cyberagentur

ECOSYSTEM FOR TRUSTWORTHY IT

LOS 2: FORMAL SECURITY VERIFICATION OF HARDWARE

Wolfgang Kunz,
Dominik Stoffel,
Johannes Müller,
Mohammad R. Fadiheh
Fachbereich Elektrotechnik und Informationstechnik, Rheinland-Pfälzische Technische Universität

Version 1.0

Herausgeberin:
Agentur für Innovation in der Cybersicherheit GmbH

Disclaimer

Die hier geäußerten Ansichten und Meinungen sind ausschließlich diejenigen der Autorinnen und Autoren und entsprechen nicht notwendigerweise denjenigen der Agentur für Innovation in der Cybersicherheit GmbH oder der Bundesregierung.

Diese Studie wurde durch die Agentur für Innovation in der Cybersicherheit GmbH beauftragt und finanziert. Eine Einflussnahme der Agentur für Innovation in der Cybersicherheit GmbH auf die Ergebnisse fand nicht statt.

Impressum

Herausgeberin: Agentur für Innovation in der Cybersicherheit GmbH
Große Steinstraße 19, 06108 Halle (Saale), Germany
E-Mail: kontakt@cyberagentur.de
Internet: www.cyberagentur.de
Twitter: https://twitter.com/CybAgBund

Die Nutzungsrechte liegen bei der Herausgeberin.

Lizenz: CC BY-NC-ND 4.0: https://creativecommons.org/licenses/by-nc-nd/4.0/
Erscheinungsdatum: 12.07.2023
Redaktion: Abteilung Sichere Systeme, Referat Sichere Hardware und Lieferketten

LOT 2 – Formal Security Verification of *Hardware*

Wolfgang Kunz, Dominik Stoffel, Johannes Müller, Mohammad R. Fadiheh

Fachbereich Elektrotechnik und Informationstechnik, Rheinland-Pfälzische Technische Universität

Version 1.0

This study provides an overview on methods of formal hardware verification in view of relevant security objectives for the *basic IT elements* in hardware at the microarchitectural level. We derive the targets of sign-off security verification from an analysis of common hardware weaknesses and the relevant security requirements for microarchitectures. The study relates these targets to the state of the art in formal hardware verification and describes strengths and weaknesses of different methods and methodologies. This leads to research recommendations for formal security verification of hardware.

Table of Contents

1. Introduction – Microarchitectural Security of Hardware

Ever since the invention of the first microprocessor in 1971, our society's reliance on electronic computing systems has been increasing at an accelerating pace. Computing systems ranging from small embedded systems to high-end server computers are part of the critical infrastructure for almost all industrial sectors, governmental and public institutions as well as for private life. System-on-Chips (SoCs) and embedded systems are ubiquitous in the modern society; with their abundance of connectivity features they create a new attack surface for cyber-attacks.

Our trust in computing systems, whether it regards the proper functioning of a power grid or the confidentiality of industrial data in edge computing, depends mainly on the provided safety and security features of the underlying computing system. Although the majority of the advanced security features, such as end-to-end encryption, are implemented at the software level, they rely on basic hardware primitives to deliver the intended functionalities. For example, encryption can be rendered useless if the hardware system does not provide a secure memory isolation ensuring the confidentiality of the encryption keys. In common terminology, such hardware primitives form the "*root-of-trust"* of the computing system. They constitute a set of trusted functionalities to ensure the security of the system. Design verification of these hardware parts is especially critical. Any security flaw in the hardware root-of-trust can affect virtually all applications deployed on the system.

Hardware systems are difficult or, in some cases, even impossible to patch, which exacerbates the challenge of dealing with hardware security flaws. Countless reports in recent years on system vulnerabilities at the hardware level, e.g., [1], [2], attest to the fact that hardware security flaws can pose a genuine threat to the overall system security. The Common Weakness Enumeration database (CWE) [3] has acknowledged this problem by including hardware vulnerabilities as a separate category of security weaknesses.

The role of hardware in system security is not limited to providing security-related features to support software functions. Weaknesses in the hardware design itself can introduce severe vulnerabilities to the computing system. At the microarchitectural level, these weaknesses mostly have two sources: the hardware circuit executing a security-critical software application may leak confidential information through side channels, in particular timing of the software execution [4], or (possibly very subtle) design bugs escape conventional verification procedures and cause security risks for the entire system.

In this Lot, we focus on the formal verification of hardware security requirements at the *microarchitectural level*. Microarchitectural descriptions at the Register Transfer Level (RTL) are the point of reference for sign-off verification before the tape-out of a chip for manufacturing. Therefore, RTL descriptions of the microarchitecture typically serve as the *golden model* of an SoC and are the basis for all design refinements at lower levels as well as for manufacturing. Security issues at these lower levels, especially those related to the supply chains, are subject to Lot 4.

2. Security Goals – Microarchitectural Security Risks and Verification Objectives

Formal security verification at the microarchitectural level requires well-defined verification targets covering all security risks or *threat models* that are relevant for a chip's intended deployment domain. A threat model reflects the global security requirements for a system, such as confidentiality and integrity and the "attacker profile" describing the capabilities of a potential attacker to interact with the system and to attack a certain category of vulnerabilities. The appropriate formalization of the relevant threat models, i.e., the specification of properties to be verified, is one of the main challenges in security research.

In the following, we provide an overview on security vulnerabilities in hardware, common security targets and the role of threat models. While security vulnerabilities are inherent to the design under verification, security targets and the profile of a potential attacker must be defined by the verification engineer based on knowledge about the environment in which the chip will operate.

2.1. Security Vulnerabilities

Formal security verification must address a wide spectrum of potential security HW vulnerabilities. In the following, we consider these vulnerabilities and related verification challenges. At the microarchitectural hardware level, there are two main categories of hardware vulnerabilities that can be distinguished: *security-violating design bugs* and *microarchitectural timing side channels*. In the latter category, so called *transient execution side channels* have recently received great attention.

2.1.1. Security-Violating Design Bugs

Security-violating design bugs are the subset of all design bugs that, besides violating the functional specification, violate a relevant security target. These bugs corrupt, for example, the functionality of memory protection mechanisms or some advanced security feature like information flow tracking. Throughout this study, the discussion of functional design bugs holds (unless noted otherwise) also for trojans present in the RTL (cf. Sec. 3.2.2.2). Trojans inserted *after* RTL sign-off, however, require additional measures. This is subject of the study for Lot 4.

In principle, security-critical design bugs (and RTL trojans) can be detected by conventional functional verification. However, this requires the complete and correct specification of the entire design as well as of all its security mechanisms. Substantial effort is demanded from verification engineers to cover all security-relevant functional behaviors by a set of properties. In many practical settings, this effort is considered prohibitively large, especially, since it is not sufficient to restrict this exercise to only the processor. Also all peripherals must be covered. Even when all modules of an SoC have been treated, verification gaps may still remain: Since functional properties are usually formulated locally for individual SoC modules, security issues related to the communication between modules or to the interaction between hardware and firmware are easily missed [5]. The challenges of the *functional verification paradigm* to security verification are further discussed in Sec. 3.2.

2.1.2. Microarchitectural Side Channels

The problem of side channels in hardware has been subject to research already for decades. It has been understood that the same degrees of freedom that a designer may use for optimizing a design at the microarchitectural level may lead to side effects that can be exploited in security attacks. At the microarchitectural level, side channels are based on timing. Although a program may not have access rights to a certain set of data, depending on this data, one and the same program may behave slightly differently in terms of its own computation results, i.e., what data it stores in which registers and at which time points. These differences only affect the detailed timing of the microarchitectural implementation and have no impact at the level of the *instruction set architecture* (ISA), i.e., they do not affect the correct functioning of the program as seen by the programmer. However, if these subtle alterations of the program's execution at the microarchitectural level are caused by secret data to which the program must not have access, this may open a "side channel". An attacker, owning (and creating) such a program, may trigger and observe these alterations to infer secret information. This is called a "microarchitectural side channel attack".

2.1.2.1. "Classical" ISA-visible Timing Side Channels

Generally, in microarchitectural side channel attacks, the possible leakage of secret information is based on a microarchitectural resource that creates a timing information channel between different software processes that share this resource. For example, the cache can be such a shared resource and

an attacker can observe timing variations when accessing the cache based on the victim's cache access pattern. Various cache-based attacking schemes have been reported which deduce critical information from the footprint of an encryption software on the cache [4], [6], [7], [8], [9]. Also, other shared resources can be (mis-)used as the channel in a side-channel attack, as has been shown for DRAMs [10] and other shared functional units [11].

The following observation is key to classifying microarchitectural side channels and the corresponding verification targets (properties): In the attack scenario described above, the attacker process by itself is not capable of controlling both ends of a side channel. In order to steal secret information, it must interact with another process initiated by the system, the “victim process”, which manipulates the secret and “makes a noise”. In addition, the attacker must possess detailed knowledge about the victim software in order to make a meaningful correlation between the observed side channel and the victim's secret assets. Because of these prerequisites, the scope of a side channel attack may be limited to specific software components. While the detection and removal of such side channels in hardware may be beneficial in certain cases, possible defense mechanisms can benefit from their visibility at the ISA level and may rely exclusively on remedies at the software level (cf. Lot 1 and 3). Such remedies are typically applied to security-critical software components like encryption algorithms. Common measures include constant-time encryption [12] and cache access pattern obfuscation [13]. They prohibit the information flow at the “sending end” of the channel, i.e., the one owned by the victim process.

Although securing encryption software against these attacks is challenging because it demands a deep understanding of microarchitectural details, in the past, the threat of microarchitectural side channels was generally perceived to be limited to a small set of software applications. This general intuition, however, was drastically changed by the discovery of *transient execution side channel (TES) attacks*.

2.1.2.2. Transient Execution Side Channels (TESs)

Despite using similar channels for exfiltrating information, TES attacks are fundamentally different from classical microarchitectural side channels. TES attacks exploit side effects of transient instruction execution, a phenomenon not visible in the sequential execution semantics of the ISA. Similarly like a factory attempts to maximize its productivity by keeping all its machines running as much as possible, modern processors attempt to maximize the use of their hardware units to achieve highest possible computing performance. Therefore, processors can reschedule the instructions of a program and may “transiently” execute instructions ahead of time, without ensuring whether or not the flow of the program actually reaches those instructions. If such “speculation” turns out to be wrong, i.e., the transiently executed instructions are not part of the correct program flow, the processor discards their results.

In a TES attack an attacker exploits advanced microarchitectural features, such as speculative execution or out-of-order execution, to transiently execute a sequence of instructions. This transient instruction execution may leak secret data through timing side channels and is the root cause for TES attacks. Without affecting the ISA-level results of the program, the attacker triggers transient executions of instructions that depend on secret data. In this way, the attacker does not rely on a vulnerability within a victim software to make a noise. In fact, the TES attacker controls both ends of the channel, the part that triggers the side effect and sends out the information as well as the part that observes it. This makes TES attacks more threatening than the earlier known timing side channels of Sec. 2.1.2.1. In the TES scenario, a single user-level attacker program can establish a microarchitectural side channel leaking parts of the memory which are not accessed by any other program. Such HW covert channels not only can destroy the usefulness of encryption and secure authentication schemes but can steal data from essentially anywhere in the system. As a result, unlike classical side channel attacks, TES attacks threaten the overall security of the system and its root of trust.

The first TES attacks to be discovered were Spectre [14] and Meltdown [15] which made world-wide headlines in 2018. The variety of attacks using TESs discovered since then (e.g., MDS attacks [16], [17], [18]), speculative store bypass [19], speculative interference [20]), with many of them targeting a previously patched system (e.g., Fallout attack [16]), has proven that the threat by TESs is not limited to Spectre and Meltdown and generally calls for new attention towards hardware security.

2.1.3. Side Channels at Physical Levels

Also side channels related to the physical implementation of a chip can cause severe concerns. This is true in particular for power side channels [21], [22]. Note that techniques of formal hardware verification, at least in their present form, operate at the logic design levels, typically at the RTL or above. Therefore, only little research has been reported how formal methods can contribute a mitigation of physical-level weaknesses.

Attacks at the physical level extract functional or non-functional information from analog signals. Checking whether or not secret information can be leaked therefore calls for methods operating at the physical or analog levels of the system. Formal analog verification, however, is in its infancy. Digitizing analog signals to make them suitable for conventional (digital) formal verification creates problem instances of enormous complexity. Instead, formal methods can have promise when applied to verify defense mechanisms that are implemented at the logical level, such as masking [23] or balancing [24]. This may result in specialized methods that can be integrated into a verification flow for security, as discussed in Sec. 3.5. Abstracting from physical behaviors, however, bears the risk of missing security gaps that are not modeled by the formal tool, for example, as demonstrated by [25] for the work of [26].

In summary, while new and promising ideas may be emerging, the question whether or not formal hardware verification can be a general new research area for physical-level side channel detection is of speculative nature. It is therefore not further elaborated in this study.

2.2. Security Targets – Confidentiality and Integrity

Confidentiality and *integrity* are commonly considered the most relevant security targets for hardware. In the software domain it is common to also consider *availability* as a third security target. In the hardware domain it is common to subsume this notion under *integrity*, as becomes apparent from the following discussion.

Confidentiality of hardware is given if all information stored or processed in the system is protected against being retrieved by an unauthorized entity.

Integrity means preventing an attacker from changing or influencing a part of the system that is specified as protected. Similarly as in the software domain, where integrity forbids the unauthorized modification, deletion or insertion of data, we can understand integrity in hardware as the *integrity of information.* Specifically, this means that a set of protected registers and memory locations cannot be overwritten by an unauthorized entity. Since hardware, as opposed to software, is a physical system, the notion of integrity is commonly defined to have a wider scope. Attackers with physical access to the system may influence the system physically, for example, by manipulating voltage levels or by other physical fault injection techniques. Therefore, besides the *integrity of information* there is another relevant class of integrity goals in hardware which we can subsume under the notion of *integrity of operation*. Integrity of operation is maintained if an untrusted entity cannot change the result or timing of a protected *operation* in the system. Note that this also includes the *availability* of security-critical functional resources.

Besides physical attacks, another relevant threat to the integrity of hardware results from the integration of third-party components into the system. Intellectual Property (IP) modules from

untrusted sources bear the risk to maliciously influence security-critical operations of protected SoC components.

Further security targets may be of importance and are often considered as subordinate goals of confidentiality and integrity. We provide two examples:

Authenticity

Authenticity ensures the correct identification of communication partners as well as the authenticity of data. This is achieved by a verifiably correct authentication process determining whether the credentials given by a user or another system component are authorized to access the resource in question.

Privacy

Privacy refers to the objective to keep some or all processing of a system user secret to the rest of the system. This may involve the anonymity of the user. Verifiable measures for privacy include for example cryptographic techniques and trusted execution environments [27], [28]. The correctness of HW/SW interaction usually plays a major role when ensuring privacy.

2.3. Threat Models

A threat model for hardware captures the security requirements for a system in combination with a profile of the attackers. The attacker profile makes assumptions about how attackers can access the system and what methods they can use to exploit potential vulnerabilities of the system. For example, an attacker may access the system by running an unprivileged user process. Another threat model may consider access to the security-critical system through a third-party IP which is added to the system and which the attacker controls (cf. Sec. 2.2). Specific threat models are the basis for the specification of verifiable properties. The challenge consists in formulating these properties in such a way that a large spectrum of different threat models is covered by a manageable set of properties.

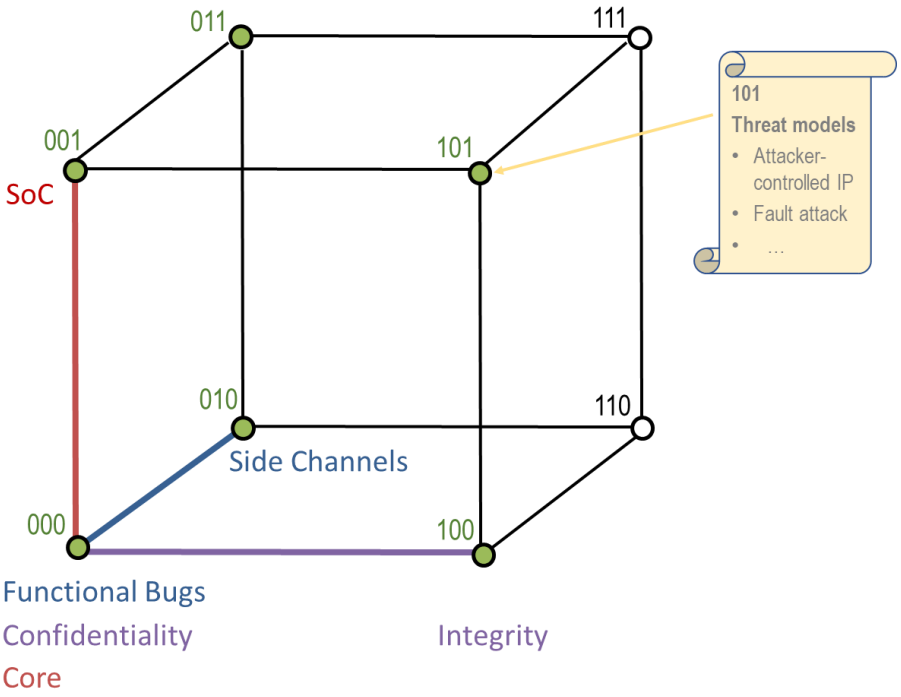

Figure 1: *Space of threat models*

Figure 1 illustrates the space of threat models that must be analyzed for specifying verifiable properties. The points associated with important threat models are marked in green color. Firstly, we must distinguish different kinds of security vulnerabilities. As elaborated in Sec. 2.1, it is meaningful to distinguish between security-violating functional bugs and non-functional vulnerabilities, in particular side channels. This is the blue dimension in the shown space. This distinction is reasonable since, by their definition, side channels are not detectable by any functional verification, such as conventional formal hardware property checking (cf. Sec.3.5). At the microarchitectural level, these side channels could be further divided in sub-classes such as TESs and other channels, as described in Sec. 2.1. For simplicity, we do not display these additional dimensions and restrict the illustration of
Figure 1 to a three-dimensional cube.

The vertical dimension (red color) of the cube in
Figure 1 distinguishes between vulnerabilities that occur only in *cores* and those requiring a global analysis of the entire *System-on-Chip (SoC)*. For example, TESs, such as Spectre and Meltdown, only require an analysis of the core while the root cause of other types of timing side channels can be distributed over several locations of the SoC. The distinction along this vertical dimension has a large impact on the kind of formal analysis that must be performed. While specialized methods exist specifically for cores, the analysis of the entire system demands more complex procedures typically based on compositional principles.

Thirdly, the horizontal axis of the cube separates threat models related to the security target of *confidentiality* from those related to *integrity*. While there exist well-defined notions of confidentiality in hardware, the formalization of hardware integrity can be more demanding and requires a careful analysis of the attacker profile relevant for the hardware's intended deployment domain. For example, in some scenarios a primary concern results from the integration of untrusted third-party IPs into an existing and trustworthy platform. In other scenarios, such as chipcard applications, fault attacks (e.g., by laser light) corrupting the system can be of particular concern.

We give some examples of threat models related to specific points of the cube in
Figure 1:

- ***Point 010***

  Threat Model 010

Security target:	Confidentiality of data in protected memory locations
Attacker Profile:	Attacker can run any program on the core with user-level privileges
Class of vulnerabilities:	Transient Execution Side Channel in cores

- ***Point 011***

  Threat Model 011

Security target:	Confidentiality of data in protected memory locations
Attacker Profile:	Attacker can run any program on the core with user-level privileges
Class of vulnerabilities:	ISA-visible timing side channels in core or peripherals

- ***Point 001***

  Threat Model 001

Security target:	Confidentiality of data in protected memory locations
Attacker Profile:	Attacker can run any program on the core and can access peripherals with user-level privileges
Class of vulnerabilities:	Functional design bugs (or trojans)

Note that several threat models can belong to each point in the space of
Figure 1, for example:

❖ ***Point 101***

Threat Model 101a

Security target:	Integrity of information and integrity of operation in security-critical parts of SoC
Attacker Profile:	Attacker controls a third-party IP which communicates with the security-critical SoC domain
Class of vulnerabilities:	Design bugs (insufficient protection mechanisms)

Threat Model 101b

Security target:	Integrity of information and integrity of operation in security-critical parts of SoC
Attacker Profile:	Attacker can inject faults by laser light anywhere in the SoC
Class of vulnerabilities:	Design bugs (insufficient protection mechanisms)

Threat Model 101c

Security target:	Integrity of information and integrity of operation in security-critical parts of SoC
Attacker Profile:	Attacker can inject faults by laser light anywhere in the SoC and can run any program on the main core with user-level privileges
Class of vulnerabilities:	Design bugs (insufficient protection mechanisms)

It is the task of the hardware security engineer to conduct a threat analysis for the considered design. The result of this analysis is a set of threat models that are relevant for the considered design and its deployment domain, such as illustrated by the above examples. The derived threat models are the basis for the verification engineer to define the verification targets. A set of properties to be verified is used for each threat model. A key challenge for the verification methodology is that the engineer must develop a full understanding about what threat models can be covered by specifying which properties. This means the engineer must understand for which security targets the developed properties provide guarantees, under what assumptions for the potential vulnerabilities and for which profile of the attacker.

Modern property languages, such as SVA, provide a strong basis for formalizing the threat models described above. Besides the language, however, the general verification methodology has a strong influence on how property sets with appropriate coverage are specified. The state of the art in these methodologies is subject to the following section.

3. State of the Art – Formal Hardware Verification Methods and Methodologies

3.1. New Era in Hardware Security

For better understanding the evolution of formal hardware verification in the field of microarchitectural security, it is helpful to pay particular attention to recent developments in the field.

In the context of cybersecurity, formal methods have particularly strong roots in the software domain. This has mainly two reasons: First, a large body of security violations in IT systems is based on software deficits; and formal methods to mitigate these weaknesses have been under intensive research for decades. Second, as elaborated in Sec. 2.1.2.1, many vulnerabilities that exploit hardware weaknesses, such as cache-based side channels, could mostly be addressed by software defenses.

In other words, a software-driven view on microarchitectural hardware security dominated the field for many years. However, this changed abruptly in January 2018 when headline news about the discovery of new hardware attacks called Spectre and Meltdown startled the general public around the globe. The industry reacted promptly and swift software updates were provided. However, they offered only little relief. It was quickly understood that Spectre and Meltdown belonged to a new class of ISA-invisible side channels, further explained in Sec. 2.1.2.2. A keynote statement of David Patterson (IEEE/ACM Design Automation Conference, 2018), co-inventor of modern RISC computer architectures and Turing Award winner 2017, underlines this insight: "*State of computer security is embarrassing for all of us in the computing field. It seems unlikely systems will ever become secure using software-only solutions"*.

It turns out that Spectre and Meltdown are only two examples of a larger class of new side channels which were named "transient execution side channels (TES)" (cf. Sec. 2.1.2.2). Almost on a monthly basis, new types of attacks of this class were discovered and reported. The variety of such attacks discovered in recent years ranges from MDS attacks (e.g., [3]) to Speculative Interference [4], with many of them successfully attacking a previously patched system (e.g., Fallout attack [3]). The initial hope for short-term and full solutions to defend against these new weaknesses were not fulfilled, as admitted by Martin Dixon, VP for Security at Intel: *"The potential for a transient execution to extract data being carried across a branch or a load is still a new field of research. Even though transient execution attacks are highly complex and difficult to carry out successfully outside of a lab, we expect it to remain a persistent focus area for researchers and the computer industry."*

Today, re-establishing trust into the microarchitectures of computing systems has become one of the main goals in the computer industry and among chip makers. Security verification and the development of defense mechanisms at the *hardware* level have become rapidly growing research fields and complement activities at the software level. There is general conviction that the formidable patch-and-pray cycles can only be overcome if comprehensive security guarantees are already provided during the design phase and when signing-off a chip before tape-out. Formal verification bears promise to provide such guarantees. However, while it is encouraging that formal methods have become mainstream in many industrial flows for hardware design, most of these techniques are tailored towards general functional design aspects and suffer from severe limitations when targeting microarchitectural security and side channels.

In the following, we first consider mainstream formal verification techniques, as they have been developed for checking the functional correctness of a design (Sections 3.2, 3.3, 3.4). They provide the basis for detecting security-critical design errors. Then, we describe the state of the art in techniques specifically targeting security (Sections 3.5, 3.6). The role of the formal hardware verification at the hardware/software interface is described in Sec. 3.7.

3.2. Formal Verification for Functional Correctness

This section summarizes basic concepts and the state of the art in common approaches to functional verification with formal methods. We compare the different classes of techniques qualitatively in terms of three important criteria for any formal method: *scalability*, *degree of automation* and *coverage*.

3.2.1. Formal Verification without Property Specification - Automatic Linting

"Automatic formal verification" of hardware, i.e., formal verification without the need of specifying verification targets by some property language, historically often served as the "appetizer" to formal methods and guided industry into a more elaborate use of formal techniques. Commonly, this class of methods, also referred to as "hardware linting", is used to check RTL design rules that must be fulfilled for *any* design, independently of its specific function. Hardware linting targets poor coding styles,

mismatches of the simulation and synthesis semantics in the RTL code, coding errors in finite state machine (FSM) implementations and similar issues. A particular focus often lies on detecting problems related to the synchronization of clocks and clock domain crossings.

Automatic linting is a standard feature in many commercial tools and is typically employed as a precursor to more advanced verification efforts, as described below. There is currently only little research on formal hardware linting in academia. Notable exceptions exist which, however, address advanced methods of formal verification that largely avoid the formalization of the design's functionality. Instead of simple design rules, sophisticated but still generic design properties are investigated. Examples of such advanced linting methods are the work of [29], [30], [31] for checking functional correctness in processor cores and of [32] for checking clock-domain crossings in very large designs.

Scalability: Checking design rules often does not require a deep logical analysis of the design or even a traversal of the system's state space. This holds, for example, if only structural conditions regarding design connectivity or coding rules are checked. Therefore, scalability of hardware linting techniques is usually very high. Entire chips can be analyzed in a single run. For more advanced approaches checking functional behavior in more depth, similar limitations apply as discussed below for property checking. In fact, the advanced techniques of hardware linting are often based on the property checking methods discussed below.

Degree of Automation: A high degree of automation is a clear differentiator of linting techniques compared to other verification techniques. In most cases, the properties are checked fully automatically.

Coverage: The beneficial characteristics of hardware linting with respect to scalability and automation come at the price of relatively poor coverage. Design-specific properties cannot be checked. Hence, this class of techniques never guarantees compliance of an implementation with a design-specific specification.

The trade-off between these criteria is illustrated in Figure 2 by the mark for "Linting".

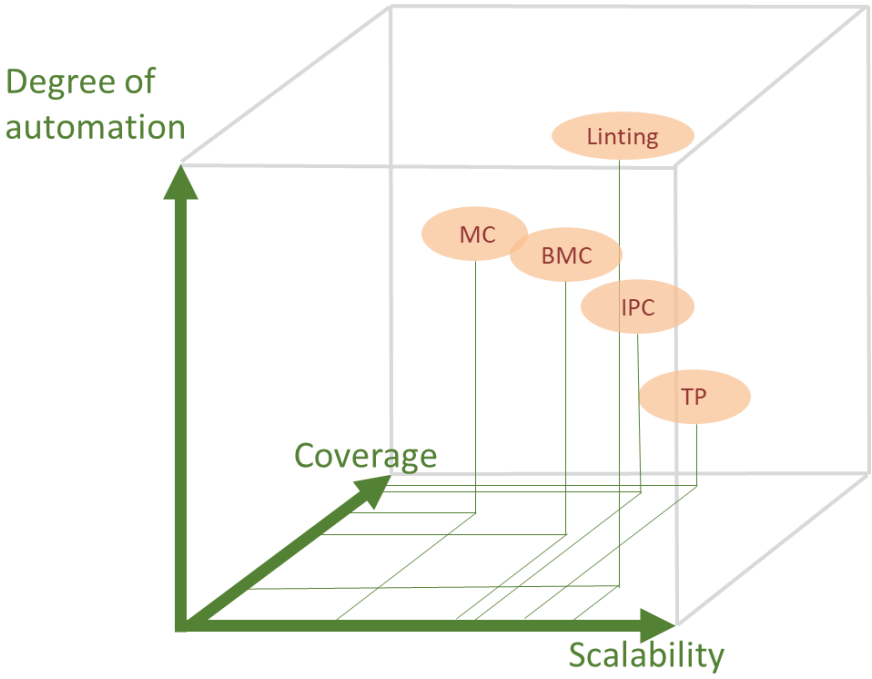

Figure 2: Qualities of methods of formal hardware verification

3.2.2. Formal Property Checking

In *property checking* a design model is checked against a functional specification using formal and automatic methods. The design model can be given at different levels of abstraction. In most industrial settings of hardware design, it is mainstream to use RTL models, typically given as VHDL or Verilog descriptions, as the basis for verification. Additionally, also descriptions at higher abstraction levels based on languages like SystemC gain popularity in the verification flow. The choice of a model and a level of abstraction have a large influence on what verification targets can be addressed. Models at a high level of abstraction allow for handling large designs but miss relevant information, for example, the clock-accurate timing of a microarchitectural implementation. Many security gaps, such as TESs, can only be detected by an analysis that operates on a model which is both bit- and clock cycle accurate. Therefore, the following discussion concentrates on formal verification at the RTL which is also the standard for SoC sign-off. The challenge of using abstraction levels above RTL in security verification is further discussed in Sec. 3.4.

The functional specification is often provided in an informal way using textual descriptions, flow charts, timing diagrams and the like. The main manual effort in formal verification by property checking results from the task to formalize this specification in terms of properties. Importantly, the properties must be written in such a way that the possibly wrong thinking of a designer and related implementation bugs are not imported into the property specification as well. This is accomplished by adopting a property specification style that describes the functional behavior abstractly and without restrictions regarding its implementation. Similar like the ISA specification of a processor denotes the behavior of a processor instruction in terms of programmer-visible registers and without any consideration of *how* this behavior is implemented (e.g. in-order pipeline or out-of-order pipeline), the property specification for hardware should only describe *what* functional behavior is expected but not *how* it is implemented. SystemVerilog Assertions (SVA) is currently the most popular language for specifying properties at the RTL.

The way how properties are formulated to specify the relevant verification targets is critical for the quality of the overall verification results. The question arises what properties shall be written and to what extent the functional behavior of a design is *covered* by a given property set. Different coverage metrics are available to answer these questions. Often, coverage in formal verification adopts concepts from simulation-based verification. Especially mutation analysis is commonly used. Faults are injected into the design to systematically generate design "mutants" and it is checked what fraction of these errors is discovered by the given property set [33], [34]. Other coverage metrics for formal methods address the completeness of the specification [35], [36], [37]. In general, *coverage* is a main concern in every verification flow and differentiates verification approaches from each other.

A formal property checker exhaustively evaluates a given property on the given model. This is the task of the underlying *proof method*. The proof method should be as automatic as possible and in many cases is indeed fully automated. The main criterion for the quality of the proof method is typically its scalability on large designs.

Scalability, degree of automation and coverage are the main differentiators between different approaches to property checking. The possible trade-offs between these three criteria have led to the development of two main classes of formal proof methods pursuing the *unbounded* or the *bounded paradigm* of property checking. The trade-offs associated with these paradigms are elaborated next.

3.2.2.1. The Unbounded Paradigm

The Unbounded Paradigm denotes the classical approach to formal verification by property checking. Invented already in the 1980s, *model checking* [38], [39], [40], [41] lays the foundation for many of today's formal verification methods both for hardware and for software. Both hardware and software

can be represented in a uniform way by well-defined sequential models. In model checking a sequential model's compliance with a temporal logic expression, the "property", is evaluated by formal and fully automatic methods. Model checking reasons on sequential models with a finite number of internal states and considers behaviors over infinite times. This allows for proving strong guarantees on a design. Modern algorithms for model checking often rely on data structures based on Binary Decision Diagrams (BDDs) [42]. BDDs are graph representation of Boolean functions and provide a powerful instrument to represent the large state space of a sequential model in a compact and manageable way.

Scalability:
Scalability is the main limitation of the unbounded paradigm. Model checking for today's hardware designs requires the traversal of huge state spaces. Substantial progress in model checking based on advanced state space representations and automatic abstraction techniques make it possible to deal with state spaces of $2^{hundreds}$ states. Although this is an astronomic number, it means that systems with up to a few hundreds of state variables can be handled. Note, however, that hardware modules of today's SoCs typically have *hundreds of thousands* of state variables. This limits the use of the unbounded paradigm to proving only local assertions in a design. In the context of security verification, the unbounded paradigm may be applied only to small SoC security elements. Therefore, in practice, the unbounded paradigm is often complemented with a methodology based on the *bounded paradigm,* described below.

Degree of Automation:
Conventional model checking techniques are fully automated. The only manual effort arises from the need of formulating the property. This effort depends largely on the considered verification tasks.

Coverage:
It is an advantage of unbounded model checking that the formulated properties can cover the specified behavior at all times during the existence of the system. This makes a strong contribution to coverage. However, coverage also depends on what properties have been formulated and how a set of several properties contributes to composing a proof with global guarantees [43]. This is a question of advanced methodologies whose adoption in practice has been hampered by the limited scalability of unbounded model checking. Therefore, in commercial practice, unbounded model checking is mostly used in the context of Assertion Based Verification (ABV) where designers sprinkle assertions into their code to build confidence into the design. Although commercial vendors support this process by certain (often proprietary) coverage-driven flows, the resulting coverage is often considered insufficient or remains unclear. In such a setting the bulk of the verification effort is therefore left to simulation.

The trade-off between these criteria is illustrated in Figure 2 by the mark for model checking "MC".

3.2.2.2. The Bounded Paradigm

3.2.2.2.1. Bounded Model Checking

Bounded Model Checking (BMC) [44] laid the foundation to the bounded paradigm in property checking. As the name suggests, in BMC the property checking problem is restricted to a finite bounded time interval. To this end, the temporal property is formulated for a finite time window starting from a known state of the system, often the reset state. This formulation has the great advantage that the reasoning on sequential systems can be mapped to the Boolean Satisfiability problem (SAT).

SAT is the problem to decide for a given Boolean function whether or not there exists a valuation to its variables such that the function assume the value '1', i.e., it is "satisfied". BMC leverages the fact that tools to solve the SAT problem, so called "SAT-solvers", have been a very active research field during the last two decades and very powerful solvers are available today, both in industry and in public domain.

In BMC the design and the property are converted into a Boolean function such that any valuation to its variables satisfying the function is a counterexample to the property. If the function is proven to be unsatisfiable, i.e., no counterexamples exist, the property is verified to hold for the given time interval and start state.

Scalability:

The scalability of BMC compares favorably with most other approaches in formal verification. Hardware designs with tens or even hundreds of thousands of inputs, outputs and internal state variables can often be handled within short proof times. Besides the size of the design also the nature of the property has an influence on the scalability of solvers. This is addressed by appropriate methodologies how properties are formulated. Different commercial providers support different flavors of such methodologies and provide guidance to the verification engineers to make best possible use of the available tools.

Degree of Automation:

The BMC proof techniques are fully automated. The only manual effort arising in BMC-based verification comes from the need to formulate properties. This effort depends largely on the considered verification tasks and is similar to most other property checking techniques.

Coverage:

Coverage is the main limitation of BMC. The functional behavior is analyzed only for a finite number of clock cycles starting from a well-defined state. Hence, there is no global proof of the property, only a "bounded proof" which guarantees correct behavior in a certain time window. On the other hand, BMC can be a very efficient tool to quickly detect counterexamples to a property. Therefore, in industrial practice, BMC quite often serves as a "bug hunting" method.

The trade-off between these criteria is illustrated in Figure 2 by the mark for model checking "BMC".

3.2.2.2.2. Advanced Methods of the Bounded Paradigm.

The limitation of BMC with respect to providing generally valid, "unbounded" proofs often motivates the use of more advanced methods under the bounded paradigm, such as Symbolic Trajectory Evaluation (STE) [45], Interval Property Checking (IPC) [46] and k-Step Induction [47]. All of them have in common with BMC that they consider behaviors within bounded time windows. However, they achieve unbounded proofs by additional concepts. K-step induction combines the local proofs over finite times with inductive reasoning. IPC and STE avoid induction but obtain unbounded proofs for properties formulated over bounded time windows by considering *any state* at the start of the time window. For example, this allows to prove that a processor instruction (executed over finite time) is correctly implemented at the RTL and complies with its ISA specification. IPC and STE have their roots in industrial developments. IPC was developed already in the 1990s within Siemens, the STE development has been driven mostly within Intel.

For the advanced methods of the bounded paradigm and IPC in particular, comprehensive case studies were conducted by German industry to assess the productivity and quality of the design and verification flow. This was compared with state-of-the-art simulation-based approaches. Some of these case studies were conducted by BMBF consortia [48], [49]. The gained insights can be summarized as follows:

Scalability:

Scalability turned out to be less of a problem than originally expected by industrial users. Similar as in BMC the advanced methods of the bounded paradigm map reasoning on sequential systems to combinational problems. Therefore, the scalability of their proof engines is similarly high as for BMC.

With few exceptions, the common verification targets of SoC modules can be proven without substantial problems. For example, large parts of the Infineon Tricore processor, a high-end processor for automotive applications were exhaustively verified by IPC [50]. Almost all properties of this design (with more than 130k lines of Verilog code) were proven, each within few minutes. This is even considered an advantage over simulation-based methods that often run for weeks on clusters of compute servers. Since then, in industrial design and verification projects conducted over the years, such results were confirmed on numerous other SoC modules of different nature, ranging from telecom and automotive to IoT applications.

Degree of Automation:

As in all other property checking methods manual effort is required to create the properties to be verified. In this aspect the advanced methods of the bounded paradigm differ only little from other methods of property checking. Due to the popularity of these approaches, however, investments have been made, especially in industry, to automatically generate the required properties from higher level models, such as within Infineon [51]. This reduces manual effort substantially. (It should be noted that this progress is not intrinsic to the nature of these proof methods itself. In principle, property generation can be combined also with any other approach to property checking.)

Compared to standard BMC, the advanced methods of the bounded paradigm, however, suffer from an additional source of manual effort. This effort is related to generalizing the bounded proof to infinite time. In IPC, for example, considering *any* state at the beginning of the time window is the root cause of *false alarms*. Counterexamples suggest a failing property but are based on starting states of the considered time window that are impossible in the design. The problem can be solved by excluding such “unreachable” states from consideration. This requires refining the starting state of the model by so called “invariants”. Identifying these invariants requires additional procedures and accounts for additional manual effort when employing this paradigm.

The productivity of a formal verification engineer covering a design’s entire functional behavior using IPC or related methods amounts to approximately 2000 lines of RTL code per person month, as measured in a number of industrial case studies [48], [49]. For specific modules, such as processor cores, this productivity can be substantially higher when properties are generated automatically [31].

Coverage:

Coverage is the strong side of the advanced methods in the bounded paradigm. This does not only result from the fact that the proofs obtained are unbounded. Additionally, systematic methodologies exist, especially for IPC, to derive sets of properties from the informal specification such that well-defined coverage metrics are met. The most advanced approaches [36], [37], [31] ensure with formal rigor that the developed property set uniquely determines the entire design behavior. In other words, two hardware designs fulfilling the same such property set are necessarily equivalent. Industrial case studies [48], [49] and today’s industrial experience confirm that a substantial number of design bugs can be identified by formal techniques which were missed previously by simulation.

In spite of the encouraging successes described above, formal verification remains a challenge in industrial practice. One of the main reasons is the effort for property specification. Although in terms of quantitative numbers this effort may not exceed the effort needed for simulation-based approaches, such as for developing testbenches, the different nature of formal techniques and simulation is quite essential. Simulation is mostly a black-box approach. The verification engineer must understand the specification but needs no knowledge about the internal architecture of a design. This has big advantages, for example, when outsourcing verification to external service providers. Formal

verification, by contrast, is a white-box technique; the verification engineer must understand internal design structures to formulate properties. This special expertise is not always available.

A common response to this problem consists in bringing verification closer to design, such as in commercially available Assertion-Based Verification (ABV) where the designers themselves insert assertions into the design. This is often considered a valuable addition to the design flow but does not fully replace the conventional simulation phase.

More recently, however, as safety- and security-critical applications are becoming more common in embedded computing, we can observe a renewed trend towards formal methods. More and more often, the extra costs for formal approaches are considered justified in view of the risks that otherwise are taken. According to the Wilson industrial survey [52] more than 40% of industrial chip design projects currently adopt formal methodologies.

Today, the state of the art in property checking for hardware is defined by commercial vendors such as Synopsys, Siemens EDA and Cadence. In contrast to the software domain, open source tools for hardware verification are rare. Partly, this can be explained by the complexity of design descriptions using hardware description languages. They support a wealth of features which impose high hurdles on creating a front-end technology for any tool. Developing a front end for state-of-the-art property checking demands a great and continuous development effort, while being rather unrewarding from a scientific point of view. Open source contributions are therefore mostly restricted to the backend solvers based on Satisfiability Solving (SAT) and Binary Decision Diagrams (BDDs). Notable exceptions are E-BMC (latest release in 2017) [53] and SymbiYosys [54]. These open-source tools provide multiple proof engines in their backend but support only a subset of SystemVerilog and SVA in the frontend. Important "quality-of-life features" like black-boxing are missing. SymbiYosys allows for extended functionality only for its commercial upgrade. The functionality supported in public domain does not allow for obtaining competitive results on realistic SoC designs. The limitations may partly result from a "chicken-egg" problem. Most academic research groups in hardware verification currently make use of the free (or low-cost) licenses for commercial tools which are superior to currently available open-source solutions. On the other hand, the quality of open-source tools might increase if there was a stronger demand from the research community.

3.3. Theorem Proving

Automated theorem proving is a subfield of mathematical logic concerning the automation of mathematical proofs. All of the techniques discussed for the formal verification of digital systems can therefore be considered automated theorem proving. However, it is useful to distinguish methods based on general purpose theorem provers–which are computer programs capable of assisting a variety of traditional deductive mathematical proofs – from the more specialized property checking techniques for hardware, as described above.

A number of theorem provers are available, differing in their emphasized problem domains and input languages. Most of them are freely available academic developments, e.g., HOL [55], Coq [56], and Isabelle [57]. Complex mathematical theorems have successfully been formalized and proven using theorem provers, including a few previously open problems, e.g., Kepler's conjecture and the four color theorem [58].

A theorem prover requires a formalization of the problem under consideration in the syntax of the tool's language. The computation of the proof is automated as part of an interactive process where axioms and reasoning techniques are specified. Verifying digital hardware based on a general-purpose theorem prover is actually a demanding process. It requires knowledge on proofs of classical

mathematical theorems and, even more critically, expertise to correctly formalize the design and verification target in a representation accepted by the tool.

Yet, industrial activities in theorem proving are alive. After the infamous Pentium FDIV bug [59] especially floating-point units have been granted the attention of such rigid formal methods. Larger chip makers and companies with particularly high requirements on hardware quality support in-house expert teams on theorem proving to support the high end of their verification flows.

Scalability: The scalability of theorem provers can be considered high. They can leverage essentially the full state of the art of proving methods that match the given model and verification target. In addition, theorem proving benefits from the possibility to build a sound stack of models. This is further discussed in Sec. 3.4. Theorem proving allows for abstractions such that verification results obtained at higher levels, thus scalable to large systems, also hold without further proof on refined implementations of these models at lower levels. This is a major strength of theorem proving resulting from the general use of mathematical logic.

Degree of Automation: High manual effort requiring advanced expertise is unfortunately the severe limitation of theorem proving. While golden models of hardware designs are described at the RTL in languages like VHDL or Verilog, theorem provers demand modeling the system and its properties in first order or higher order logic. This is considered disruptive in standard industrial design flows and requires a team of experts.

An industry setting generally requires more automation and support for the standardized description languages than what is offered by general-purpose theorem provers. Theorem provers used in industry today are therefore either tailored to specific niches, such as floating-point units, or they make use of the more automated model checking methods described above. In the latter case, the limitations associated with these more automated techniques apply, as described above.

Coverage: Coverage, in principle, is the great strength of theorem proving. If the entire system is represented by mathematical logic, global verification targets can be fully covered. More than any other method, theorem proving can create connections between different abstraction layers and between hardware and software such that well-defined formal relationships exist between the different levels. This was demonstrated, for example, in the national BMBF projects Verisoft and Verisoft-XT [49]. In the hardware domain, however, the great challenge remains to connect these mathematical models with the concrete descriptions of microarchitectural implementations, in particular the RTL descriptions of the golden models for sign-off. This connection is hard to realize and, if not properly addressed, creates a verification gap because only mathematical models and not the concrete implementations are verified. There is the vision to generate the RTL descriptions for the implementation correct-by-construction from the logic models. However, this faces limited acceptance in the hardware community since design experts wish to leverage their human expertise for design refinements and architectural optimizations. This process relies on standardized hardware description languages such as Verilog and SystemVerilog which are tailored towards this purpose.

3.4. The Role of Abstraction in Formal Hardware Verification

Reasoning in complex digital systems is done through a hierarchy of model descriptions at different levels of abstraction, ranging from transistor-level descriptions through gate-level and RTL to electronic system level (ESL) descriptions. In programmable systems the ISA level serves as the interface between hardware and software. Also the software can be considered at different levels ranging from a hardware-dependent assembler level to hardware-independent descriptions based on high-level programming languages.

Choosing the right level of abstraction for a given verification task is of key importance. Low-level descriptions allow for the verification of local but detailed implementation properties while high-level descriptions facilitate the global analysis of a system's behavior.

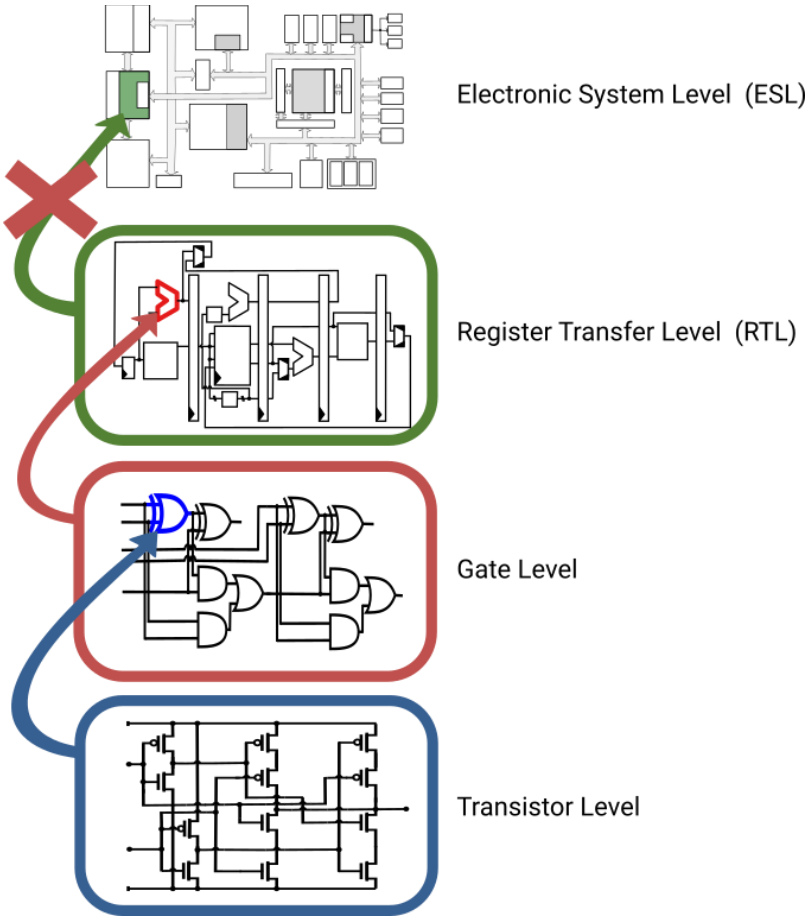

Figure 3: Hardware abstraction levels – bottom up reasoning in verification

For hardware the relationships between abstraction layers are summarized in Figure 3. In verification the reasoning is traditionally bottom-up. A new abstraction level can be used for design and verification only when it stands in a well-defined relationship with the lower abstraction levels. Otherwise, the semantics of the abstract model cannot be understood in terms of the „real" circuit to be implemented. Therefore, the verification at each level must build confidence in the model of the next higher level. For example, starting at the transistor level, verification, such as by analog simulation techniques, is used to achieve confidence in the correct behavior of a logic gate, such as a NAND gate. This trust in the NAND gate is not only important at the transistor level. It allows us to move to the next higher level, the gate level and to analyze the behavior of a large gate netlist only in terms of its abstract logic behavior. By merit of trust in the correct transistor-level implementation of the NAND gate, the results of the *logic* simulation extend also to the transistor-level implementation *without further proof*. We say that the gate netlist description is a „sound abstraction" of the transistor-level implementation. Similarly, by merit of formal equivalence checking, the gate netlist is set into a well-defined formal relationship with the RTL models of a design. This makes also RTL a sound abstraction of the physical circuit implementation and justifies the role of RTL descriptions as the golden model in today's design and verification flows.

Unfortunately, the „chain of trust" breaks when considering even higher levels. Descriptions at the electronic system level (ESL) normally lack a well-defined relationship with RTL. While ESL models are of great value for early design exploration and for parallelizing hardware and software development,

ESL abstractions are not formally sound with respect to RTL. Verification results obtained at the ESL level do not necessarily hold for the RTL design models. This „semantic gap" is one of the main hurdles in today's flows when attempting to lift design and verification to higher levels than RTL. Even at the presence of high-level synthesis, RTL typically remains the point of reference for sign-off verification. Higher-level models, on the other hand, only serve as early "prototypes" of the system.

As pointed out in Sec. 3.3, theorem proving and mathematical logic, in principle, can provide a complete and sound stack of models for both hardware and software descriptions. In fact, theorem proving also has the potential to close the semantic gap between RTL and ESL models. Industrial adoption of such an approach, however, is difficult since, as explained in Sec. 3.3, theorem proving is not easily compatible with today's design flows that are based on other languages, modeling techniques and tools.

In order to overcome this limitation, approaches based on path predicate abstraction [60] and instruction-level abstraction [61] have been developed to close the semantic gap between ESL models and RTL exclusively with commonly accepted and standardized design and verification languages. While such approaches successfully connect transaction-level behavioral models with RTL, they do not cover all common abstraction features of high-level models. Therefore, in summary, leveraging the advantages of high-level hardware models for a better scalability of formal verification of concrete design implementations remains one of the main challenges in today's flows.

3.5. Formal Security Verification by Targeting Non-Functional Properties

Most commonly, security verification for hardware has become part of the verification of functional correctness. This means that the characteristics of conventional verification flows for functional correctness, as described in previous sections, apply also to verifying security features.

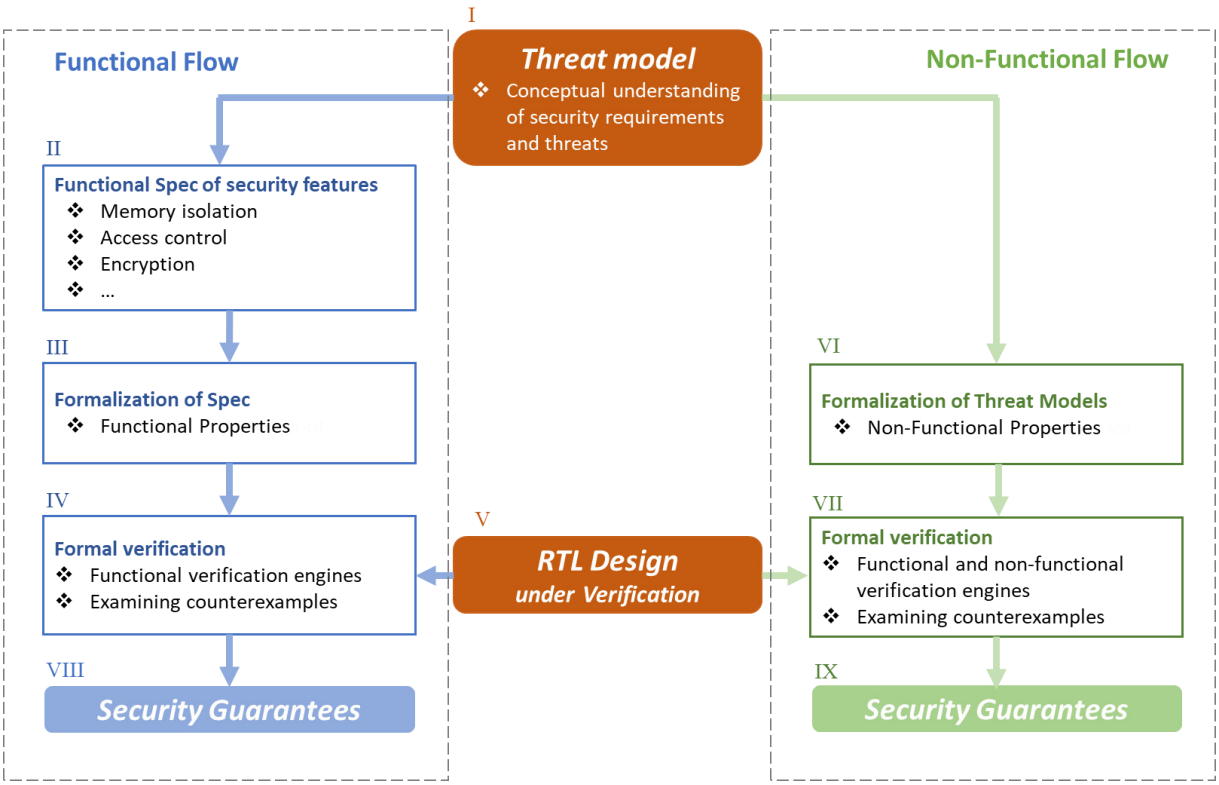

Figure 4: Functional vs. Non-functional verification flow for security

Besides detecting security-critical design bugs (cf. 2.1.1) functional verification techniques can equally detect trojans that are visible at the RTL. Especially techniques using advanced coverage metrics

(cf. 3.2.2.2.2) prove adequate to detect not only any violation of the functional specification but also any additional (undocumented) and possibly malicious functionality that may be triggered only under special circumstances. This holds also for infrastructures used in debugging and test. In special cases, however, tailor-made methods for security analysis, for example for reconfigurable scan networks [62], can be advisable.

In general, if the RTL description is available for analysis, the risk of RTL trojans, as opposed to trojans introduced after sign-off (cf. Lot 4), is usually managed in the same way as the risk of security-critical design bugs. State-of-the-art property checking can be used to verify the design's functionality together with its security features. If the RTL description is not provided, this is a greater challenge. It needs to be addressed in the verification flow by appropriate threat models, as described in Sec. 2.3, taking into account the possibly malicious role of third-party IPs.

The conventional flow for security verification based on functional property checking is shown in the left part of Figure 4. The targeted security features (Box II) typically result from a high-level, architectural perspective. The design specification is extended by an additional *functional* specification (Box III) of these security features which guides their integration into the RTL implementation. This is followed by functional verification procedures (Box IV) which are rooted in established methodologies, as described in Sec. 3.2, for checking an implementation against the functional specification.

It turns out, however, that this classic approach is not always sufficient. Not only does conventional functional verification miss side channels, also the abstract security requirements can be extremely difficult to map to functional specifications, requiring a detailed, microarchitectural understanding of security threats. Therefore, the specification (Box II) itself may miss to cover relevant aspects of the global threat model. Experience shows that, even when choosing security features conservatively, the conventional design process can miss subtle, yet hazardous security gaps and gives rise to the widely spread complaint about a never-ending "patch-and-pray" cycle.

Therefore, another approach to formal hardware security verification has moved into the focus of research which targets security properties directly. This flow is shown on the right side of Figure 4. Instead of developing a detailed (and error-prone) functional specification, this approach starts from the applicable threat model (Box I) and formalizes security requirements rather than detailed functional behaviors (cf. Sec. 2.3). This leads to specifying non-functional security properties (Box VI) which are orthogonal to conventional functional specifications. Since these properties directly target global security requirements without the need of a functional specification for the intended defense mechanisms, these methods have the potential to cover security breaches which are easily missed by the conventional approach. Both conventional solvers for functional verification (cf. Sec. 3.2) as well as specialized solvers, such as [63], can serve as a basis to extend formal hardware verification for such non-functional targets (Box VII).

This relatively new category of formal security verification methods often adopts the view of "information flow tracking" or "taint analysis" which have been popular in the software domain already for a long time. The adoption of taint analysis in the hardware domain was proposed in [64], extending over previous work [65] and the work of [66]. In this paradigm, hazardous information flows are identified between different components of the hardware system and formalized in terms of "information flow properties" or "non-interference properties". This relates to the notion of "hyperproperties" in SW security verification [67]. Formal hardware verification methods based on the same or similar solvers as in Sec. 3.2 are then used to check these properties.

Commercial EDA companies support this non-functional paradigm by providing tools for *formal path analysis*. These tools check whether or not an illegal information flow can happen through a certain suspicious path between two points in the design. This can be effective in checking selected paths of a

design for information leakages, but faces limitations with so far unknown security weaknesses or unexpected information flows. In order to select a suspicious path, the user must rely on a priori knowledge about potential vulnerabilities. Moreover, experience shows that these methods often suffer from scalability issues when considering global information flows in complex designs, limiting their applicability in practice.

A different approach is taken by a recently proposed method, called Unique Program Execution Checking (UPEC) [68], [69], [70]. UPEC does not decompose the problem in terms of structural paths, but "semantically" in terms of possible propagation scenarios for confidential information. This makes UPEC exhaustive with respect to the formalized threat models and has the advantage that no a priori knowledge on possible attacks is required. Technically, UPEC can be understood as a light-weight form of sequential equivalence checking and therefore partly inherits the high scalability of established methods for equivalence checking. UPEC uses IPC as its underlying proof engine. Therefore, similar as for IPC, also UPEC may cause false alarms which have to be removed by the use of invariants. This inhibits the full automation of the UPEC approach.

Other approaches [71], [72], [73], [74] have targeted verifying hardware security by non-functional properties specified at abstract levels above the RTL. InSpectre [72] provided a framework for formal reasoning about different security countermeasures, using a formal microarchitectural model with speculative and out-of-order execution semantics. Also UCLID [73] can be used to verify the security of different microarchitectural design schemes. CheckMate [74] is a program-synthesis based technique to synthesize attacks based on certain execution patterns and abstract microarchitecture models. The synthesized attacks can be used to test the security of the system. This class of techniques provides important insight into the design flow by an early capturing of security problems and consequently delivering RTL designs with higher quality leading to lower RTL verification efforts. However, these techniques operate on abstractions that lack formal soundness with respect to RTL, as discussed in Sec. 3.4. Therefore, they can miss security-critical details and cannot serve as tools for sign-off verification.

3.6. Language-based HW security

Another line of research is language-based security. This paradigm supports hardware design with formal guarantees for security properties and advocates the use of new security-driven hardware description languages. These languages usually use a type system that forbids explicit (direct value assignment) and/or implicit (conditional assignment) information flows between certain security types (security labels), according to a security policy.

Caisson [75] and Sapper [76] are examples of hardware description languages for security. They enable the user to design hardware with the desired information flow properties. In Caisson and Sapper, the designer must annotate each register in the design with security types (labels). The code can be checked with the designed type system for security violations. Caisson uses a static type system forcing the designer to duplicate the logic for those information paths that violate the type system. Sapper improved over Caisson by using dynamic types. This approach can be linked to theorem proving (cf. Sec. 3.3). The VeriCoq-IFT framework [77] automatically converts the design to the Coq formal language [78] and generates a security property theorem based on an information flow policy. Although this removes the need for using a new hardware description language, the designer still needs to annotate the Coq code with security types.

All of the above languages employ a conservative information tracking scheme in their type system, which creates an overestimate about the possible information flows. This bears the risk of overly conservative and possibly inefficient designs.

SecVerilog [79] extends the Verilog language with a security type system. The designer needs to label storage elements with security types which allow for enforcing information flow properties. SecVerilog implements precise information flow tracking by using predicate analysis and constraint solving. This solves the overestimation problem of other languages. Similar to SecVerilog, ChiselFlow [80] is proposed to extend the Chisel language [81] with a security type system. ChiselFlow partially automates the labeling process to mitigate the incurred manual effort. Although the use of Verilog and Chisel as the base language eases the adoption of the method, the labeling process is complicated and security violations are hard to debug in these approaches. Furthermore, the designer may need to adapt the labels in the design in order to verify different security properties. A considerable effort also results from the need of a system-wide labeling of the RTL design. Such changes of design methodologies in established design teams are usually hard to implement.

HyperFlow [80] is an example of an SoC completely designed (and formally verified by construction) using a special, security-driven hardware description language. The non-interference property implemented by HyperFlow is a strong security measure that can block many attack scenarios in a computing system, including both classical side-channel attacks (cf. Sec. 2.1.2.1) and attacks based on transient execution side channels (cf. Sec.2.1.2.2). However, this security guarantee comes at a high price in performance and memory overhead, and imposes drastic changes in the overall design flow, from hardware design to operating system development.

3.7. Formal Verification at the Hardware/Software Interface

At the hardware/software interface strategies largely depend on the considered vulnerabilities. We can distinguish three cases:

Security-critical design bugs (cf. Sec. 2.1.1): Generally, it is not desirable that software developers have to take any hardware issues into account that are related to implementation details. Therefore, security-critical design bugs associated with module bugs, such as a poor implementation of physical memory protection, are typically addressed exclusively in the hardware. The situation becomes more difficult, however, if security gaps are found which concern the interaction between different modules and their integration into the system. Such gaps are often related to specific firmware configurations and impose restrictions on software development. Hardware fixes that avoid any restrictions on the software may be possible but can cause significant overhead in the system. Therefore, in practice, trade-offs must be considered where restrictions in the software are balanced against the overhead of special hardware mechanisms for security. For formal security verification at the hardware level, software restrictions must be modelled adequately as assumptions for the hardware proofs. Conversely, from counterexamples to the hardware security properties software assertions must be formulated and proven at the software level. Such a flow has been demonstrated for example in [70].

Security violations by ISA-visible microarchitectural side channels (cf. Sec. 2.1.2.1): ISA visibility is the key characteristic of many classical microarchitectural side channels. In most cases no special measures are taken at the hardware level. Security is ensured by measures exclusively at the software level that prohibit security-critical software components from leaving exploitable footprints. Many of these techniques are subsumed under the notions of *data-oblivious programming*. This well-known paradigm enforces that run time, resource usage and memory access patterns of a security-critical program are independent of confidential data. Formal hardware verification plays only a minor role in this context. However, it must provide the guarantee for the hardware that the security of a data-oblivious program is not compromised by data-dependent timing of hardware operations [63], [82].

Security violations by transient execution side channels (TESs) (cf. Sec. 2.1.2.2): This is where the bulk of recent research related to the hardware/software interface has been conducted. Several attempts are made for formally defining TES attacks at the software level to enable effective software

verification against Spectre attacks. The main goal of such verification techniques is to find the exploitable "gadgets" in the software in order to apply proper hardening, e.g., by inserting synchronization barriers such as fence instructions. Gadgets are exploitable instruction sequences existing within the kernel software, triggered by the attacker and running in privileged mode within a victim process.

The approach of [83] is among the first attempts to developing a mathematical model for microarchitectural side channels and TES attacks from a software perspective. The model provides a good basis for understanding the security implications of microarchitectural optimizations. However, it does not cover all possible TES attacks and misses attack scenarios in which the secret is leaked through the timing of the victim program rather than its final state (cf. the Spectre-STC attack in [69]).

As elaborated in Sec. 2.1.2, transient instructions are not observable by analyzing the software based on the semantics provided by the ISA. As a result, new techniques are proposed to verify software against speculative attacks. Most of these techniques extend the existing software verification approaches by augmenting the ISA semantics with some abstract annotations of underlying hardware features [84], [85], [86].

The requirement of security against speculative attacks can be formalized based on the notions of secure speculation [87] and speculative non-interference [88]. The basic idea is to verify at the software level whether there is any security violation (w.r.t. a defined security policy) that can only occur if the program is executed using the speculative semantics. Therefore, these formulations are effective for verification at the software level. However, secure speculation, speculative non-interference, and other efforts, such as [89] and [90], for formalizing TES attacks do not provide a generic method for detecting TES attacks in hardware since they do not take the microarchitecture into account. Transferring software security concepts to hardware verification is not trivial, as shown at the example of constant time execution by [63].

New hardware/software contracts have been developed in [91] as a framework to reason more precisely about what information hardware leaks and what consequences this has for software security requirements. This provides a framework to evaluate security at the software level, without leaving a gap due to certain microarchitectural features. It has not been examined yet, however, how the hardware side of the high-level contracts can be mapped to the hardware implementation by RTL properties and whether these properties scale for state-of-the-art commercial property checking.

4. Research Needs and Recommendations

In the context of cybersecurity, the overall goal of next-generation design and verification flows for hardware must be to re-establish hardware as a trustworthy foundation for all software executions of the IT system. This is of particular importance for all hardware security features, such as cryptographic functions, that constitute the hardware-implemented root-of-trust of a system and lay the foundation for the chain of trust at the software levels.

In view of this global objective, the preceding analysis of commonly addressed hardware security targets and the state of the art in formal hardware verification shows a rather patchy picture of currently available solutions.

In the following, we sketch research areas for formal hardware verification at the microarchitectural level which can contribute to reaching the above overall goal. We begin with a brief look at the possibility of linking these research activities to open-source initiatives.

4.1. Research Contributions to Open-Source Initiatives

Making research results available in public domain is highly desirable. However, as elaborated in Sec. 3.2, open-source tools for formal hardware verification are hardly available and there are significant hurdles for the research community to develop them. Fortunately, this is not a show stopper for supporting open-source developments also in the field of formal security analysis of hardware. A large part of the developments described in the following can be made available to the public. In fact, the following research, to a large extent, can be considered in separation from commercial tools. The new methods can be designed to rely on commercial tools only in their backends as generic verification engines, i.e., the new methods can be agnostic to these standard tools. Consequently, a new tool for formal security analysis can be made public, however, with the restriction that it is fully functional only when combined with one of the commercial property checkers. This is not a severe limitation since standard (commercial) property checkers are available today throughout industry and in academia.

The hardware platforms for the research directions to be described can largely be retrieved from various open-source hardware initiatives. The evolution of the open-source ISA RISC-V [92] is certainly a game changer for academic research. There is a vibrant community for RISC-V hardware development which continuously releases new cores and platforms of varying complexity. They provide an excellent basis for the hardware demonstrators needed to evaluate the research results in formal hardware verification for security. Conversely, research results in formal security verification can contribute to the hardware development by identifying security holes in existing designs, proposing fixes or providing secure designs.

4.2. Research Challenge – Tools for Formal Security Analysis

Tools for formal security analysis at the microarchitectural level are currently rooted in formal methods for functional correctness and for path analysis. Significant extensions to this state of the art are needed to meet the requirements of future design flows for security-critical systems.

4.2.1. Formalization of Threat Models

Considering the vast variety of deployment domains for SoCs, there is an ever-expanding attack surface. Hardware verification engineers need to consider a large diversity of threat models (cf. Sec. 2.3) to cover different use case scenarios and each threat model may impose unique challenges on the verification task.

This results in great research need to create *re-usable formalizations* of the different threat models, ranging from confidentiality violations by transient execution side channel attacks to integrity violations by physical fault injection or third-party IP integration (cf. Sec. 2). The relevant threat models depend on the type of hardware modules (in-order core, out-of-order core, accelerator, etc.) and their integration into a larger design (System-on-Chip, Network-on-Chip (NoC), Multi-Core, etc.). A unified approach is desirable which covers all relevant threat models based on standardized languages, such as SVA, to integrate different verification targets into today's verification plans without disruption. The resulting verification IPs should be of generic value and can be made available in public domain.

Importantly, the formalization of threat models must be done as globally as possible. Rather than targeting specific and known attack situations, global verification targets must be formulated that cover a wide spectrum of vulnerabilities and attacker profiles. This increases the chance to cover also the "unknown unknowns", i.e., so far unknown vulnerabilities that potentially exist in the design. Exploiting a priori knowledge on existing attacks must be avoided as much as possible in the formalization of threat models.

Figure 5 provides an estimate for the failure risk and time horizon of this research. The arrows denote dependencies in the research flow. Although the threat models may vary greatly for different

processor architectures, the risk and time horizon of this research hardly depend on the type of processor. This is because threat models are formalized globally and must abstract from the specific implementation, such as the type of processor pipeline. However, research efforts and research risk for formalizing threat models depend largely on the size and architecture of the SoC which is composed from different cores and other modules. Formalizing the threats in such composed systems is widely a new field of research.

We consider confidentiality a prerequisite for integrity. If confidentiality is violated, for example, a key can be stolen and used by an attacker to violate the integrity of the system. Therefore, in Figure 5 we only distinguish the two cases that confidentiality or both, confidentiality and integrity, are covered by the threat model to be formalized.

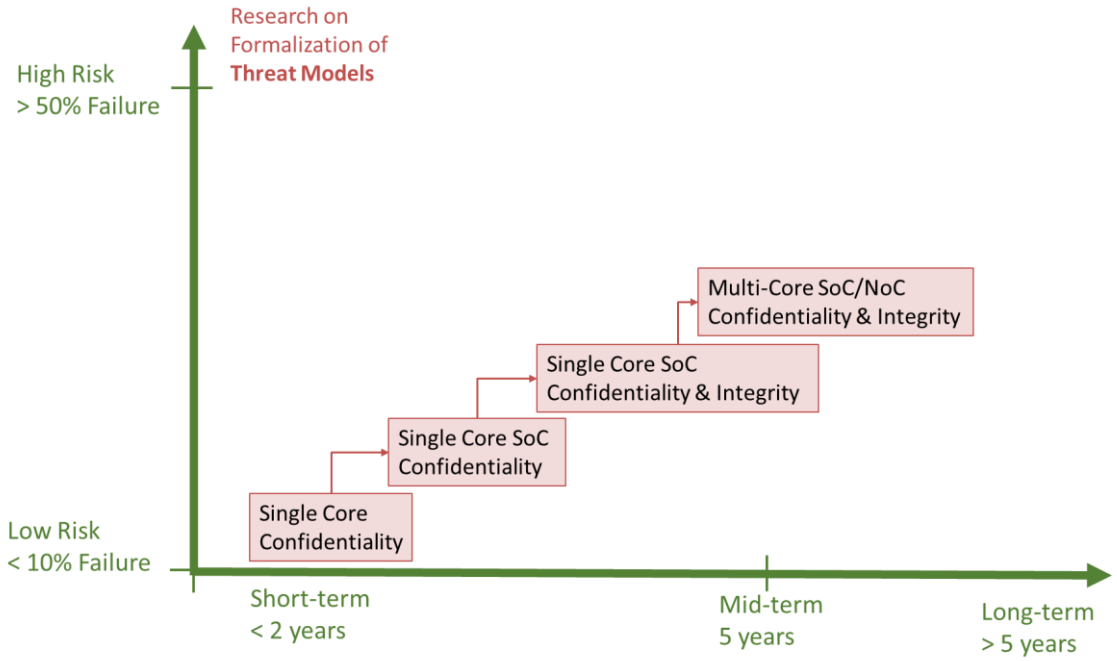

Figure 5: Risk and time horizon for research on formalization of threat models

4.2.2. Functional Verification for Avoiding Security-Critical Bug Escapes

Late detection of hardware security flaws can incur tremendous costs. Patching a design is either very expensive, in terms of sacrificing performance and limiting functionality of the design, or simply impossible for technical reasons. This calls for *new functional verification* techniques targeting hardware security. Rather than being exhaustive with respect to a complete functional design specification, the new tools must be *exhaustive with respect to a well-defined threat model* (cf. Sec. 3.2 and Sec. 3.5) and deliver well-defined security guarantees. The new tools must be scalable and, at the same time, amenable to adoption by current industrial hardware design flows.

While state-of-the-art functional verification of a design, in principle, also avoids security-critical bug escapes, the manual effort for such an exercise is prohibitively large, especially if not only a single core but an entire SoC with multiple peripherals or a multi-core architecture is considered (cf. Sec. 3.2.2.2.2). The new methods to be researched may draw advantages from prioritizing security objectives over other aspects of functional correctness so that the formal analysis can be tailored towards security with the benefit of reducing manual effort and increasing scalability. Note that targeting non-functional properties may still detect functional design bugs. In particular, the methods directly targeting the (non-functional) objective of security (integrity, confidentiality), as shown on the

right side of Figure 4, have the potential to detect all *functional* design bugs that violate the security target.

Figure 6 provides an estimate for the risk and time horizon of this research and depicts research dependencies. Confidentiality in single cores is already fairly well understood. Formalizing the verification target can build upon a large body of previous research (cf. Sec. 2). A formal proof of functional confidentiality is currently doable for in-order cores of medium complexity. Even for such simple cores, however, additional research is still advisable and can concentrate on better trade-offs between scalability and the degree of automation (under exhaustive coverage of the proof target). For out-of-order cores, by contrast, proving the absence of confidentiality-violating bugs is very difficult. In spite of substantial efforts in the past, the authors are not aware of any successful attempt to fully verify an (industry-scale) out-of-order processor against its functional specification. Therefore, proving the functional correctness with respect to confidentiality, even if this is a simpler problem, bears significant research risks and demands a long-term research effort.

Besides targeting cores there is research needed pursuing the target of confidentiality and integrity in SoCs consisting of multiple components. Depending on the type and complexity of these components the challenge for research varies. Considering integrity on top of confidentiality increases the challenges. The associated dependencies, risks and time horizon are sketched in Figure 6.

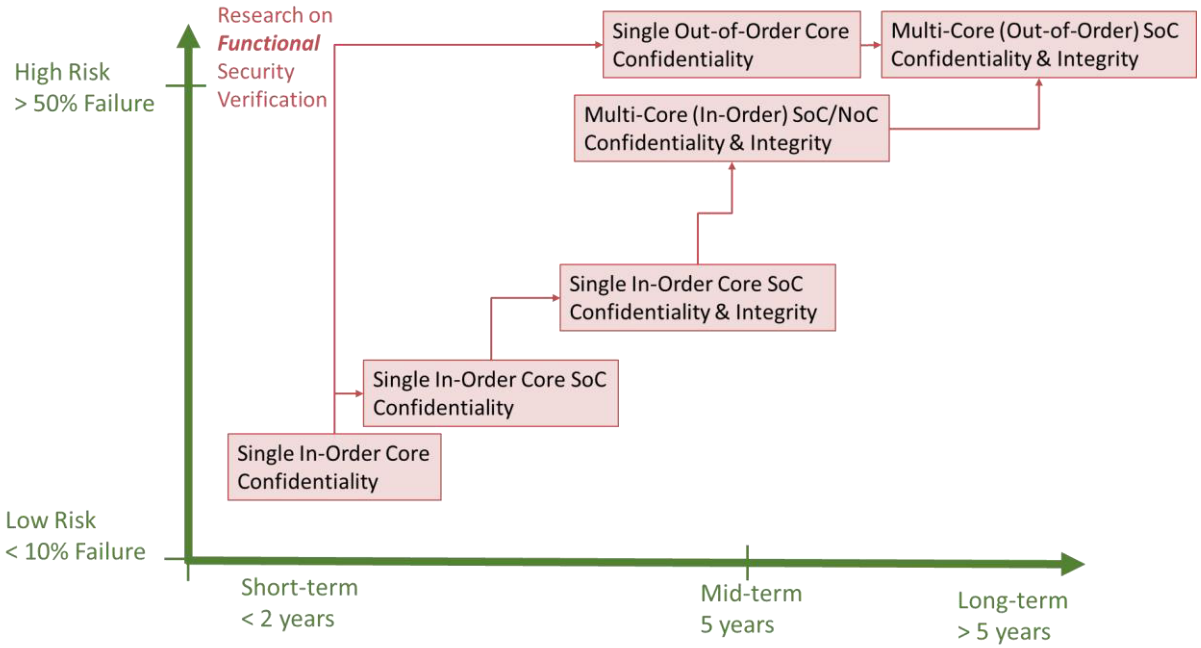

Figure 6: Risk and time horizon for research on functional security verification

4.2.3. Non-Functional Verification for Detecting Timing Side Channels

In addition to the functional or explicit information leakages that violate the security requirements, also implicit information flows through timing side channels must be considered (cf. Sec. 2.1.2). This complicates the verification process significantly because the functional specification, which is untimed, cannot cover such requirement. Therefore, there is a lack of proper specification techniques for security against timing side channels (cf. Sec. 4.2.1). Most commercial formal verification tools were designed to verify functional properties and are not suitable for checking such non-functional requirements.

New formal tools are needed which target the threats by timing side channels in a systematic way. This research can build upon initial successes in academia to detect transient execution side channels but must extend the scope to all other side channels relevant at the microarchitectural level. This research can take different facets depending on the type of the hardware system under verification:

Single Processor Cores: The bulk of previous research on microarchitectural side channels has addressed single cores. This is justified by the fact that the single programmable hardware components are the main source for side channels at the microarchitectural level. The type of core plays a role for the class of side channels that must be considered. While high-end out-of-order cores are infamous for their vulnerability to transient execution side channels, other types of side channels are linked to cache-based systems and depend to a lesser degree on the complexity of the core itself (cf. Sec. 2.1.2). Therefore, research is needed *to systematically address all timing side channels* and to create methods capable of handling different architectures ranging from simple in-order processors without speculation to out-of-order processors with speculation.

SoCs and Multi-Core Systems: Only little research has been reported on detecting side channels beyond single cores. To this day, it is poorly understood whether the nature of side channels and the methods for their detection substantially differ when moving from single cores to systems with multiple modules and processors. For transient execution side channels, the common conjecture is that vulnerabilities can be identified by exclusively analyzing cores individually. Such an approach, however, has never been formally certified and can be insufficient, especially, for other types of side channels. Research is needed to better understand the role of timing side channels in large systems composed from numerous modules. This new understanding must spark research on new methods for detecting cross-modular vulnerabilities (cf. Sec. 4.3.1).

For a given hardware design and its deployment domain, the relevance of side channels is often a controversial issue. While formal methods detect such vulnerabilities, they are not adequate to assess their relevance. Research is needed, on how to combine formal methodologies with a risk assessment, for example, based on a *quantitative assessment* of side channel occurrence.

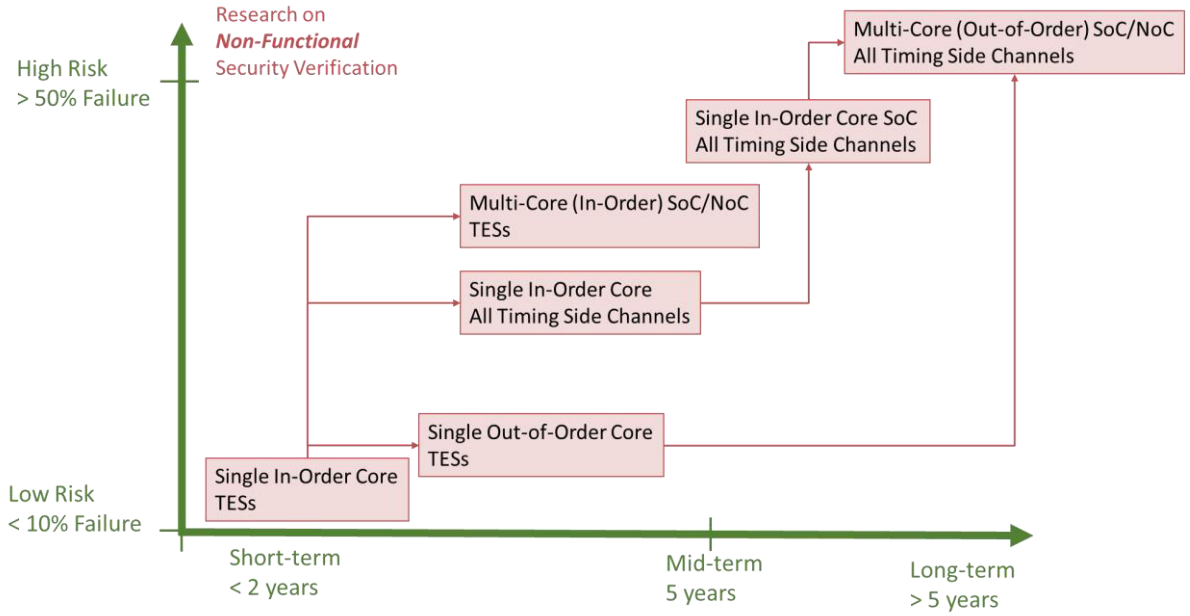

Figure 7: Risk and time horizon for research on non-functional security verification

Figure 7 provides an estimate for the risk and time horizon for research on non-functional security verification. Note that no distinction between confidentiality and integrity is needed since timing side channels are only a topic for confidentiality. Furthermore, TESs are a topic for cores only. Therefore, TESs in single-core SoCs demand the same analysis as the single core itself. For other timing side channels, this is an open question. The research efforts and risks therefore grow, especially if the scope of the analysis is extended to all timing side channels and to SoCs with increasingly complex architectures.

4.2.4. Cost Estimate for Research

We sketch research work packages which are motivated by the research goals formulated above. The work packages have three components each relating to one of the three research directions shown in Figure 5 to Figure 7. We also distinguish between systems of medium and high complexity. Research efforts are denoted in terms of person years (PY). The given numbers are preliminary estimates. Actual research costs may deviate from these figures depending on the employed methods and demonstrators and depending on synergies within a collaborative research consortium. We further annotate each work package with an estimate for the Technology Readiness Level that can be achieved by the described research.

Work Package 1: Confidentiality in Cores

In-order cores (RISC-V examples: Ibex, RocketChip, Ariane):

- Formalization of threat models (1 PY): For single cores, research can build upon academic results covering functional bugs and transient execution side channels. Research is still needed to cover other microarchitectural side channels and microarchitectural footprints.
- Functional Verification (1 PY): There is a strong academic and commercial basis for functional confidentiality verification. However, improvements are to be explored to reduce manual effort (e.g., by flow on right side of Figure 4: Functional vs. Non-functional verification flow for security).
- Non-Functional Verification (3 PY): Commercial state of the art is very limited. Academic results are available for transient execution side channels. The main challenge is to extend the scope of formal analysis to other microarchitectural side channels.

Total: 5 PY
Technology Readiness Level: 4

Additional research on Out-of-order cores (RISC-V example BOOM):

- Formalization of threat models: Formalization of threat models is largely unaffected by the type of core. No extra effort.
- Functional Verification: Gap-free functional verification of out-of-order cores is an unsolved problem, even if only security targets are considered. Since no activities or new ideas in this area are currently visible in the worldwide research community, no cost estimate is given.
- Non-Functional Verification (2 PY): Commercial state of the art is very limited. Some academic results are reported for TES attacks. Lifting them to higher maturity justifies 2 PYs of effort. Extending the analysis to other side channels is an important research target justifying further research effort.

Total: 2 PY
Technology Readiness Level: 4

Work Package 2: Confidentiality in SoCs

SoCs of medium complexity, including multiple modules of different types
(RISC-V examples: OpenTitan, Pulpissimo)

- Formalization of Threat Models (2 PY): Formalization of SoC-wide confidentiality matching the needs of formal technology still needs investigation. This is especially true if all microarchitectural side channels shall be covered and if multi-core systems are considered.
- Functional Verification (2 PY): Commercial state-of-the art requires too much effort for gap-free SoC-wide confidentiality verification. New approaches, for example, based on the right part of Figure 4, must be explored.
- Non-Functional Verification (2 PY): The main challenge here is to understand and cover all microarchitectural timing side channels relevant in an SoC. Previous research using formal verification for this purpose is very sparse.

Total: 6 PY
Technology Readiness Level: 4

SoCs of high complexity, including out-of-order multi-core, Network on-Chip (NoC)
(RISC-V examples: none)

- Confidentiality of high-end SoCs should be researched in conjunction with integrity because of similar complexity challenges for the formal methods. No extra cost estimate for confidentiality is provided.

Work Package 3: Integrity in SoCs

SoCs of medium complexity, including multiple modules of different types
(RISC-V examples: OpenTitan, Pulpissimo)

- Formalization of Threat Models (2 PY): Almost no research is available. Exploring possible attacker profiles and attack scenarios in multi-module SoCs is demanding since case studies on multi-module designs are needed.
- Functional Verification (2 PY): The state of the art for integrity verification is insufficient but can be developed by extending methods originally built for confidentiality.
- Non-Functional Verification: Side channels are only a topic for confidentiality.

Total: 4 PYs
Technology Readiness Level: 4

Additional research on SoC of high complexity, including out-of-order multi-core, NoC
(RISC-V examples: none)

- Formalization of threat models: Formalization of threat models is largely unaffected by the type of core so that high-end designs do not require extra research effort.
- Functional Verification (2PY): No research on formal integrity verification (RTL) for high-end SoCs has been reported yet. First concepts can be developed by extending the notions for SoCs of medium complexity in combination with abstraction.
- Non-Functional Verification (2PY): Extending confidentiality as a pre-condition for integrity in high-end SoCs is relevant and justifies research effort for developing first concepts.

Total: 4 PYs
Technology Readiness Level: 2

For high-end SoCs the availability of demonstrator designs is very limited. Therefore, extra effort for building demonstrators may be required.

4.3. Research Challenge – Flow and Methodology

Running a single tool on a specific design or design component can never lead to global security guarantees formulated for a large hardware system and its interface with software. Instead, such guarantees must result from a security-driven flow that combines the results of different methods and tools across components and design layers. Such extensions to today's flows face the following challenges.

4.3.1. Cross-Modular Security Flow – Horizontal Dimension

Hardware designs usually consist of several interconnected modules. Many security issues are introduced into the system through the integration of these components, and a vulnerability in one component may only be exploited through its communication with other components (horizontal dimension). Detecting such vulnerabilities requires analyzing information flows across multiple components which is usually a computationally expensive task for formal verification techniques (cf. Sec. 4.2).

This calls for new scalable verification *methodologies* exploiting specific advantages of different methods to cover system-wide security for a given threat model. The new methodologies are required to combine the results of different tools in order to compose global guarantees on the entire system. Formal verification in such a setting can be based on sound abstraction techniques over different stages of the design flow.

In these new methodologies, different formal verification tools are applied to different design components and at different stages of the design flow, however, always under the strict regime of formally sound compositions and abstractions (cf. Sec. 3.4). This creates new requirements not only for the single tools but the entire flow. For these reasons, the risks for research targeting system-wide security issues are generally relatively high. A change of methodology in an industrial setting is associated with high costs and demands strong justifications.

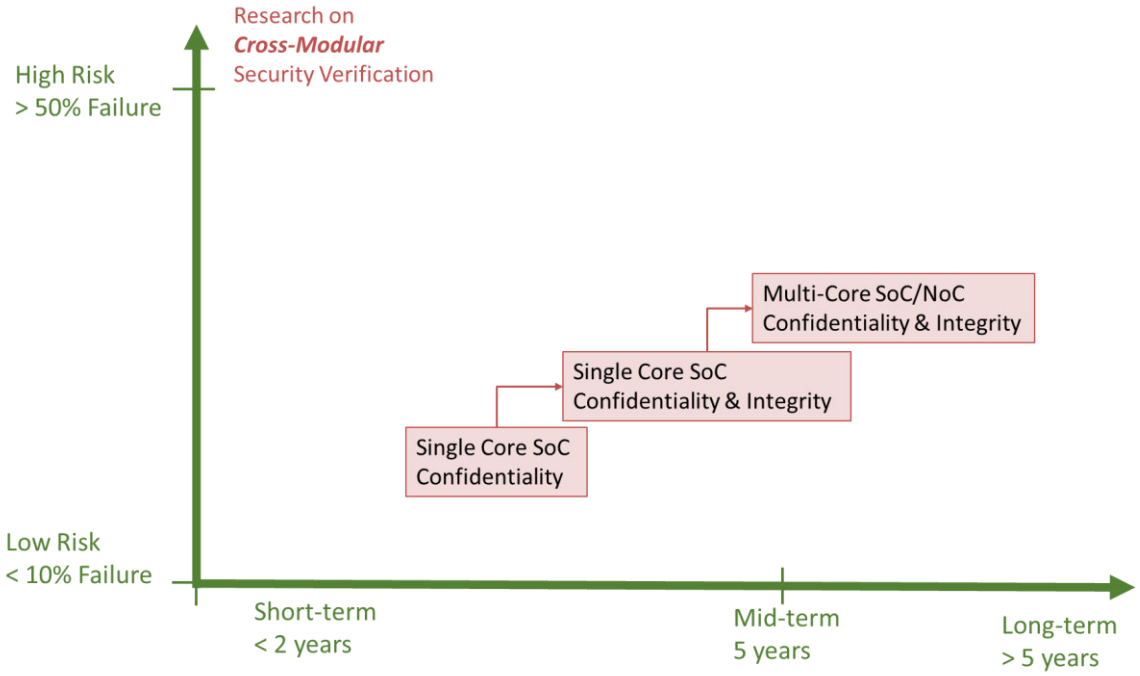

Figure 8: Risk and time horizon for research on a cross-modular security flow

Figure 8 provides an estimate for the risk and time horizon of this research. It ignores the risks and complexity of cross-modular tasks for individual tools, as they have been described in Sec. 4.2, but orthogonally addresses these issues for flow and methodology.

While composition and sound abstraction have been explored in the software domain using theorem proving, this is mostly a completely new research area in hardware, especially, if such an approach is intended for property checking using standard languages (cf. Sec. 3.4). In the context of confidentiality, security-related integration conditions of SoC modules may be easier to formulate than for the general case of integrity. Combining this with sound abstractions further aggravates the challenge. This is reflected in Figure 8.

4.3.2. Hardware/Software Interface – Vertical Dimension

Many hardware security issues are only exposed if triggered by specific interaction between hardware and software. For example, a bug in the access control mechanism of the bus may only be triggered if the firmware configures the access control unit in a specific way. Similarly, access control mechanisms in the hardware may rely on the operating system to enforce authentication of other parties demanding access to security-critical data or computing resources. Formal verification of such mechanisms demands proper modeling of hardware/software interaction. This can be based on compositional principles (e.g., proven software guarantees are assumptions for hardware proofs) or the even more demanding approaches by hardware/software co-verification.

Proof techniques across the HW/SW stack (vertical dimension) are of heterogeneous nature. For example, theorem proving is used at the software level while property checking may be the method of choice for hardware. Therefore, obtaining global security proofs across the hardware/software boundary is a great challenge. Compositional procedures based on well-defined assumptions and guarantees at the interface are generally more tractable for analyzing global behavior than co-verification creating a joint computational model for both hardware and software. Therefore, co-verification can only be applied locally but, on the other hand, may have benefits over compositional approaches by avoiding complicated interface definitions. In those cases they can be an optional addition to the flow (dashed arrow in Figure 9). We may distinguish two main scenarios.

> Interface between hardware and low-level software (firmware): Research is needed to formally verify security for hardware in combination with the (often huge space of) possible firmware configurations in SoCs. Modeling these firmware configurations correctly is key to detect hardware bugs, for example, in local access control mechanisms. Both compositional methods and co-verification may be worthwhile research topics, while the latter are more demanding and bear more risk due to limited scalability.
>
> Interface between hardware and system software (operating system): A key research challenge is to extend the software-level security guarantees provided by an operating system to the entire hardware/software system. This means that neither side channels nor hardware bugs must compromise the security guarantees of the operating system. For example, starting from the fully verified open-source seL4 microkernel [93], the question must be answered what security objectives have to be met by the hardware and how this can be formally proven. This involves a global view on the system considering the correct and secure integration of different hardware modules and their interaction between each other and the system software. Co-verification is hardly tractable for this purpose. A compositional approach seems more realistic but demands high effort from all involved parties and bears substantial risk.

The risk and time horizon for research on the hardware/software interface is shown in Figure 9.

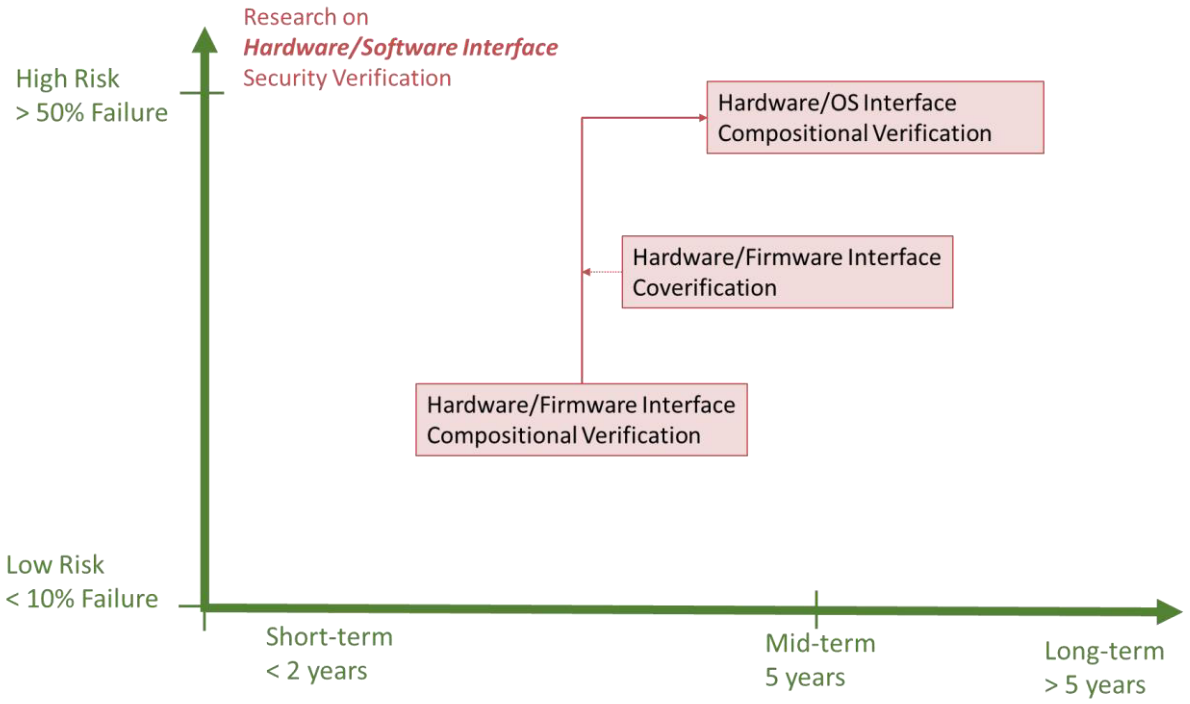

Figure 9: Risk and time horizon for research on the hardware/software interface

4.3.3. Verification-Driven Design – Secure-by-Construction Design

Experience in industry and academia shows that most SoC hardware designs suffer from numerous security flaws based on both microarchitectural timing side channels and functional design bugs. Fixing design bugs is usually an ad hoc process which solves the problem by making design changes and/or communicating possible restrictions for the software layer with the software developers (cf. Sec. 4.3.2). Fixing timing side channels is a more demanding procedure. Advanced security features have been proposed, such as those based on information flow tracking [94], [95], that promise effective measures against these vulnerabilities. However, this comes with significant costs: the manual RTL design effort increases drastically and the new architectures come with a significant hardware overhead that so far has only been estimated at the implementation-abstract gem5 level. Only few RTL architectures with security guarantees for side channels have been proposed. A notable exception is the approach of [80] which, however, is demonstrated only for relatively small designs, requires new and laborious design methodologies, demands new operating systems and entails a significant performance overhead.

Clearly, research is needed to explore new security architectures at the RTL. Within the scope of Lot 2, formal security analysis can make a major contribution to developing new design methodologies leading to new security architectures. Formal verification precisely and exhaustively determines *all* attack scenarios that are relevant under the specified threat model. This knowledge can be very valuable to i) determine the root causes of the security weakness and ii) to derive fixes that avoid excessive conservatism. Note that, without the knowledge of formal tools, current measures for security often employ “blanket fixes” that cover a large (but not fully understood) spectrum of weaknesses.

Research is needed to develop new formal verification-driven methodologies for secure-by-construction hardware designs which

- automate the design and customization of security features, thus avoiding excessive design costs,

- provide pinpoint mitigations to security flaws, thus avoiding unneeded hardware overhead and loss of performance for the secure computing system,
- provide formal guarantees for the final design with respect to the relevant threat models and
- integrate seamlessly into existing design flows, thus avoiding a disruptive change of design methodologies.

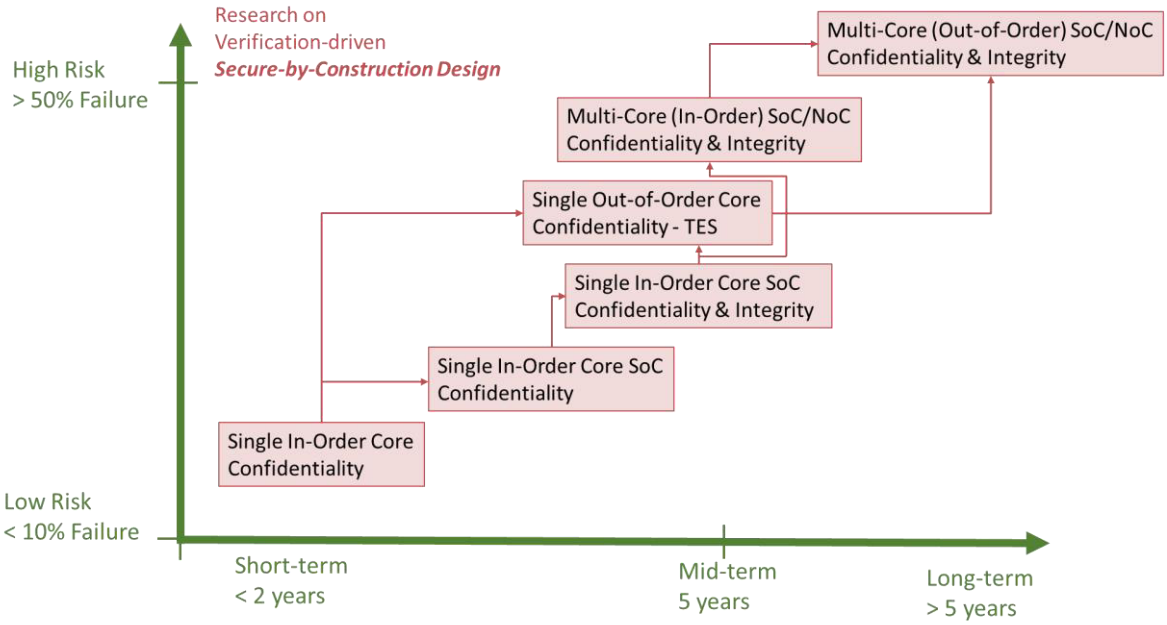

Figure 10: Risk and time horizon for research on verification-driven secure-by-construction design

The risks and time horizon of this research, as sketched in Figure 10, are in line with those of the formal tools and methodologies being employed (cf. Sec. 4.2) and grow with the complexity of the considered hardware systems and security architectures. Note that also here, broad security guarantees for multi-core systems involving out-of-order cores mark the long-term goal of security research. However, significant progress seems possible in shorter time frames for problems of lower complexity which still are highly relevant. For example, the problem of detecting and fixing transient execution side channels in out-of-order cores was determined in Sec. 3.1 to mark the beginning of a new era in awareness for hardware security. In fact, building upon the state of the art described in Sec. 3, there is a realistic chance that solutions to this problem are within the reach of mid-term research projects.

4.3.4. Cost Estimate for Research

We sketch research work packages which are motivated by the research goals formulated above. The work packages have different components relating the research agendas depicted in Figure 8 to Figure 10. The proposed WPs can be based on the tools and verification methods, as described in Sec. 4.2.. Their research costs are not included in the costs for the following WPs. The following costs cover the additional methodological aspects of Sections 4.3.1, 4.3.2 and 4.3.3.

The research efforts are denoted in terms of person years (PY). The provided numbers are preliminary estimates. Actual research costs may deviate from these figures depending on the employed methods and demonstrators and depending on synergies within a collaborative research consortium. We further annotate each work package with an estimate for the Technology Readiness Level that can be achieved by the described research.

Work Package 4: Formal Verification Flow for SoCs

- Horizontal (cross-modular) dimension (2PY): If the quality and type of verification methods differs between different modules (e.g., formal for core, simulation for peripherals, unknown for third-party IPs), methods are needed that compose the verification results to provide global security guarantees. Depending on the individual methods appropriate assumptions have to be made and sound abstractions can be applicable. A general methodology under these circumstances can be developed. For high-end SoCs it imports the risks of the verification methods described in Sec. 4.2.

- Vertical (HW/SW) dimension (3 PY): A cross-layer approach to re-establish hardware as root-of-trust for the entire system is of particular relevance. A compositional approach can be researched which creates a chain of trust by matching proof assumptions at the hardware level with verification targets at the software level. The nature of this interface depends highly on the type of verification method used at each level. Coverification may optionally (dashed line in Figure 9) reduce the manual effort for defining this interface. For high-end SoCs this research imports the risks of the verification methods described in Sec. 4.2.

Total: 5 PY
Technology Readiness Level: 4 (for systems of medium complexity)

Work Package 5: Verification-Driven (Secure-by-Construction) Design:

Secure-by-Construction Design driven by formal methods is a completely new field of research. It can be divided into two sub-fields addressing cores (and in particular side channels in cores) and addressing entire SoCs (and in particular their correct access control mechanisms).

- Cores (7 PY): Research must address the tasks described in Sec. 4.3.3. In particular, ISA-invisible side channels are a relevant research target since they cannot be fixed at the software level. The core produced by this methodology should provide formal guarantees with respect to the data-oblivious programming paradigm. Functional bugs can be addressed for cores of medium complexity. (For out-of-order cores this topic bears very high risk, as explained in Sec. 4.2.2.) This research involves substantial design efforts for complex cores, for advanced architectures of information flow tracking and other security features.

- SoC (8 PY): When considering entire SoCs consideration of side channels is less important. If side channels are eliminated in the programmable units (cores and accelerators) the main task that remains is ensuring functional correctness of each security feature and the correct interplay between all security mechanisms distributed over the SoC. Access control mechanisms for local components and the entire SoC must be designed that comprehensively cover information flows in the entire chip. For RISC-V, for example, such mechanisms (beyond IOPMP) are not yet available and must be designed from scratch. In spite of this large effort, research on a systematic methodology providing such mechanisms together with global security guarantees for the SoC is highly advisable.

Total: 14 PY
Technology Readiness Level: 4

5. Literature

[1] S. J. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. Fletcher and D. Kohlbrenner, "Augury: Using data memory dependent prefetchers to leak data at rest," *IEEE Symposium on Security and Privacy (SP),* p. pp. 1518–1518, 2022.

[2] M. Gross, N. Jacob, A. Zankl and G. Sigl, "Breaking trustzone memory isolation through malicious hardware on a modern FPGA-SoC," in *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*, 2019.

[3] "Common Weakness Enumeration," [Online]. Available: https://cwe.mitre.org/.

[4] Y. Y. a. K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," *USENIX Security Symposium,* p. 22–25, Vol. 1, 2014.

[5] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs," in *USENIX Security Symposium*, 2019.

[6] Y. Yarom, D. Genkin and N. Heninger, "CacheBleed: a timing attack on OpenSSL constant-time RSA," *Journal of Cryptographic Engineering,* vol. 7, p. 99–112, 2017.

[7] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[8] D. Gullasch, E. Bangerter and S. Krenn, "Cache games–Bringing access-based cache attacks on AES to practice," in *IEEE Symposium on Security and Privacy (SP)*, 2011.

[9] A. Purnal, F. Turan and I. Verbauwhede, "Prime+ Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[10] P. Pessl, D. Gruss, C. Maurice, M. Schwarz and S. Mangard, "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[11] O. Aciicmez and J.-P. Seifert, "Cheap hardware parallelism implies cheap security," in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2007.

[12] D. Jayasinghe, R. Ragel and D. Elkaduwe, "Constant time encryption as a countermeasure against remote cache timing attacks," in *IEEE 6th International Conference on Information and Automation for Sustainability (ICIAfS)*, 2012.

[13] J. Kong, O. Aciicmez, J.-P. Seifert and H. Zhou, "Deconstructing new cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 2nd ACM workshop on Computer security architectures*, 2008.

[14] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203,* 2018.

[15] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207,* 2018.

[16] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Von Bulck and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *Proc. ACM Conference on Computer and Communications Security (CCS)*, 2019.

[17] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," *arXiv preprint arXiv:1905.05726,* 2019.

[18] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos and C. Giuffrida, "RIDL: Rogue In-Flight Data Load," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[19] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," *arXiv preprint arXiv:1807.03757,* 2018.

[20] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison and others, "Speculative interference attacks: Breaking invisible speculation schemes," *arXiv preprint arXiv:2007.11818,* 2020.

[21] P. C. Kocher, J. Jaffe and B. Jun, "Differential Power Analysis," *Advances in Cryptology,* pp. 388-397, 1999.

[22] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu and Q. Xu, "On code execution tracking via power side-channel," *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security,* p. 1019–1031, 2016.

[23] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs and I. Verbauwhede, "Consolidating Masking," *CRYPTO,* 2015.

[24] D. Jayasinghe, A. Ignjatovic, J. A. Ambrose, R. Ragel and S. Parameswaran, "QuadSeal: Quadruple algorithmic symmetrizing countermeasure against power based side-channel attacks," *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES),* pp. 21-30, 2015.

[25] V. Arribas, S. Nikova and V. Rijmen, "VerMI: Verification Tool for Masked Implementations," *IEEE International Conference on Electronics, Circuits and Systems (ICECS),* pp. 381-384, 2018.

[26] R. Bloem, H. Gross, R. Iusupov, B. Könighofer, S. Mangard and J. Winter, "Fromal Verification of Masked Hardware Implementations in the Presence of Glitches," *Annual International Conference on the Theory and Applications of Cryptographic Techniques,* pp. 321-353, 2018.

[27] *ARM TrustZone Technology.*

[28] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović and D. Song, "Keystone: An Open Framework for Architecting Trusted Execution Environments," in *EUROSYS*, 2020.

[29] E. Singh, D. Lin, C. Barrett and S. Mitra, "Logic bug detection and localization using symbolic quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 2018.

[30] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *Design, Automation & Test in Europe Conference (DATE)*, 2018.

[31] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel and W. Kunz, "Gap-free processor verification with S²SQED and property generation," in *2020 Design, Automation Test in Europe Conference Exhibition (DATE), Grenoble, France*, 2020.

[32] G. Plassan, H.-J. P. Peter, M.-A. Katell, F. Rahhim, S. Shaker and B. Dominique, "Conclusively verifying clock-domain crossings in very large hardware designs," in *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2016.

[33] Y. Hoskote, T. Kam, P.-H. Ho and X. Zhao, "Coverage estimation for symbolic model checking," in *Proc. International Design Automation Conference (DAC)*, New York, NY, USA, 1999.

[34] R. Hojati, "Determining Verification Coverage Using Circuit Properties". Patent US Patent 6594804, 15 Juli 2003.

[35] S. Katz, O. Grumberg and D. Geist, ""Have I written enough Properties?" - A Method of Comparison between Specification and Implementation," in *Proc. Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, London, 1999.

[36] J. Bormann and H. Busch, *Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties),* 2005.

[37] K. Claessen, "A Coverage Analysis for Safety Property Lists," in *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2007.

[38] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic Verification of Finite-state Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Program. Lang. Syst.,* vol. 8, p. 244–263, April 1986.

[39] E. M. Clarke, O. Grumberg and D. A. Peled, Model Checking, London, England: MIT Press, 1999.

[40] K. L. McMillan, Symbolic Model Checking, Boston: Kluwer Academic Publishers, 1993.

[41] K. L. McMillan, "Interpolation and SAT-based Model Checking," in *Proc. International Conference on Computer Aided Verification (CAV)*, 2003.

[42] R. E. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers,* vol. 35, pp. 677-691, August 1986.

[43] E. Clarke, D. Long and K. McMillan, "Compositional Model Checking," in *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS)*, Piscataway, 1989.

[44] A. Biere, A. Cimatti, E. Clarke, O. Strichman and Y. Zhu, Bounded Model Checking, Advances In Computers Volume 58, Academic Press, 2003.

[45] C.-J. Seger and R. E.Bryant, "Formal Verification By Symbolic Evaluation And Partially-Orderd Trajectories," *Formal Methods in System Design,* vol. 6, pp. 147-189, 1999.

[46] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel and W. Kunz, "Unbounded Protocol Compliance Verification using Interval Property Checking with Invariants," *IEEE Transactions on Computer-Aided Design,* vol. 27, pp. 2068-2082, November 2008.

[47] M. Sheeran, S. Singh and G. Stalmarck, "Checking Safety Properties Using Induction And A SAT-Solver," in *Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, 2000.

[48] BMBF Collaborative Project 01M3069B, *Valse-XT,* 2003-2005.

[49] BMBF Collaborative Project 01/S07008D, *Verisoft, Verisoft-XT,* 2004 -2010.

[50] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore and F. Bruno, "Complete Formal Verification of TriCore2 and Other Processors," in *Design & Verification Conference & Exhibition (DVCon)*, 2007.

[51] K. Devarajegowda and W. Ecker, "Meta-model Based Automation of Properties for Pre-Silicon Verification," in *2018 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2018.

[52] H. Foster, *The Wilson Research Group Functional Verification Study,* 2020.

[53] "EBMC," [Online]. Available: http://www.cprover.org/ebmc/.

[54] "SymbiYosys," [Online]. Available: https://github.com/YosysHQ/sby.

[55] "HOL Interactive Theorem Prover," [Online]. Available: https://hol-theorem-prover.org/.

[56] "The Coq Proof Assistant," [Online]. Available: https://coq.inria.fr/.

[57] "Isabelle," [Online]. Available: https://isabelle.in.tum.de/.

[58] G. Gonthier, "The Four Colour Theorem: Engineering of a Formal Proof," in *Computer Mathematics, 8th Asian Symposium, ASCM, Singapore, December 15-17, 2007. Revised and Invited Papers*, 2007.

[59] D. Price, "Pentium FDIV Flaw-lessons Learned," *IEEE Micro,* vol. 15, p. 86–88, April 1995.

[60] J. Urdahl, D. Stoffel and W. Kunz, "Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs," *IEEE Transactions on Computer-Aided Design of Circuits and Systems,* vol. 33, p. 291–304, February 2014.

[61] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta and S. Malik, "Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification," *ACM Trans. Des. Autom. Electron. Syst,* vol. 24, p. 10:1–10:24, January 2019.

[62] R. Baranowski, M. A. Kochte and H.-J. Wunderlich, "Reconfigurable Scan Networks: Modeling, Verification, and Optimal Pattern Generation," *ACM Trans. Des. Autom. Electron. Syst.,* vol. 20, pp. 30:1-30:27, March 2015.

[63] K. v. Gleissenthall, R. G. Kıcı, D. Stefan and R. Jhala, "IODINE: Verifying Constant-Time Execution of Hardware," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[64] G. Cabodi, P. Camurati, F. Finocchiaro and D. Vendraminetto, "Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification," in *Intl. Conf. on Codes, Cryptology, & Information Security*, 2019.

[65] G. Cabodi, P. Camurati, S. F. Finocchiaro, F. Savarese and D. Vendraminetto, "Embedded Systems Secure Path Verification at the HW/SW Interface," *IEEE Design & Test,* vol. 34, p. 38–46, 2017.

[66] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," in *Design and Test In Europe (DATE)*, 2014.

[67] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security,* vol. 18, p. 1157–1210, 2010.

[68] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra and W. Kunz, "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking," in *IEEE Design Automation and Test in Europe (DATE)*, 2019.

[69] M. R. Fadiheh, A. Wezel, J. Mueller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel and W. Kunz, "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," *in preview of IEEE Transactions on Computers,* 2022.

[70] J. Müller, M. R. Fadiheh, A. L. Duque Anton, T. Eisenbarth, D. Stoffel and W. Kunz, "A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level," in *IEEE/ACM Design Automation Conference (DAC)*, 2021.

[71] M. Goli and R. Drechsler, "Early SoCs Information Flow Policies Validation Using SystemC-Based Virtual Prototypes at the ESL," *ACM Transactions on Embedded Computing Systems, https://doi.org/10.1145/3544780,* 2022.

[72] R. Guanciale, M. Balliu and M. Dam, "InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[73] S. A. Seshia and P. Subramanyan, "UCLID5: Integrating modeling, verification, synthesis and learning," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2018.

[74] C. Trippel, D. Lustig and M. Martonosi, "CheckMate: Automated synthesis of hardware exploits and security litmus tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[75] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *ACM SIGPLAN Notices*, 2011.

[76] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," *ACM SIGARCH Computer Architecture News,* vol. 42, p. 97–112, 2014.

[77] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware IP," in *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*, 2015.

[78] Y. Bertot and P. Castéran, Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions, Springer Science & Business Media, 2013.

[79] D. Zhang, Y. Wang, G. E. Suh and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM SIGPLAN Notices,* vol. 50, p. 503–516, 2015.

[80] A. Ferraiuolo, M. Zhao, A. C. Myers and G. E. Suh, "HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security," in *ACM SIGSAC Conf. on Computer & Communications Security*, 2018.

[81] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," *IEEE/ACM Design Automation Conference (DAC),* p. 1212–1221, 2012.

[82] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel and W. Kunz, "Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing," in *IEEE/ACM Design Automation Conference (DAC)*, 2022.

[83] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *arXiv preprint arXiv:1902.05178,* 2019.

[84] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[85] G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra and A. Roychoudhury, "KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution," *arXiv preprint arXiv:1909.00647,* 2019.

[86] S. Guo, Y. Chen, P. Li, Y. Cheng, H. Wang, M. Wu and Z. Zuo, "SPECUSYM: Speculative symbolic execution for cache timing leak detection," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[87] K. Cheang, C. Rasmussen, S. Seshia and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019.

[88] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke and A. Sánchez, "SPECTECTOR: Principled detection of speculative information flows," *arXiv preprint arXiv:1812.08639,* 2018.

[89] R. J. Colvin and K. Winter, "An abstract semantics of speculative execution for reasoning about security vulnerabilities," in *International Symposium on Formal Methods*, 2019.

[90] M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kıcı, R. Jhala, D. Tullsen and D. Stefan, "Automatically eliminating speculative leaks from cryptographic code with blade," in *Proceedings of the ACM on Programming Languages*, 2021.

[91] M. Guarnieri, B. Köpf, J. Reineke and P. Vila, "Hardware-Software Contracts for Secure Speculation," *arXiv preprint arXiv:2006.03841,* 2020.

[92] RISC-V Foundation, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version1.10,* A. Waterman and K. Asanović, Eds., 2017.

[93] "The seL4® Microkernel," [Online]. Available: https://sel4.systems/.

[94] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas and C. W. Fletcher, "Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[95] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy and B. Kasikci, "DOLMA: Securing Speculation with the Principle of Transient Non-Observability," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.


cyberagentur