methods. The signature bit rate f_{wm} is:

$$f_{wm} = \frac{f_{BPSK}}{32} = \frac{f_{clk}}{10 \cdot 32} = \frac{f_{clk}}{320} \tag{5.12}$$

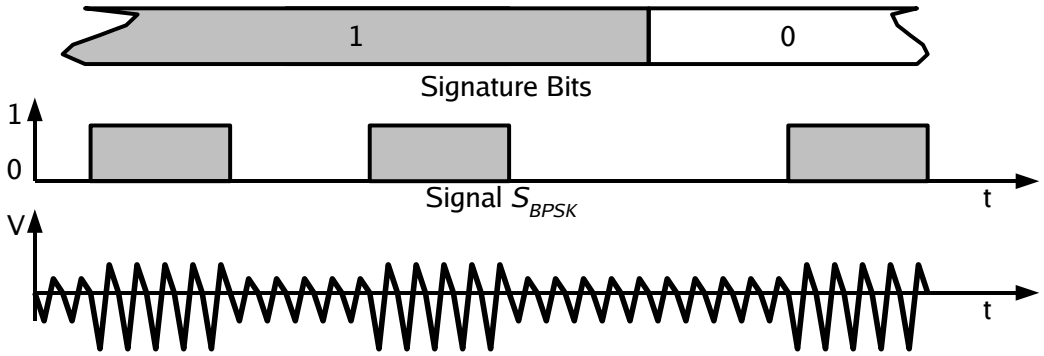


Figure 5.15: The signal S_{BPSK} is the *BPSK* modulated signal of the signature above. The signal below is the voltage signal which is the OOK modulated signal of S_{BPSK}. This figure also illustrate the different frequencies.

The watermark control inside the wrapper (see Section 5.2.2) is altered to control the power pattern generator in this way. Only few additional resources are used to implement this enhanced watermark protocol.

If we look at the spectrum of the recorded signal (see Figure 5.16), we detect the clock frequency f_{clk} and two side bands from the OOK modulation $f_{clk} - f_{BPSK}$ and $f_{clk} + f_{BPSK}$.

The detection algorithm for this method is different from the previous methods. Only the first (down sampling) and the last steps (quantization) are identical (see

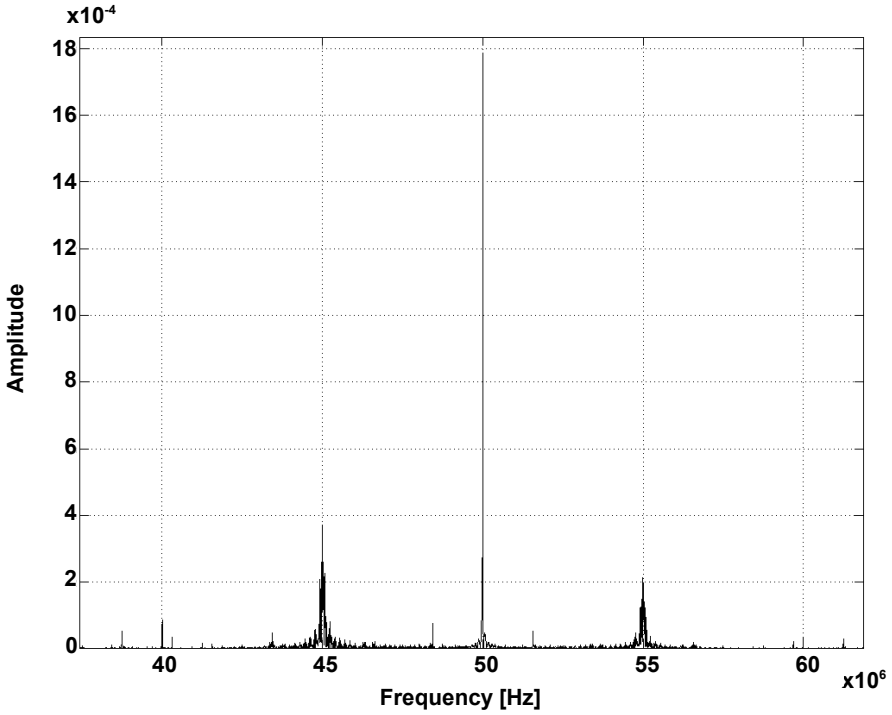


Figure 5.16: The spectrum of a measured signal. The clock frequency of $50MHz$ and the two side bands of the modulated signal S_{BPSK} are shown at $45MHz$ and $55MHz$.

Figure 5.17). After down sampling, the two side bands of the carrier signal are mixed down into the base band (S_{sb1} and S_{sb2}) and are combined (S_{cc}) as follows:

$$S_{DS}[k], \qquad k = 0, 1, .., K_{max} - 1 \tag{5.13}$$

$$S_{sb1}[k] = S_{DS}[k] \cdot e^{-j2\pi \cdot (\frac{1}{4} - \frac{1}{40}) \cdot k}, \tag{5.14}$$

$$S_{sb2}[k] = S_{DS}[k] \cdot e^{-j2\pi \cdot (\frac{1}{4} + \frac{1}{40}) \cdot k}, \tag{5.15}$$

$$S_{cc}[k] = S_{sb1} + S_{sb2}, \tag{5.16}$$

where S_{DS} is the voltage signal after down sampling with index k. The clock frequency is $f_{clk} = \frac{1}{4} \cdot f_{sample}$, and the frequency $f_{BSPK} = \frac{1}{10} \cdot f_{clk} = \frac{1}{40} \cdot f_{sample}$. The sample frequency of the recorded voltage signal is f_{sample}. After low pass filtering of S_{cc}, we get the complex carrier signal S_{BPSK} (see Figure 5.18).

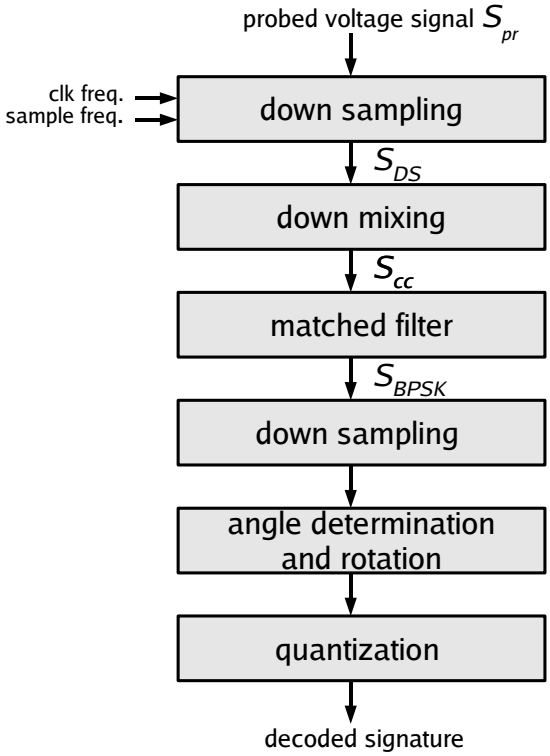


Figure 5.17: The different steps of the BPSK detection algorithm.

S_{cc} is filtered using a *matched filter* to obtain the limits of one signature bit and the correct sample point. All samples of S_{BPSK} which belong to one signature bit are summed up into this sample point by the matched filter. At the down sampling step, only these points are used to represent the signature bits. Now, the angle of the

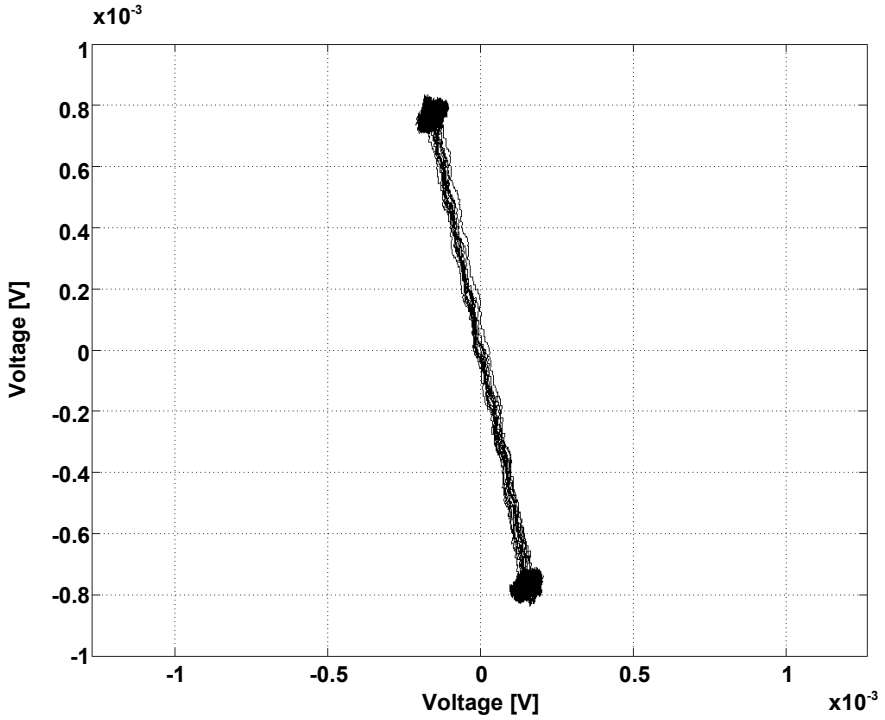


Figure 5.18: The *constellation diagram* of the down mixed complex signal S_{BPSK}. Here, the two different BPSK constellation points for the signature bit '1' and '0' are shown.

signal is calculated from the signature bit with the highest amplitude, and the signal is rotated into the real plane. From the real valued signal, the value of the bits and the quality of the signal are determined similar to the other detection algorithms (see Section 5.2.2).

The advantage of the BPSK method is its robustness with respect to interferences coupled with the clock frequency. The disadvantages are the longer reset phase and the fact that we can only detect bit value changes and not the signature bit value directly due to the BPSK modulation. Using proper encoding methods and preambles, the bit values can be reconstructed.

Additional Reading This section gave an overview of multiplexing methods suitable for power watermarking with many watermarked cores inside an FPGA sending simultaneous signatures. More details about these methods can be found in [ZT06, ZT08b, Zie10, Bau08].

5.2.3 Bitfile Cores

The approach of Lach and others watermarks bitfile cores by encoding the signature into *unused lookup tables* [LMSP98]. At first, the signature will be hashed and coded with an error correction code (ECC) to be able to reconstruct the signature even if some lookup tables are lost, e.g., during tampering. After the initial place and route pass, the number of unused lookup tables will be determined. The signature is split into the size of the lookup tables and additional LUTs are added to the design. Then, the place and route process will be started again with the watermarked design. Later, the approach was improved by using many small watermarks instead of a single large one [LMSP99]. The size of the watermarks should be limited by the size of a lookup table. The advantage is that small watermarks are easier to search for, and for verification, only a part of all of watermark positions must be published. With the knowledge of the published position, the watermark can be easily removed by an attacker. At the verification process, only a few positions of the watermark need to be used to establish the ownership. A second improvement is that a *fingerprinting technology* is added to the approach that enables the owner to see which customer has given the core away [LMSP01]. The fingerprinting technology is achieved by dividing the FPGA into tiles. In each tile, one lookup table is reserved for the watermark. The position of the mark in the tile encodes the fingerprint. For verification, it is possible to read out the content of the lookup table from a bitfile. So, these methods are easy to verify. It's more difficult to determine the position of the watermark in a tile, but it's still generally possible. However, if an attacker knows the position of the watermark, it is easy to overwrite it.

Saha and others present a watermarking strategy for FPGA bitfiles by subdividing the lookup table locations into sets of 2×2 tiles [SSK07]. The number of used lookup tables in a set is used as signature. From an initial level, additional lookup tables are added to achieve the fill level according to the signature. The input and output are connected to the *don't care inputs* of the neighboring cells. Kahng and others show in [KLMS$^+$98] that the configuration of the multiplexer of unused CLB outputs in FPGA bitfiles can carry a signature. The signature is embedded after the bitfile creation and by knowing the encoding of the bitfile. These configuration bits can be later extracted to verify the signature.

Van Le and Desmedt show that these additional watermark schemes for bitfile cores can be easily attacked by reverse engineering, watermark localization, and subsequent watermark removal [LD03]. A simple algorithm is introduced which identifies lookup tables or multiplexers whose outputs are not connected to any output pins. However, these attacks are only successful if reverse engineering of the bitfile is possible and the costs of reverse engineering are not too high.

Finally, Kean and others present a watermarking strategy where a signature is embedded into an FPGA bitfile core or design [KMM08]. The read out of the signature

is done by measuring the temperature of the FPGA. This approach is commercially available as the product *DesignTag* from *Algotronix*.

Watermarks in LUTs for Bitfile Cores

In this section, we introduce our first watermarking technique for IP cores. The easiest way to watermark an FPGA design is to place the watermarks into the bitfiles. Bitfiles are very inflexible because they were specifically generated for a certain FPGA device type, however, it makes sense to sell bitfile IP cores for common development platforms which carry the same FPGA type. Usually, a bitfile core is a whole design which is completely placed and routed and therefore ready to use. There also exist partial bitfiles which carry only one core. These partial bitfile cores can be combined into one FPGA which increases the flexibility of these cores and therefore may increase the trade possibilities.

In this approach, we hide our signature inside unused lookup tables. It is very unlikely that a design or bitfile core uses all available lookup tables in an FPGA. Before a design reaches this limit, the routing resources are exhausted and the timing degenerates rapidly. Therefore, many unused lookup tables exist in usual designs. On the other hand, lookup table content extraction is not difficult. Using lookup tables for hiding a watermark which are far away from the used ones, makes it easier for an attacker to identify and remove them. Even if an attacker is able to extract all lookup tables from a bitfile core, the lookup tables which carry the watermark should not be suspicious.

In Xilinx devices, lookup tables are grouped together with *flip-flops* into slices. A slice usually consists more than one lookup table, e.g., the Virtex-II and Virtex-II Pro devices have two lookup tables in one slice. It is not unusual that only one lookup table of a slice is used and the other remains unused. Hiding a watermark in the unused lookup table of a used slice is less obvious than using lookup tables in unused slices. Even if the attacker is able to extract the lookup table content and coordinates, the watermarks are hard to detect.

The extraction and verification of the watermark is rather easy. First of all, the content and the coordinates of all used lookup table of the core are extracted. For the verification there exist two approaches: a *blind approach* and a *non-blind approach*. In the blind approach, the watermarks are searched in all extracted lookup table contents, whereas in the non-blind approach the location of the watermarks are known. Having the right coordinates, the watermarked lookup table content can be directly compared to the watermarks of the core developer. The locations of the watermarks delivered from the core developer, however, should be kept secret, because otherwise it is very easy for an attacker to remove the marks.

Concept In the following, the watermark approach is described in detail. For watermarking a bitfile core, the watermarks which should be embedded into the unused

lookup tables must be generated. This is done by the watermark generator function: $\mathcal{G}_B(K) = W_B$. The generator needs a unique key K which identifies the author as well as the core and the authors private key as input. The output is a set of watermarks $W_B = (w_{B_1}, w_{B_2}, \cdots, w_{B_m})$. Each element w_{B_i} must fit into a single lookup table. For Xilinx Virtex-II and II Pro FPGAs, which use 4-input lookup tables, the size is 16 bit.

Additionally, the number of usable lookup tables which can carry a watermark must be determined. This can be done by extracting all lookup table contents and coordinates: $\mathcal{L}_B(I_B) = \{x_{B_1}, x_{B_2}, \ldots, x_{B_q}\}$. The next step is to find suitable location candidates which can carry a watermark. For Xilinx Virtex-II and II Pro FPGAs, possible candidates are unused lookup table in a used slice. Such candidates can be easily determined, because they carry the initialization value `0xFFFF`, whereas unused lookup tables in unused slices have `0x0000` as initialization value. The higher the number of location candidates and therefore watermarked lookup tables is, the more reliable is the proof of authorship. For example, if only one lookup table candidate was found, only $2^4 = 16$ different watermark values overall exist, which makes the proof of authorship contradictable.

The content of the chosen locations of the bitfile core I_B can be replaced by the watermarks W_B with the embedder $\widetilde{I_B} = \mathcal{E}_B(I_B, W_B)$ (see Figure 5.19). The result is the watermarked bitfile core $\widetilde{I_B}$. The distance $Dist_B(I_B, \widetilde{I_B})$ between the watermarked and original core is low, because of the functional correctness and all electrical properties of the core are preserved. Furthermore, if the watermarks are near to the functional lookup tables, the watermarks cannot be easily distinguished from the functional lookup tables.

For extracting the watermarks, we need the bitfile $\widetilde{I_B}$ from the accused company, and the locations of the watermarks (see Figure 5.19). The first step is to extract the content and coordinates from all lookup table in $\widetilde{I_B}$: $\mathcal{L}_B(\widetilde{I_B}) = \{\widetilde{x}_{B_1}, \widetilde{x}_{B_2}, \ldots, \widetilde{x}_{B_q}\}$. Using the locations from the core developer, the watermarks $\widetilde{W_B}$ can be identified. By comparing these watermarks to the watermarks W_B of the core developer, the detection process $\mathcal{D}_B(\widetilde{I_B}, W_B) = true/false$ can be finished.

Robustness Analysis Attacks against the watermarking scheme are *ambiguity, removal*, and *key copy attacks*. The prevention of the *copy attack*, where an attacker watermarks an IP core which he illegally obtained with his own signature, is almost impossible. A possible solution of this dilemma is to watermark all published works or register the core on a trusted third party institute.

In case of *removal attacks*, the attacker tries to remove the watermarks. If he knows the location of the watermarks this task is easy. Therefore, it is utmost important that the locations of the watermarked are kept secret. If the attacker does not know the locations, he can try to analyze the bitfile. If he is only able to extract the lookup table content and the locations of the lookup tables, it is almost impossible to detect the

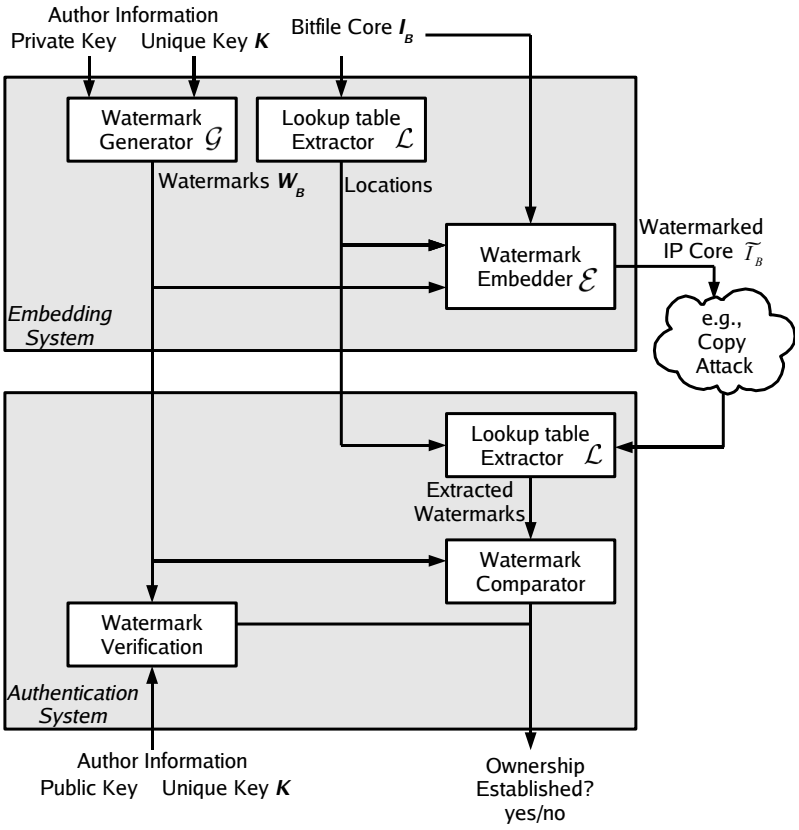


Figure 5.19: The watermarking bitfile IP core approach consists of an embedding system and an authentication system. The embedder needs the author information and the bitfile core and the result is the watermarked core $\widetilde{I}_B$ which can be published. The authorship of the core can be established by extracting and comparing the watermark and verifying the authentication of the watermark with the author information.

watermark, because the locations are near the functional lookup tables and the content is not distinguishable from the other lookup tables. However, if the attacker is able to reverse engineer the bitfile core to the logic level ($\widetilde{I}_L = \mathcal{T}_{L \leftarrow B}(\widetilde{I}_B)$), the watermarks are easy to detect and can be removed. This task is, however, very expensive if no reverse engineering tool is available. For Virtex-II devices the Xilinx "reverse engineering" tool *JBits* [Xilc] is available, which is in fact able to remove the watermarks.

The attacker may analyze the bitfile core and search for lookup table content which he can present as his own watermark in case of *ambiguity attacks*. He can use the inserted watermarks and assert that these watermarks belong to him. To be successful with such an attack, he must also present the procedure to generate the watermarks. Hereby, the attacker must generate a signatures or key which identifies him as the author and fits to the watermarks inside the core. This is very hard to achieve due to

the usage of one way cryptographic functions. Furthermore, the attacker can present some functional lookup tables as his watermarks. This should also be nearly impossible due to the characteristics of one way cryptographic functions. Another possibility to check this attack, is to remove the watermarks from the bitfile core. The correct watermarks are inserted after the implementation of the core and therefore the core should keep the functional correctness. Whereas the removal of the wrong watermarks which are functional lookup table contents, destroys the core.

Using asynchronous public/private key cryptographic functions for the watermark generation and verification and further storing information about the core into the unique key successfully prevents *key copy attacks*.

Additional Reading More information about this method can be found in [SZT08].

5.3 Constraint-Based Watermarking of IP Cores

All optimization problems have constraints which must be satisfied to achieve a valid solution. Solutions which satisfy this constraints are the *solution space*. Constraint-based watermarking techniques represent a signature as a set of *additional constraints* which are applied to the hardware optimization and synthesis problem. These additional constraints reduce the solution space (see Figure 5.20) since the chosen solution must also satisfy the additional constraints. The same solution could be achieved with neglecting these additional constraints with probability P_c. The probability P_c of this event is given by the following formula:

$$P_c = \frac{n_w}{n_o} \tag{5.17}$$

where n_o is the number of solutions which satisfy the original constraints and n_w is the number of solutions which satisfy both the original and the additional constraints [KLMS+01, KHPC98]. If P_c is very small, a solution that also satisfies the additional (watermarking) constraints is a *strong proof* of the existence of the watermark.

Qu proposes a methodology to make a part of the watermark – for constraint-based watermarking, some additional constraints – public which should deter attackers [Qu02]. The other parts, called *private watermark*, are only known by the core author and are used to verify the authorship in case that the public watermark was attacked. A similar approach is used by Qu and others to generate different fingerprints by dividing the additional constraints into two parts [QP00]: The first part is a set of relaxed constraints which denote the watermark. By applying distinct constraints to the second part, different independent solutions can be generated which may be used as diverse fingerprinted designs.

Charbon proposed a technique to embed watermarks on different abstraction levels which he called *hierarchical watermarking* [Cha98]. The idea is, if an attacker is

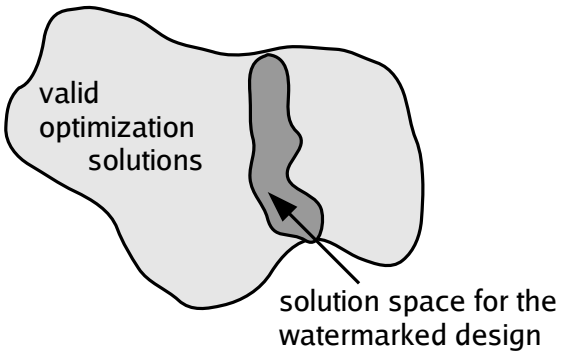


Figure 5.20: The solution space of an original and a watermarked design. If a design satisfies the original and the additional constraints, then the design is protected by a watermark. The probability that the additional constraints are satisfied by chance should be low to have a strong proof of authorship.

able to remove a watermark, for example, embedded into the layout of a circuit, the watermarks added at higher abstraction levels are still present. However, Charbon focused more on *layout, nets*, and *latch watermarking techniques* which are only applicable for ASIC layout cores.

The verification of a constraint-based watermark is usually done with the watermarked core as it is. This means the watermarked core can be purchased or published and from the distributed cores the watermark can be verified. However, if the core is combined with other cores and traverses further design steps, the watermark information is usually lost or it cannot be extracted.

Van Le and Desmedt [LD03] present an ambiguous attack for constraint-based watermarking techniques. The authors add further constraints to the watermarked solution by allowing only a minimal increase of the overhead. The result is a slightly degenerated solution which satisfies many additional constraints. This means that in this solution, a lot of different signatures can be found which destroys the unique identification of the core developer. They choose, for example, the constraint-based watermarking approach for *graph coloring*. Further, this attack might be applicable to other constraint-based watermarking techniques.

As it was the case with additive watermarking strategies, constraint-based watermarking strategies are applicable for HDL, netlist, and bitfile cores.

5.3.1 HDL Cores

HDL code is usually produced by human developers or high-level synthesis tools. Both can set additional constraints to watermark a design. One approach is to use a watermarked *scan chain* [KP98]. Scan chains are usually used in ASIC designs to

access the internal registers for debugging purposes. The use of scan chains in FPGA designs is rather unusual, but might be helpful in some cases. At first, a number will be assigned to each register, and the registers will be sorted. Now, a *pseudo random sequence* will be generated from the signature. Registers are selected with an algorithm which uses a random sequence as input. For K_c scan chains, the first K_c selected registers are chosen as the first register in each chain. Depending on the signature, we have a variation on the scan chains which can be used to detect the watermark. It is possible that an unfortunately chosen start of a chain could result in the allocation of more routing resources. Moreover, the maximum clock frequency for the scan chain can be limited. This approach is easy to verify, if the scan chains can be accessed from outside of the chip. Problems occur, if the scan chain is only used internally or is not connected to any device. In such a case, there is no verification possibility.

Some work was done for watermarking *digital signal processing* (DSP) functions [RAMSP99, CD00]. This kind of watermarking has more in common with media watermarking instead if IP watermarking. Both approaches alter the function of the core slightly by embedding a watermark. In [RAMSP99], the coefficients of *finite impulse response* (FIR) filters are slightly varied according to the watermark. Additionally, the authors use different structures to build the FIR filter which also corresponds to the signature. In [CD00], these ideas are extended and proven correct by mathematical analysis.

5.3.2 Netlist Cores

An approach to watermark netlist cores is to preserve certain nets during *synthesis* and *mapping* [KHPC98]. Synthesis tools merge signals or nets together and produce new nets. Only a few nets from the synthesis input will be visible in the synthesis result. The technology mapping tool also eliminates nets by assembling gates together in a lookup table. Kirovski's approach enumerates and sorts all nets in a design. The first K_c (see previous section) nets of the input are chosen by the synthesis tools according to a signature. These nets will be prevented from elimination by the design tools by connecting these nets to a temporary output of the core. The new outputs from additional constraints for the synthesis tool, and the corresponding result is related to the watermark. A disadvantage is that it is easy to remove the additional logic. If the content of the lookup table is synthesized again, the watermark will be removed.

Meguerdichan and others presented a similar approach for netlist cores where additional constraints are added during the *technology mapping step* of the synthesis process [MP00]. In this approach, critical signals are not altered which preserves the timing and the performance of the core. The signature is encoded into the number of allowed inputs of a certain primitive cell, e.g., a gate or a lookup table. The primitive cells which are not in the critical path are enumerated, and according to the signature, the number of usable inputs are constrained.

Khan and others watermark netlist cores by doing a *rewiring* after synthesis [KT05]. Rewiring means that redundant connections between primitive cells are added in the netlist which makes other original connections redundant. These new redundant connections are removed.

Bai and others introduce a method for watermarking transistor netlists for *full custom designs* [BGXC07]. The transistors are enumerated and sorted into a list like in the approach above. Corresponding to the pseudo random stream generated from the signature, the width of the transistor gate is altered. If the transistor is assigned a '1' from the random stream, the transistor width is increased by a constant value.

5.3.3 Bitfile and Layout Cores

Additional placement, routing, or timing constraints can be added to watermark bitfile cores. To embed a watermark with placement constraints, Kahng and others place the *configurable logic blocks* (CLBs) in even or odd rows depending on the signature [KMM+98]. In this approach, the signature is transformed into even/odd row placement constraints. The placed core will be tested on preserving the constraints and, if necessary, CLBs are swapped. This method has no logical resource overhead and the additional costs of routing the resources are very small or tend to zero, because the placement is altered only marginally. The problem of verification is to extract the CLB placement information. Only if knowing how the CLBs correspond to the signature, the watermark can be verified. A strategy to achieve this is to uniquely enumerate the CLBs in an FPGA from the top left corner.

Kahng and others [KMM+98] propose a second approach by adding constraints to the router. The constraints achieve that a net selected by the signature is routed with some additional, unusual routing resources. These unusual resources can be, for example, *wrong way* segments. A wrong way segment is a segment in which the net goes to the wrong direction and then back in the right direction to form a backstrap. The authors claim that this is unlikely for a normal router, and so such a net can be verified as a watermarked net.

Furthermore, additional timing constraints can be used to watermark a core. Timing constraints limit the route and logic delay between two registers. Kahng and others propose a technique to select paths which have timing constraints according to the signature. The timing constraints for these paths are split into two separate constraints. For example, let a path have six logic gates and a timing constraint of 10 *ns*. The new constraint is 4 *ns* for the first 3 cells and 6 *ns* for the rest [KLMS+01].

Another approach by Jain and others measures the delay on selected paths and adds new timing constraints on these paths [JYPQ03]. The new constraints are chosen based on the measured delay by setting the last digit to a value of a bit from the signature. For example, let a path have a delay of 5.73 *ns*. If the coded bit is a '1', the new constraint for this path is 5.71 *ns*, if the coded bit is '0', the constraint is 5.70 *ns*.

Narayan and others present a watermarking approach of the layout by modifying the number of *vias* and *bends* of certain nets [NNC+01]. Like in other approaches, the nets are enumerated, and additional vias or bends are inserted according to the signature.

Saha and others present a watermarking scheme by altering the size of the repeaters according to the signature [SSK07]. In high performance ASIC designs, *repeaters* (a buffer for amplification of the signal) are inserted into critical nets to decrease the delay.

5.4 Other Approaches

Many other approaches exist for protecting IP cores or designs from unlicensed usage or alteration. For example, the *VHDL Obfuscator & Watermarker* [VIS] is able to obfuscate VHDL cores in a way that the algorithm is hidden, but leaves the core synthesizeable. This approach make reverse engineering and alteration of the core much harder. Further, by using different scrambling techniques, a watermark can be embedded in the obfuscated code. Clearly, this watermark can only be detected at the RTL in the HDL core and is lost once the core is synthesized.

Other approaches prevent the copying of bitfile designs by using *unique FPGA* or *board identification*. If the obtained bitfile is programmed on another board, the function will not work. The unique identification can be done by using a *non-volatile external device*, a *unique key* embedded into the FPGA, or by *physical unclonable functions* (PUFs) [Dri09].

Kessner uses a non-volatile CPLD for board identification [Kes00]. A bit sequence is calculated by a cryptographic algorithm implemented on the FPGA and additionally on the CPLD. If the results of both implementations are the same, then the design "knows" that it is executed on the "right" board and starts its operation. Similarly, *challenge-response approaches* are published in an Altera white paper [Alt] and a Xilinx application note [Xila]. A challenge consisting of a sequence produced from a *random generator* is sent to a cryptographic algorithm implemented into a non-volatile device. The response of the device calculated with a secure key is compared with the result of the same algorithm and key implemented on the FPGA. The application is enabled if both results are the same. Couture and Kent propose a method where the IP core reads out a secure token, stored in a non-volatile memory in periodic time-lags [CK06]. Inside the token, the type and the life span of the license is encoded. For preventing the cloning of the bitfile, the token also includes a *unique FPGA identification number*.

In the Spartan-3A FPGA family, Xilinx implants a factory-set read-only unique number called *Device DNA* in each device [Xile]. This 57-bit number can be used to develop cores which allow execution only on specified FPGA devices.

A *physical unclonable function* (PUF) returns a unique value, which is extracted from physical properties of an object. *Silicon PUFs* (SPUFs) generate this device-dependent value from different manufacturing-related variations of timing and delay behaviors on nets of the silicon device [GCvDD02]. Related work of SPUFs can be categorized into approaches using *ring oscillators* and approaches using so-called *Arbiter-PUFs*. Gassend and others propose a method using ring oscillators in FPGAs for generating device-dependent values [GCvDD02]. The ring oscillators swing with a certain frequency, and the output is used to enable a counter, clocked with the operational clock. Lee and others [LLG+04] and Lim and others [LLG+05] present SPUFs, realized with an Arbiter-PUF at the IC fabric. An Arbiter-PUF, shown in Figure 5.21, consists of two identical designed delay lines, one for a data signal and one for a clock signal which can be crossed with multiplexers. The challenge vector enables a route through the multiplexers for both signals. Edges are generated and propagate through the network according to the challenge vector. If the clock signal reaches the flip-flop, the current value of the data signal is registered. The result is a '1' if the clock signal is faster than the data signal; otherwise the result is '0'. Varying the challenge vector will cause different results, which can only be reproduced on the same device. Using another device, the achieved results are completely different. Mjzoobi and others [MKP09] show an implementation of an Arbiter-PUF for Virtex-5 FPGAs whereas Suh and Devadas [SD07] implement an Arbiter-PUF for Virtex-4. Holcomb and others [HBF07] and Guajardo and others [GKST07] present approaches where the *initial state* of SRAMs is used as a PUF. During the power up of SRAMs, some memory cells switch to a '1', others to a '0', depending on the process variations. Guajardo and others reported that *Block RAMs* of some FPGAs can be used for generating a unique key which might be used for design protection.

Simpson and Schaumont propose an authentication system for software, running in a soft core on an FPGA, by using a PUF [SS06]. Later, Guajardo and others enhanced this approach [GKST07].

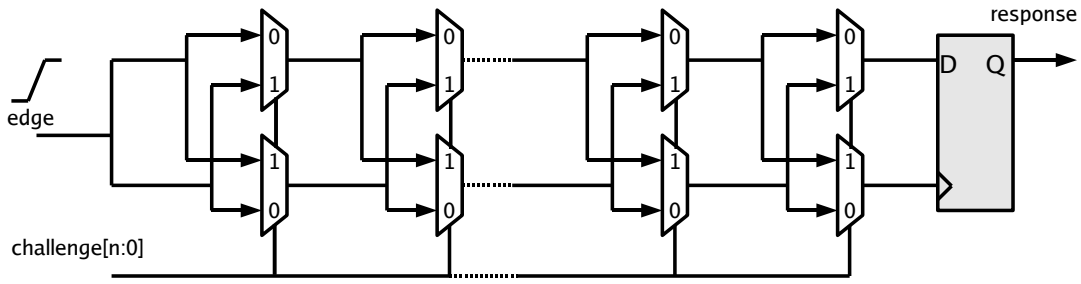


Figure 5.21: An Arbiter-PUF consists of a flip-flop and two delay lines the routing of which can be altered by different challenge values. An edge propagates through the multiplexer network to the flip-flop. The registered response is determined by which signal arrives first. The responses of different challenges are device dependent, hence to minimal uncontrollable path delay variations of different devices.

Bibliography

[AARR03] Dakshi Agrawal, Bruce Archambeault, Josyula R. Rao, and Pankaj Rohatgi. The EM Side-Channel(s). In *CHES '02: 4th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 29–45, London, UK, 2003. Springer-Verlag.

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity. In *CCS '05: Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, New York, NY, USA, 2005. ACM Press.

[ABEL09] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity: Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):1–40, 2009.

[ABMF04] Todd Austin, David Blaauw, Trevor Mudge, and Krisztián Flautner. Making Typical Silicon Matter with Razor. *Computer*, 37(3):57–65, 2004.

[AHTA04] Amr T. Abdel-Hamid, Sofiéne Tahar, and El Mostapha Aboulhamid. A Survey on IP Watermarking Techniques. *Design Automation for Embedded Systems*, 9(3):211–227, 2004.

[Ajl95] Cheryl Ajluni. Two new Imaging Techniques Promise to Improve IC Defect Identification. *Electronic Design*, 43(14):37–38, 1995.

[AK96] Ross Anderson and Markus Kuhn. Tamper Resistance: A Cautionary Note. In *WOEC'96: Proceedings of the 2nd conference on Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 1–11, Berkeley, CA, USA, 1996. USENIX Association.

[Ale96] Aleph One. Smashing the Stack for Fun and Profit. *Phrack magazine*, 49(7), 1996.

[All00] VSI Alliance. Intellectual Property Protection White Paper: Schemes, Alternatives and Discussion Version 1.1. *Issued by Intellectual Property Protection Development Working Group, Ver*, 1.1, 2000.

[All07] Business Software Alliance. Fifth Annual BSA and IDC Global Software Piracy Study. Technical report, 2007.

[ALR01] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Fundamental concepts of dependability. *Technical Report Series – University of Newcastle upon Tyne Computing Science*, 2001.

[Alt] Altera. FPGA Design Security Solution Using MAX II Devices. *URL:* `http://www.altera.com/literature/wp/wp_m2dsgn.pdf`.

[And72] James P. Anderson. Computer Security Technology Planning Study, 1972.

[ANKA99] Z. Alkhalifa, V. S. Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):627–641, 1999.

[ARRJ06] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Hardware-assisted Run-time Monitoring for Secure Program Execution on Embedded Processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(12):1295–1308, 2006.

[ARS04] André Adelsbach, Markus Rohe, and Ahmad-Reza Sadeghi. Overcoming the Obstacles of Zero-knowledge Watermark Detection. In *MM&Sec '04: Proceedings of the 2004 workshop on Multimedia and security*, pages 46–55, New York, NY, USA, 2004. ACM.

[Aus95] Kenneth Austin. Data Security Arrangements for Semiconductor Programmable Devices, February 7 1995. US Patent 5,388,157.

[Aus99] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, Washington, DC, USA, 1999. IEEE Computer Society.

[BAFS05] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, and Darko Stefanović. Randomized Instruction Set Emulation. *ACM Trans. Inf. Syst. Secur.*, 8(1):3–40, 2005.

[Bar] Scott Barrick. Designing Around an Encrypted Netlist: Is The Pain Worth the Gain? *D&R Industry Articles. URL:* `http://www.design-reuse.com/articles/18205/encrypted-netlist.html`.

[Bau08] Florian Baueregger. Identifikation von signierten Schaltungen anhand von Leistungsverbrauchsmessungen. Dilpomarbeit, Department of Computer Science 12, University of Erlangen-Nuremberg, January 2008.

[BBC05] BBC. 1986: Coal Mine Canaries made Redundant. *URL: http://news.bbc.co.uk/onthisday/hi/dates/stories/december/30/newsid_2547000/2547587.stm*, 2005.

[BDH+98] Feng Bao, Robert H. Deng, Yongfei Han, Albert B. Jeng, A. Desai Narasimhalu, and Teow-Hin Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults. In *Proceedings of the 5th International Workshop on Security Protocols*, pages 115–124, London, UK, 1998. Springer-Verlag.

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[BEA06] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural Support for Software-based Protection. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 42–51, New York, NY, USA, 2006. ACM.

[BGB06] Lilian Bossuet, Guy Gogniat, and Wayne Burleson. Dynamically Configurable Security for SRAM FPGA Bitstreams. *International Journal of Embedded Systems*, 2(1):73–85, 2006.

[BGXC07] Fujun Bai, Zhiqiang Gao, Yi Xu, and Xueyu Cai. A Watermarking Technique for Hard IP Protection in Full-custom IC Design. In *International Conference on Communications, Circuits and Systems (ICCCAS 2007)*, pages 1177–1180, 2007.

[BK00] Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack Magazine*, 2000.

[BKIL03] Saurabh Bagchi, Zbigniew Kalbarczyk, Ravishankar Iyer, and Y. Levendel. Design and Evaluation of Preemptive Control Signature (PECOS) Checking. *IEEE Transactions on Computers*, 2003.

[BLW+01] Saurabh Bagchi, Y Liu, Keith Whisnant, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Y. Levendel, and Larry Votta. A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller. In *DSN '01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*, pages 225–234, Washington, DC, USA, 2001. IEEE Computer Society.

[BM07] Jan A. Bergstra and C. A. Middelburg. Instruction Sequences with Indirect Jumps. *Electronic Report PRG0709, Programming Research Group, University of Amsterdam*, 2007.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software-Practice Experience*, 30(7):775–802, 2000.

[BRSS08] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions go Bad: Generalizing Return-oriented Programming to RISC. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38, New York, NY, USA, 2008. ACM.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *CRYPTO '97: Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, pages 513–525, London, UK, 1997. Springer-Verlag.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Runtime Defense against Stack Smashing Attacks. In *ATEC '00: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 251–262, Berkeley, CA, USA, 2000. USENIX Association.

[BTH96] Laurence Boney, Ahmed H. Tewfik, and Khaled N. Hamdy. Digital Watermarks for Audio Signals. In *International Conference on Multimedia Computing and Systems*, pages 473–480, 1996.

[BWWA06] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-Based Transparent and Comprehensive Control-Flow Error Detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[CBB+01] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic Protection from printf Format String Vulnerabilities. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2001. USENIX Association.

[CBD+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Linux Expo*, 1999.

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[CD00] Roy Chapman and Tariq S. Durrani. IP Protection of DSP Algorithms for System on Chip Implementation. *IEEE Transactions on Signal Processing*, 48(3):854–861, 2000.

[CE99] Cristina Cifuentes and Mike Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, pages 192–199, Washington, DC, USA, 1999. IEEE Computer Society.

[CH01] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: A Compile-time Solution to Buffer Overflow Attacks. In *International Conference on Distributed Computing Systems*, volume 21, pages 409–420. IEEE Computer Society; 1999, 2001.

[Cha98] Edoardo Charbon. Hierarchical Watermarking in IC Design. In *Proceedings of the IEEE Custom Integrated Circuits Conference*, pages 295–298, 1998.

[CHP97] Po-Yung Chang, Eric Hao, and Yale N. Patt. Target Prediction for Indirect Jumps. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 25(2):274–283, 1997.

[CK06] Nathaniel Couture and Kenneth B. Kent. Periodic Licensing of FPGA based Intellectual Property. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 234–234, New York, NY, USA, 2006. ACM.

[Con99] Matt Conover. w00w00 on Heap Overflows. *URL:* `http://www.w00w00.org/files/articles/heaputut.txt`, 1, 1999.

[CPG+06] Encarnacion Castillo, Luis Parrilla, Antonio Garcia, Antonio Loris, and Uwe Meyer-Baese. IPP Watermarking Technique for IP Core Protection on FPL Devices. In *International Conference on Field Programmable Logic and Applications, 2006. FPL'06*, pages 487–492, 2006.

[CPG+08] Encarnacion Castillo, Luis Parrilla, Antonio Garcia, Uwe Meyer-Baese, Guillermo Botella, and Antonio Lloris. Automated Signature Insertion in Combinational Logic Patterns for HDL IP Core Protection.

In *4th Southern Conference on Programmable Logic, 2008*, pages 183–186, 2008.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium*, Berkeley, CA, USA, 1998. USENIX Association.

[CPW74] J. R. Connet, E. J. Pasternak, and B. D. Wagner. Software Defenses in Real-time Control Systems. *Digest of papers*, page 94, 1974.

[CSB92] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[Dau06] Andrew Dauman. An Open IP Encryption Flow Permits Industry-wide Interoperability. *Synopsys, Inc. White Paper*, June 2006.

[Des97] Solar Designer. Non-executable Stack Patch. *URL:* `http://www.openwall.com/linux/`, 1997.

[DMW98] J. H. Daniel, D. F. Moore, and J. F. Walker. Focused Ion Beams for Microfabrication. *Engineering Science and Education Journal*, 7(2):53–56, 1998.

[Dob03] Igor Dobrovitski. Exploit for CVS Double free() for Linux pserver. *Neohapsis Archives (*`http://www.security-express.com/archives/fulldisclosure/2003-q1/0545.html`*)*, 2003.

[Dri09] Saar Drimer. Security for Volatile FPGAs. November 2009.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. *ACM SIGPLAN Notices*, 38(5):155–167, 2003.

[DS90] X. Delord and Gabriele Saucier. Control Flow Checking in Pipelined RISC Microprocessors: the Motorola MC88100 Case Study. In *Proceedings of the Euromicro'90 Workshop on Real Time*, pages 162–169, 1990.

[DS91] X. Delord and Gabriele Saucier. Formalizing Signature Analysis for Control Flow Checking of Pipelined RISC Multiprocessors. In *Proceedings of the IEEE International Test Conference on Test*, pages 936–945, Washington, DC, USA, 1991. IEEE Computer Society.

[DSG05] Nij Dorairaj, Eric Shiflet, and Mark Goosman. PlanAhead Software as a Platform for Partial Reconfiguration. *Xilinx XCELL Journal, Art*, 55:68–71, 2005.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[EL02] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. *IEEE Software*, 19(1):42–51, 2002.

[Erl07] Úlfar Erlingsson. Low-level Software Security: Attacks and Defenses. *Foundations of Security Analysis and Design IV*, 4677:92, 2007.

[ES84] James B. Eifert and John Paul Shen. Processor Monitoring Using Asynchronous Signatured Instruction Streams. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995,'Highlights from Twenty-Five Years', Reprinted from FTGS-14 1984*, pages 394–399, 1984.

[Est] Chip Estimate. ChipEstimate.com. *URL:* `http://www.chipestimate.com/`.

[Fed01] Federal Information Processing Standards Publication 197. Announcing the Advanced Encryption Standard (AES). *URL:* `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`, 2001.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[FKF+03] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol – Version 3.0. *URL:* `http://www.mozilla.org/projects/security/pki/nss/ssl/draft302.txt`, 1996.

[Fra] Fraunhofer IISB. FIB Focused Ion Beam - Anwendungsbeispiele, Spezifikationen und Prinzip. *URL:* `http://www.iisb.fraunhofer.de/de/arb_geb/technologie_an_fib.htm`.

[FS01] Mike Frantzen and Mike Shuey. StackGhost: Hardware Facilitated Stack Protection. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 55–66, Berkeley, CA, USA, 2001. USENIX Association.

[FS03] Niels Ferguson and Bruce Schneier. *Practical Cryptography*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[FT03] Y. C. Fan and H. W. Tsao. Watermarking for Intellectual Property Protection. *Electronics Letters*, 39(18):1316–1318, 2003.

[Gai94] Jiri Gaisler. Concurrent Error-detection and Modular Fault-tolerance in a 32-bit Processing Core for Embedded Space Flight Applications. In *Twenty-Fourth International Symposium on Fault-Tolerant Computing FTCS, 1994*, pages 128–130. IEEE Computer Society Press, 1994.

[GCvDD02] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160, New York, NY, USA, 2002. ACM.

[GDWL92] Daniel D. Gajski, Nikil D. Dutt, Allen C.-H. Wu, and Steve Y.-L. Lin. *High-level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[Ger91] Anne Geraci. *IEEE Standard Computer Dictionary: Compilation of IEEE Standard Computer Glossaries*. IEEE Press, Piscataway, NJ, USA, 1991.

[GHJM05] Dan Grossman, Michael Hicks, Trevor Jim, and Greg Morrisett. Cyclone: A Type-safe Dialect of C. *C/C++ Users Journal*, 23(1):112–139, 2005.

[GKST07] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. FPGA Intrinsic PUFs and Their Use for IP Protection. In *CHES '07: Proceedings of the 9th international workshop on Cryptographic Hardware and Embedded Systems*, pages 63–80, Berlin, Heidelberg, 2007. Springer-Verlag.

[GO98] Anup K. Ghosh and Tom O'Connor. Analyzing Programs for Vulnerability to Buffer Overrun Attacks. In *Proceedings of the 21st National*

Information Systems Security Conference, Crystal City, VA, pages 274–382, 1998.

[GRRV03] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Soft-Error Detection Using Control Flow Assertions. In *DFT '03: Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 581–588, Washington, DC, USA, 2003. IEEE Computer Society.

[GRRV05] Olga Goloubeva, Maurizio Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Improved Software-based Processor Control-flow Errors Detection Technique. In *Reliability and maintainability symposium*, pages 583–589, 2005.

[Hag] Hagai Bar-El, Discretix Technologies Ltd. Known Attacks Against Smartcards. *URL:* `http://www.discretix.com/PDF/KnownAttacksAgainstSmartcards.pdf`.

[HB03] Eric Haugh and Matt Bishop. Testing C Programs for Buffer Overflow Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, volume 2. Citeseer, 2003.

[HBF07] Daniel E. Holcomb, Wayne P. Burleson, and Kevin Fu. Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags. In *Proceedings of the Conference on RFID Security*. Citeseer, 2007.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion Detection using Sequences of System Calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[HJ92] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, volume 136, 1992.

[HP99] Inki Hong and Miodrag Potkonjak. Behavioral Synthesis Techniques for Intellectual Property Protection. In *Design Automation Conference*, pages 849–854, 1999.

[IBM] IBM. Rational Purify. *URL:* `http://www-01.ibm.com/software/awdtools/purify/`.

[IFI90] IFIP WG. Dependability: Basic Concepts and Terminology: in English, French, German, Italian and Japanese: IFIP WG 10.4. *Dependable Computing and Fault Tolerance*, 1990.

[ISO05] ISO JTC. 1/SC 27: Information Technology–Security Techniques–Code of Practice for Information Security Management. 2005.

[ITR07] ITRS. International Technology Roadmap for Semiconductors, 2007 Edition, URL: `http://www.itrs.net/Links/2007ITRS/Home2007.htm`. Technical report, 2007.

[ITS91] ITSEC. Information Technology Security Evaluation Criteria (ITSEC). *Office for Official Publications of the European Communities*, 1991.

[Jal94] Pankaj Jalote. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible Bounds Checking for Arrays and Pointers in C Programs. *Automated and Algorithmic Debugging*, pages 13–26, 1997.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association.

[JMKP07] José A. Joao, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Dynamic Predication of Indirect Jumps. *IEEE Computer Architecture Letter*, 6(2):25–28, 2007.

[JYPQ03] Adarsh K. Jain, Lin Yuan, Pushkin R. Pari, and Gang Qu. Zero Overhead Watermarking Technique for FPGA Designs. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 147–152. ACM Press, 2003.

[KBT08] Dirk Koch, Christian Beckhoff, and Jürgen Teich. ReCoBus-Builder - a Novel Tool and Technique to Build Statically and Dynamically Reconfigurable Systems for FPGAs. In *Proceedings of International Conference on Field-Programmable Logic and Applications (FPL 08)*, pages 119–124, Heidelberg, Germany, September 2008.

[KC08] Kris Kaspersky and Alice Chang. Remote Code Execution through Intel CPU Bugs. In *Hack In The Box (HITB) 2008 Malaysia Conference*, 2008.

[KE91] David R. Kaeli and Philip G. Emma. Branch History Table Prediction of Moving Target Branches due to Subroutine Returns. *SIGARCH Computer Architecture News*, 19(3):34–42, 1991.

[Kea01] Tom Kean. Secure Configuration of a Field Programmable Gate Array. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 259–260, Washington, DC, USA, 2001. IEEE Computer Society.

[Kes00] David Kessner. Copy Protection for SRAM based FPGA Designs. *Application Note, Free IP Project, URL:* `http://web.archive.org/web/20031010002149/http://free-ip.com/copyprotection.html`, 2000.

[KHPC98] Darko Kirovski, Yean-Yow Hwang, Miodrag Potkonjak, and Jason Cong. Intellectual Property Protection by Watermarking Combinational Logic Synthesis Solutions. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 194–198, New York, NY, USA, 1998. ACM.

[KHT08] Dirk Koch, Christian Haubelt, and Jürgen Teich. Efficient Reconfigurable On-Chip Buses for FPGAs. In *16th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2008)*, pages 287–290. IEEE Computer Society, April 2008.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397, London, UK, 1999. Springer-Verlag.

[KK99] Oliver Kömmerling and Markus G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology (Smartcard '99)*, pages 9–20, 1999.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-injection Attacks with Instruction-set Randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[KLMR04] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new Dimension in Embedded System Design. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 753–760, New York, NY, USA, 2004. ACM. Moderator-Ravi, Srivaths.

[KLMS+98] Andrew Byun Kahng, John Lach, William Henry Mangione-Smith, Stefanus Mantik, Igor Leonidovich Markov, Miodrag M. Potkonjak, Paul Askeland Tucker, Huijuan Wang, and Gregory Wolfe. Watermarking Techniques for Intellectual Property Protection. In *DAC '98:*

Proceedings of the 35th annual Design Automation Conference, pages 776–781, New York, NY, USA, 1998. ACM.

[KLMS+01] Andrew Byun Kahng, John Lach, William Henry Mangione-Smith, Stefanus Mantik, Igor Leonidovich Markov, Miodrag M. Potkonjak, Paul Askeland Tucker, Huijuan Wang, and Gregory Wolfe. Constraint-Based Watermarking Techniques for Design IP Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(10):1236–1252, 2001.

[klo99] klog. The Frame Pointer Overwrite. *Phrack magazine*, 55(9), 1999.

[KMM+98] Andrew Byun Kahng, Stefanus Mantik, Igor Leonidovich Markov, Miodrag M. Potkonjak, Paul Askeland Tucker, Huijuan Wang, and Gregory Wolfe. Robust IP Watermarking Methodologies for Physical Design. In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 782–787, New York, NY, USA, 1998. ACM.

[KMM08] Tom Kean, David McLaren, and Carol Marsh. Verifying the Authenticity of Chip Designs with the DesignTag System. In *HOST '08: Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 59–64, Washington, DC, USA, 2008. IEEE Computer Society.

[KNKA96] Ghani A. Kanawati, V. S. Sukumaran Nair, Narayanan Krishnamurthy, and Jacob A. Abraham. Evaluation of Integrated System-level Checks for on-line Error Detection. In *IPDS '96: Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, pages 292–301, Washington, DC, USA, 1996. IEEE Computer Society.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 104–113, London, UK, 1996. Springer-Verlag.

[Kot06] Arun Kottolli. The Economics of Structured- and Standard-cell-ASIC Designs. *EDN Magazine*, 51(6):61–68, 2006.

[KP98] Darko Kirovski and Miodrag Potkonjak. Intellectual Property Protection Using Watermarking Partial Scan Chains For Sequential Logic Test Generation. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, 1998.

[KR06] Ian Kuon and Jonathan Rose. Measuring the Gap between FPGAs and ASICs. In *FPGA '06: Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, pages 21–30, New York, NY, USA, 2006. ACM.

[Kre03] Andreas Krennmair. ContraPolice: A libc Extension for Protecting Applications from Heap-smashing Attacks, 2003.

[KT05] M. Moiz Khan and Spyros Tragoudas. Rewiring for Watermarking Digital Circuit Netlists. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(7):1132–1137, 2005.

[LB94] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software-Practice and Experience*, 24(2):197–218, 1994.

[LBD+04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting Software. *IEEE Software*, 21(3):92–100, 2004.

[LBMC94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[LC02] Kyung-suk Lhee and Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88, Berkeley, CA, USA, 2002. USENIX Association.

[LC06] Qiming Li and Ee-Chien Chang. Zero-knowledge Watermark Detection Resistant to Ambiguity Attacks. In *MMSec '06: Proceedings of the 8th workshop on Multimedia and security*, pages 158–163, New York, NY, USA, 2006. ACM.

[LD03] Tri Van Le and Yvo Desmedt. Cryptanalysis of UCLA Watermarking Schemes for Intellectual Property Protection. In *IH '02: Revised Papers from the 5th International Workshop on Information Hiding*, pages 213–225, London, UK, 2003. Springer-Verlag.

[LKMS04] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. *Security in Pervasive Computing*, pages 237–252, 2004.

[LLG+04] Jae W. Lee, Daihyun Lim, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srini Devadas. A Technique to Build a Secret Key in

Integrated Circuits for Identification and Authentication Applications. In *Proceedings of the IEEE VLSI Circuits Symposium*, pages 176–179. Citeseer, 2004.

[LLG+05] Daihyun Lim, Jae W. Lee, Blaise Gassend, G. Edward Suh, Marten Van Dijk, and Srini Devadas. Extracting Secret Keys from Integrated Circuits. *IEEE Transactions on Very Large Scale Integration Systems*, 13(10):1200, 2005.

[LMSP98] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Signature Hiding Techniques for FPGA Intellectual Property Protection. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 186–189, New York, NY, USA, 1998. ACM.

[LMSP99] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Robust FPGA Intellectual Property Protection through Multiple Small Watermarks. In *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 831–836, New York, NY, USA, 1999. ACM.

[LMSP01] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak. Fingerprinting Techniques for Field-Programmable Gate Array Intellectual Property Protection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 20, 2001.

[Lu82] David J. Lu. Watchdog Processors and Structural Integrity Checking. *IEEE Transaction on Computers*, 31(7):681–685, 1982.

[MBS07] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, Washington, DC, USA, 2007. IEEE Computer Society.

[MBS08] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. *IEEE Micro-Institute of Electrical and Electronics Engineers*, 28(1):52–59, 2008.

[MdR99] Todd C. Miller and Theo de Raadt. strlcpy and strlcat: Consistent, Safe, String Copy and Concatenation. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 1999. USENIX Association.

[MH91] Edgar Michel and Wolfgang Hohl. Concurrent Error Detection using Watchdog Processors in the Multiprocessor System MEMSY. In *Fault-tolerant computing systems: tests, diagnosis, fault treatment: 5th International GI/ITG/GMA Conference, Nürnberg, September 25-27, 1991: proceedings*, page 54. Springer, 1991.

[MHPS96] István Majzik, Wolfgang Hohl, András Pataricza, and Volker Sieh. Multiprocessor Checking using Watchdog Processors. *Computer Systems Science and Engineering*, 11(5):301–310, 1996.

[MKGT92] Ghassem Miremadi, Johan Karlsson, Ulf Gunneflo, and Jan Torin. Two Software Techniques for On-line Error Detection. In *Digest of Papers, Twenty-Second International Symposium on Fault-Tolerant Computing. FTCS-22.*, pages 328–335, 1992.

[MKP09] Mehrdad Majzoobi, Farinaz Koushanfar, and Miodrag Potkonjak. Techniques for Design and Implementation of Secure Reconfigurable PUFs. *ACM Transaction on Reconfigurable Technology Systems*, 2(1):1–33, 2009.

[MKSL03] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A Processor Architecture Defense against Buffer Overflow Attacks. In *Proceedings of the IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, pages 243–250. Citeseer, 2003.

[MLS91] Thierry Michel, Régis Leveugle, and Gabriele Saucier. A New Approach to Control Flow Checking Without Program Modification. In *Digest of Papers of Twenty-First International Symposium of Fault-Tolerant Computing*, pages 334–343, 1991.

[MM88] Aamer Mahmood and Edward J. McCluskey. Concurrent Error Detection Using Watchdog Processors-A Survey. *IEEE Transaction on Computers*, 37(2):160–174, 1988.

[MN98] Lee D. McFearin and V. S. Sukumaran Nair. Control Flow Checking Using Assertions. *Dependable Computing and Fault Tolerant Systems*, 10:183–200, 1998.

[MP00] Seapahn Meguerdichian and Miodrag Potkonjak. Watermarking while Preserving the Critical Path. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 108–111, New York, NY, USA, 2000. ACM.

[MS91] Henrique Madeira and João G. Silva. On-line Signature Learning and Checking: Experimental Evaluation. In *CompEuro'91: Proceedings of the 5th Annual European Computer Conference of Advanced Computer Technology, Reliable Systems and Applications*, pages 642–646, 1991.

[MV05] Matt Messier and John Viega. Safe C String Library v1.0.3. *URL: http://www.zork.org/safestr/*, 2005.

[MWB+10] Matthias May, Norbert Wehn, Abdelmajid Bouajila, Johannes Zeppenfeld, Walter Stechele, Andreas Herkersdorf, Daniel Ziener, and Jürgen Teich. A Rapid Prototyping System for Error-Resilient Multi-Processor Systems-on-Chip. In *Proceedings of DATE'10*, pages 375–380, March 2010.

[Nam82] Masood Namjoo. Techniques for Concurrent Testing of VLSI Processor Operation. In *Proceedings of International Test Conference*, pages 461–468, 1982.

[Nam83] Massod Namjoo. CERBERUS-16: An Architecture for a General Purpose Watchdog Processor. In *Proceedings of Symposium on Fault-Tolerant Computing*, pages 216–219, 1983.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[NNC+01] Naveen Narayan, Rexford D. Newbould, Jo Dale Carothers, Jefrey J. Rodriguez, and W. Timothy Holman. IP Protection for VLSI Designs via Watermarking of Routes. In *Proceedings of the 14th Annual IEEE International ASIC/SOC Conference*, pages 406–410, 2001.

[OCK+75] Severo M. Ornstein, William R. Crowther, M. F. Kraley, R. D. Bressler, A. Michel, and Frank E. Heart. Pluribus: A Reliable Multiprocessor. In *AFIPS '75: Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 551–559, New York, NY, USA, 1975. ACM.

[Oli01] Arlindo L. Oliveira. Techniques for the Creation of Digital Watermarks in Sequential Circuit Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(9):1101–1117, 2001.

[OR95] Joakim Ohlsson and Marcus Rimen. Implicit Signature Checking. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium*

on Fault-Tolerant Computing, pages 218–227, Washington, DC, USA, 1995. IEEE Computer Society.

[OSM02] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111–122, 2002.

[OVB+06] Hilmi Özdoganoglu, T. N. Vijaykumar, Carla E. Brodley, Benjamin A. Kuperman, and Ankit Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. *IEEE Transaction on Computers*, 55(10):1271–1285, 2006.

[PAX03] PAX Team. Non Executable Data Pages. *URL:* `http://pax.grsecurity.net/docs/pax.txt`, 2003.

[PB04] Jonathan Pincus and Brandon Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 02(4):20–27, 2004.

[PBA10] Andrea Pellegrini, Valeria Bertacco, and Todd Austin. Fault-Based Attack of RSA Authentication. In *Proceedings of Design and Test in Europe (DATE'10)*, pages 855–860, March 2010.

[PMHH93] András Pataricza, István Majzik, Wolfgang Hohl, and Joachim Hönig. Watchdog Processors in Parallel Systems. *Microprocessing and Microprogramming*, 39(2-5):69–74, 1993.

[QP00] Gang Qu and Miodrag Potkonjak. Fingerprinting Intellectual Property using Constraint-addition. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 587–592, New York, NY, USA, 2000. ACM.

[Qu02] Gang Qu. Publicly Detectable Watermarking for Intellectual Property Authentication in VLSI Design. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 21(11):1363–1367, 2002.

[Rag06] Roshan G. Ragel. *Architectural Support for Security and Reliability in Embedded Processors*. PhD thesis, University of New South Wales, Sydney, Australia, 2006.

[RAMSP99] Azra Rashid, Jeet Asher, William H. Mangione-Smith, and Miodrag Potkonj. Hierarchical Watermarking for Protection of DSP Filter Cores. In *Proceedings of the Custom Integrated Circuits Conference. Piscataway, NJ*, pages 39–45. IEEE Press, 1999.

[RBD+01] Rob A. Rutenbar, Max Baron, Thomas Daniel, Rajeev Jayaraman, Zvi Or-Bach, Jonathan Rose, and Carl Sechen. (When) will FPGAs kill ASICs? (panel session). In *DAC '01: Proceedings of the 38th annual Design Automation Conference*, pages 321–322, New York, NY, USA, 2001. ACM.

[RCV+05] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

[Reu] Design & Reuse. Catalyst of Collaborative IP Based SoC Design. *URL:* `http://www.design-reuse.com/`.

[Ric02] Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *URL:* `http://www1.corest.com/files/files/11/StackGuardPaper.pdf`, 2002.

[Ric08] Robert Richardson. CSI Computer Crime and Security Survey. Technical report, 2008.

[RKMV03] William Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time Detection of Heap-based Overflows. In *LISA '03: Proceedings of the 17th USENIX conference on Large Installation Systems Administraton*, pages 51–60, Berkeley, CA, USA, 2003. USENIX Association.

[RL04] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.

[RLC+07] Eduardo L. Rhod, Calisboa A. Lisbôa, L. Carro, Massimo Violante, and Matteo Sonza Reorda. A Non-intrusive On-line Control Flow Error Detection Technique for SoCs. In *IEEE Latin-Americam Test Workshop, LATW*, volume 8, 2007.

[RRKH04] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in Embedded Systems: Design Challenges. *ACM Transaction on Embedded Computer Systems*, 3(3):461–491, 2004.

[RSA78] Ronald Linn Rivest, Adi Shamir, and Leonard Max Adleman. A Method for Obtaining Digital Signatures and Public-key Cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[SAH] Bassel Soudan, Wael Adi, and Abdulrahman Hanoun. Enabling Secure Integration of Multiple IP Cores in the Same FPGA. *D&R Industry Articles. URL:* `http://www.design-reuse.com/articles/21638/secure-integration-ip-cores-fpga.html`.

[SBDB01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, Washington, DC, USA, 2001. IEEE Computer Society.

[SBE+07a] Walter Stechele, Oliver Bringmann, Rolf Ernst, Andreas Herkersdorf, Katharina Hojenski, Peter Janacik, Franz Rammig, Jürgen Teich, Norbert Wehn, Johannes Zeppenfeld, and Daniel Ziener. Autonomic MPSoCs for Reliable Systems. In *Proceedings of Zuverlässigkeit und Entwurf (ZuD 2007)*, pages 137–138, Munich, Germany, March 2007.

[SBE+07b] Walter Stechele, Oliver Bringmann, Rolf Ernst, Andreas Herkersdorf, Katharina Hojenski, Peter Janacik, Franz Rammig, Jürgen Teich, Norbert Wehn, Johannes Zeppenfeld, and Daniel Ziener. Concepts for Autonomic Integrated Systems. In *Proceedings of edaWorkshop07*, Munich, Germany, June 2007.

[SD07] G. Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 9–14, New York, NY, USA, 2007. ACM.

[SFF+02] Oliverio J. Santana, Ayose Falcón, Enrique Fernández, Pedro Medina, Alex Ramírez, and Mateo Valero. A Comprehensive Analysis of Indirect Branch Prediction. In *ISHPC '02: Proceedings of the 4th International Symposium on High Performance Computing*, pages 133–145, London, UK, 2002. Springer-Verlag.

[Sha07] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, New York, NY, USA, 2007. ACM.

[SKB02] Li Shang, Alireza S. Kaviani, and Kusuma Bathala. Dynamic Power Consumption in Virtex-II FPGA Family. In *FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays*, pages 157–164, New York, NY, USA, 2002. ACM Press.

[SM90] Nirmal Raj Saxena and Edward J. McCluskey. Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Transactions on Computers*, 39(4):554–559, 1990.

[SPA] SPARC International, Inc. The SPARC Architecture Manual V8. *URL:* `http://www.sparc.com/standards/V8.pdf`.

[SPE] SPEC (Standard Performance Evaluation Corporation). SPEC CPU2000 V1.3. *URL:* `http://www.spec.org`.

[SS87] Michael A. Schuette and John Paul Shen. Processor Control Flow Monitoring using Signatured Instruction Streams. *IEEE Transaction on Computers*, 36(3):264–277, 1987.

[SS06] Eric Simpson and Patrick Schaumont. Offline Hardware/Software Authentication for Reconfigurable Platforms. *Cryptographic Hardware and Embedded Systems (CHES 2006)*, pages 311–323, 2006.

[SSK07] Debasri Saha and Susmita Sur-Kolay. Fast Robust Intellectual Property Protection for VLSI Physical Design. In *ICIT '07: Proceedings of the 10th International Conference on Information Technology*, pages 1–6, Washington, DC, USA, 2007. IEEE Computer Society.

[ST82] Thirumalai Sridhar and Satish M. Thatte. Concurrent Checking of Program Flow in VLSI Processors. In *Proceedings International Test Conference (ITC 1982), Philadelphia, PA, USA*, pages 191–199, 1982.

[ST87] John Paul Shen and Stephen P. Tomas. A Roving Monitoring Processor for Detection of Control Flow Errors in Multiple Processor Systems. *Microprocessing and Microprogramming*, 20(4-5):249–269, 1987.

[SXZ+04] Zili Shao, Chun Xue, Qingfeng Zhuge, Edwin Hsing Mean Sha, and Bin Xiao. Security Protection and Checking in Embedded System Integration Against Buffer Overflow Attacks. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, pages 409–412, Washington, DC, USA, 2004. IEEE Computer Society.

[Syn] Synopsys. Synplify Premier. *URL:* `http://www.synopsys.com/Tools/Implementation/FPGAImplementation/CapsuleModule/syn_prem_ds.pdf`.

[SZHS03] Zili Shao, Qingfeng Zhuge, Yi He, and Edwin Hsing Mean Sha. Defending Embedded Systems Against Buffer Overflow via Hardware/-Software. In *ACSAC '03: Proceedings of the 19th Annual Computer*

Security Applications Conference, pages 352–361, Washington, DC, USA, 2003. IEEE Computer Society.

[SZT08] Moritz Schmid, Daniel Ziener, and Jürgen Teich. Netlist-Level IP Protection by Watermarking for LUT-Based FPGAs. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2008)*, pages 209–216, Taipei, Taiwan, December 2008.

[TC00] Ilhami Torunoglu and Edoardo Charbon. Watermarking-based Copyright Protection of Sequential Functions. *IEEE Journal of Solid-State Circuits*, 35(3):434–440, 2000.

[TH07] Jürgen Teich and Christian Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung*. Springer, 2007.

[UR94] Shambhu J. Upadhyaya and Bina Ramamurthy. Concurrent Process Monitoring with No Reference Signatures. *IEEE Transaction on Computers*, 43(4):475–480, 1994.

[US-08] US-CERT. Vulnerability Notes Database CERT Coordination Center. *URL:* `http://www.kb.cert.org/vuls/`, 2008.

[VBKM00] John Viega, J. T. Bloch, Yoshi Kohno, and Gary E. McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257, Washington, DC, USA, 2000. IEEE Computer Society.

[vdV04] Arjan van de Ven. New Security Enhancements in Red Hat Enterprise Linux v.3, update 3. *Red Hat, August*, 2004.

[Ven00] Vendicator. Stack Shield: A Stack Smashing Technique Protection Tool for Linux. *URL:* `http://www.angelfire.com/sk/stackshield/info.html`, 2000.

[VIS] VISENGI. VHDL Obfuscator & Watermarker. *URL:* `http://www.visengi.com/en/products/software/vhdl_obfuscator`.

[WD01] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, Washington, DC, USA, 2001. IEEE Computer Society.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, 2000.

[Whe] David A. Wheeler. Flawfinder Home Page. *URL:* `http://www.dwheeler.com/flawfinder`.

[Wil07] Ron Wilson. Panel Unscrambles Intellectual Property Encryption Issues. *EDN Magazine URL:* `http://www.edn.com/article/CA6412249.html`, 2007.

[Woj98] Rafal Wojtczuk. Defeating Solar Designer Nonexecutable Stack Patch. *Bugtraq mailinglist*, 1998.

[Wri08] Craig Wright. Hacking Coffee Makers. *URL:* `http://www.securityfocus.com/archive/1/493387`, 2008.

[WS90] Kent Wilken and John Paul Shen. Continuous Signature Monitoring: Low-cost Concurrent Detection of Processor Control Errors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(6):629–641, 1990.

[Xila] Xilinx Inc. FPGA IFF Copy Protection Using Dallas Semiconductor/Maxim DS2432 Secure EEPROMs. *URL:* `http://www.xilinx.com/support/documentation/application_notes/xapp780.pdf`.

[Xilb] Xilinx Inc. ISE Design Suite Software Manuals and Help - PDF Collection These. *URL:* `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/manuals.pdf`.

[Xilc] Xilinx Inc. JBits 3.0 SDK for Virtex-II. *URL:* `www.xilinx.com/labs/projects/jbits/`.

[Xild] Xilinx Inc. MicroBlaze Processor Reference Guide. *URL:* `http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf`.

[Xile] Xilinx Inc. Protect Your Brand with Extended Spartan-3A FPGAs. *URL:* `http://www.xilinx.com/products/design_resources/security/devicedna.htm`.

[Xilf] Xilinx Inc. Virtex-II Platform FPGAs: Complete Data Sheet. *URL:* `http://www.xilinx.com/support/documentation/data_sheets/ds031.pdf`.

[Xil03] Xilinx Inc. Next-Generation Virtex Family From Xilinx to top one Billion Transistor Mark. *URL:* `http://www.xilinx.com/prs_rls/silicon_vir/03131_nextgen.htm`, 2003.

[XKI03] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 260–272, 2003.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture Support for Defending against Buffer Overflow Attacks. In *Workshop on Evaluating and Architecting Systems for Dependability*, 2002.

[YC80] Stephen S. Yau and Fu-Chung Chen. An Approach to Concurrent Control Flow Checking. *IEEE Transactions on Software Engineering*, 6(2):126–137, 1980.

[ZAT06] Daniel Ziener, Stefan Aßmus, and Jürgen Teich. Identifying FPGA IP-Cores based on Lookup Table Content Analysis. In *Proceedings of 16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, pages 481–486, Madrid, Spain, August 2006.

[Zie10] Daniel Ziener. *Techniques for Increasing Security and Reliability of IP Cores Embedded in FPGA and ASIC Designs*. PhD thesis, University of Erlangen-Nuremberg, 7 2010. Verlag Dr. Hut, Munich, Germany.

[Zim95] Philip R. Zimmermann. *The official PGP user's guide*. MIT Press, Cambridge, MA, USA, 1995.

[ZT05] Daniel Ziener and Jürgen Teich. Evaluation of Watermarking Methods for FPGA-Based IP-cores. Technical Report 01-2005, University of Erlangen-Nuremberg, Department of CS 12, Hardware-Software-Co-Design, Am Weichselgarten 3, D-91058 Erlangen, Germany, March 2005.

[ZT06] Daniel Ziener and Jürgen Teich. FPGA Core Watermarking Based on Power Signature Analysis. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT 2006)*, pages 205–212, Bangkok, Thailand, December 2006.

[ZT08a] Daniel Ziener and Jürgen Teich. Concepts for Autonomous Control Flow Checking for Embedded CPUs. In *Proceedings of the 5th International Conference on Autonomic and Trusted Computing (ATC 2008)*, pages 234–248, Oslo, Norway, June 2008.

[ZT08b] Daniel Ziener and Jürgen Teich. Power Signature Watermarking of IP Cores for FPGAs. *Journal of Signal Processing Systems*, 51(1):123–136, April 2008.

[ZT09] Daniel Ziener and Jürgen Teich. Concepts for Run-time and Error-resilient Control Flow Checking of Embedded RISC CPUs. *Int. Journal of Autonomous and Adaptive Communications Systems*, 2(3):256–275, July 2009.

[ZZPL04] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Hardware Supported Anomaly Detection: Down to the Control Flow Level. Technical report, Georgia Institute of Technology, 2004.

[ZZPL05] Tao Zhang, Xiaotong Zhuang, Santosh Pande, and Wenke Lee. Anomalous Path Detection with Hardware Support. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 43–54, New York, NY, USA, 2005. ACM.