Using Universal Composition to Design and Analyze Secure Complex Hardware Systems

Ran Canetti[††], Marten van Dijk [*], Hoda Maleki[†], Ulrich Rührmair[§], Patrick Schaumont[♯]

***Abstract*—Modern hardware typically is characterized by a multitude of interacting physical components and software mechanisms. To address this complexity, security analysis should be modular: We would like to formulate and prove security properties of individual components, and then deduce the security of the overall design (encompassing hardware *and* software) from the security of the components. While this seems like an elusive goal, we argue that this is essentially the only feasible way to provide rigorous security analysis of modern hardware.**

This paper investigates the possibility of using the Universally Composable (UC) security framework towards this aim. The UC framework has been devised and successfully used in the theoretical cryptography community to study and formally prove security of arbitrarily interleaving cryptographic protocols. In particular, a sophisticated analytical toolbox has been developed using this framework. We provide an introduction to this framework, and investigate, via a number of examples, ways by which this framework can be used to facilitate a novel type of modular security analysis. This analysis applies to combined hardware and software systems, and investigates their security against attacks that combine both physical and digital steps.

***Index Terms*—Universal Composition Framework, Hardware Security, Physical Cryptography and Security**

I. Introduction

A. Overview and Motivation

Due to miniaturization, multiple functionalities, concurrently running apps, high connectivity, and yet other aspects, modern computers and hardware systems have turned into outstandingly intricate objects. As Google's Ulfar Erlingsson once put it [1], hardware systems *"have become these incredibly complex spaceships that have landed in our backyards, and we use them to make coffee."* Besides the well-known underload in average scenarios the quote alludes to, it also points to some of the resulting, severe security and privacy issues: *(i)* How can users make sure that the "spaceships" in their backyards only have the desired functionalities (and no others)? *(ii)* How can system designers ensure that the complex interplay of software and hardware, and of high-level and low-level components, still creates an overall secure system? Both aspects represent highly non-trivial challenges for the security community.

This paper is concerned mostly with question *(ii)*. We observe that the situation of complex modern hardware is partly similar to the intricate composition of concurrently running, multiple protocol instances. The so-called *"Universal Composition" (UC)* framework [2] has been introduced almost two decades ago in the theoretical cryptography community in order to deal with exactly such situations, and currently probably represents the most advanced framework for their analysis. Its sophisticated tool box allows, for example, proving security properties of arbitrary parallel protocol compositions within certain, well-defined circumstances — or also disproving the existence of schemes that compose securely in certain situations.

The main novelty of our work now lies in suggesting that this existing UC machinery may be beneficially adapted and translated to a hardware context, too. This relates both to using the tools and theorems of the UC framework for novel formal proofs in a hardware setting (a future research avenue mainly for theoreticians), as well as to applying the general thinking behind the UC model in the practical construction of secure, multi-layered hardware (a task relevant for practitioners, engineers, and hardware designers). Furthermore, thinking about security analysis already at the stage of designing basic components might, we hope, lead to further collaboration between the communities and result both in more secure hardware and in more practically relevant and fruitful research.

At the same time, we would like to stress that this work is just an expository first step, and does not include full-fledged theorems or comprehensive adaptations of the UC framework for general secure hardware design. This must remain subject to future activities. Instead, this paper aims to make the hardware community aware of the large potential that lies in the intersection of hardware security and the UC formalism. To this end, we provide a number of concrete examples that hopefully guide readers at least through the outskirts of the UC jungle. We also attempt to communicate some of the general UC-thinking to practitioners and engineers, trying to make them more sensitive to security concerns and pitfalls that arise whenever even seemingly simple software and hardware primitives are combined across various system layers. By doing so, we hope to prepare the grounds for a new and potentially fruitful research area in the interplay of hardware security and theoretical cryptography.

B. Related Work

There are three existing research strands in the wider intersection between the UC framework and hardware security that require mentioning. The first one deals with the use of (assumedly) tamper-proof hardware tokens as a tool in cryptographic protocols, and shows how the physical exchange of such tokens allows secure communication schemes, for

[††]Boston University, canetti@bu.edu. Support by the NSF MACS Project.
[*]CWI Amsterdam and UConn. Supported by the NSF MACS Project.
[†]Augusta University, hmaleki@augusta.edu
[§]LMU Munich and University of Connecticut, ruehrmair@ilo.de
[♯]WPI, pschaumont@wpi.edu. Supported by NSF Award 1617203.

978-3-9819263-4-7/DATE20/©2020 EDAA

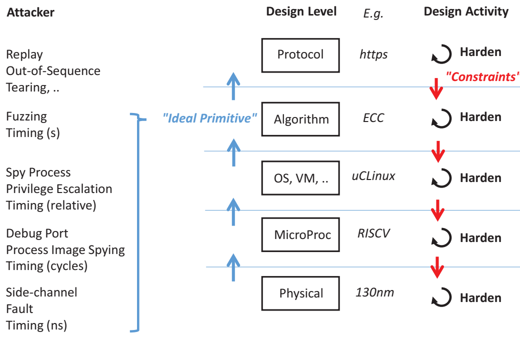

Fig. 1. Layers of adversarial environments for computing systems.

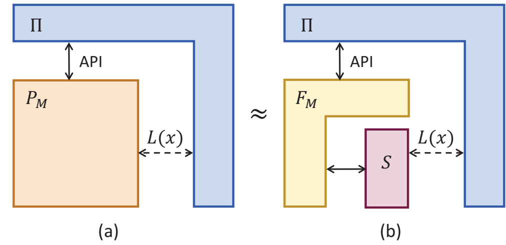

Fig. 2. Universally Composable framework. The module communicates with the outside world through its defined API. Dashed arrows indicate the information leakage to the adversary. a) illustrates the physical realization of module M, while b) shows the ideal functionality of module M. The simulator communicates with F_M in order to simulate the leakage function $L(x)$.

example secure multiparty computation [3], [4]. The second strands treats a physical cryptographic primitive, namely so-called physical unclonable functions (PUFs), in the UC framework [5]. As in the first strand, PUFs are used as a protocol tool in this line of work, which must be exchanged physically between different parties. Both strands strictly differ from our efforts, however, as they do not aim to prove statements about secure hardware, particularly not about the secure composition of hardware and/or software components in complex devices, as we do. The third strand, which is closer in spirit to the goals of this work, considers UC security of multi-party computation protocols in a setting where the adversary may, in addition to its usual capabilities of controlling the communication and some of the parties, also obtain arbitrary "leakage" information on the internal states of parties (see [6] and a number of follow up works). However, both the formalism and the analysis provided there are more high-level and are not sufficiently fine-grained to address realistic hardware design issues. Indeed, to the best of our knowledge, our work thus is the first that suggests to apply the UC framework to analyze and prove the security of complex, multi-layered hardware systems.

II. Applying the UC Paradigm to Hardware Systems

Figure 1 describes the common design abstractions in modern electronic design covering the (micro) transistor level up to the (macro) application level. Designers express functionality and structure of a design in terms of abstractions common at each level. Abstraction is essential to deal with complexity, and it plays a fundamental role in verifying the correctness of the design. However, these abstractions do not apply to the attacker, as the following example illustrates.

Consider an attack on a secure web server by an attacker who is physically remote. The attacker may use a combination of attack vectors that target different components within the web server: The attacker may use replay or out-of sequence messages at the protocol level; it can use malformed requests at the algorithm level; it can use timing effects at the processor/instruction level; it can also use any combination of the above. In other words, attacks are not restricted by the abstraction levels used within the design. This leads to the following design challenge for secure design. How can a designer rely on the design abstractions that are so crucial for correct and efficient design, while at the same time taking into account that an attacker can exploit any aspect of the implementation? This is, in essence, what secure composition of hardware and software aims to achieve. Figure 1 describes an intuitive solution. The design is decomposed into 'ideal primitives', which guarantee a perfect implementation not only from the functional point of view, but also from the security point of view. A designer builds a secure design using such ideal primitives.

While we do not know, today, how to solve this design challenge in the general sense, designers have proposed a myriad of techniques that achieve 'ideal primitives' hardened against specific attackers. First, software and hardware can be hardened using common countermeasures. Second, software and hardware designs can introduce appropriate constraints towards the lower layers to guarantee that countermeasures remain correct as the design is refined into implementation.

The objective of our work is to investigate how this challenge can be addressed in a formal manner. From the Universal Composability framework [2] we learn that we can specify the expected behavior of a physical realization P_M of a module M via an ideal functionality F_M. The main idea is to have F_M precisely describe how we would like a realization of the module to react to each and every external impetus. The security definition will essentially then say that P_M is "at least as secure as" the ideal functionality F_M, in the sense that no "external environment," which is represented as a single centralized, adversarial algorithm that interacts with a system, can tell whether it is interacting with the real system or the ideal system.

An ideal functionality can impose both functionality and secrecy requirements: Having F_M, on input a function f, responds with $f(x)$ for some stored value x, means that the implementation must also output $f(x)$; Furthermore, it means that no information other than $f(x)$ should be leaked.

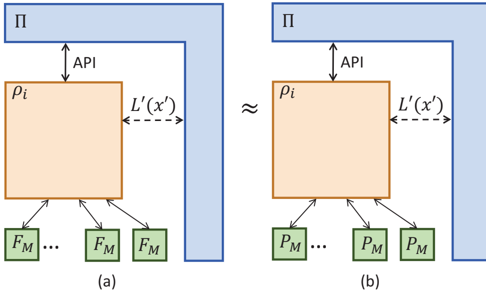

Fig. 3. a) Each functionality in layer i may interact with multiple instances of ideal functionality F_M. The UC composition theorem indicates that replacing the ideal functionalities F_M of module M with its physical realization P_M preserves the promised security properties. In other words, (a) and (b) are computationally indistinguishable.

For example, if module M maintains a secret key, then it suffices to write F_M so that it will simply not leak sufficient information to the external environment.[1] This will mean that any physical realization must not leak the key either (or else the adversary will be able to distinguish the realization from the ideal behavior).

More precisely, in order to prove that the physical realization P_M is as secure as the ideal functionality F_M, we first model all the side information given to the adversary Π by a constraint on the side information (see Figure 2). This is represented by a possible set of leakage functions $\mathcal{L}$ from which adversary Π selects one specific leakage function $L(x) \in \mathcal{L}$ (say, all functions that leak at most B bits on the internal state). Here, input x represents all the information/state of P_M and $L(x)$ filters this information/state. In order for the designer of the larger system to prove that P_M realizes the ideal functionality F_M, the designer needs to construct a simulator S which sits in between F_M and adversary Π such that S simulates $L(x)$ as seen from the real physical system P_M with only the limited information provided by the ideal functionality F_M. The goal is to show that P_M is computationally indistinguishable from $F_M \circ S$.

The elegance of the UC framework comes from its composition theorem which is illustrated in Figure 3. Applied to our setting, the theorem states that if a system ρ_i at layer L_i of an electronic system uses a module M with ideal functionality F_M, then we may replace each use of F_M by its physical realization P_M, without changing the overall security of the overall system. In particular, ρ_i with plugins to F_M is computationally indistinguishable from ρ_i with plugins to P_M. This theorem allows us to reason about security in a modular fashion.

Above we explained how modules compose in the UC framework, but we need to be careful. We stress that the process of abstraction in the UC modeling does not restrict the adversary: Indeed, the adversary remains the same throughout. Instead, the thinking is to replace more and more components of the proposed solution with their ideal counterparts. In particular, in UC modeling the adversary remains the same throughout the process of the successive abstractions in Figure 1. What changes is the replacement of lower-level components with idealized ones. For example, if for one module M we prove security with respect to a class of leakage functions $\mathcal{L}$ (this describes the strength of the adversary Π) and if for another module M' we prove security with respect to a class of leakage functions $\mathcal{L}'$, then M and M' with their interactions may not necessarily be secure against the combined/composed adversary described by $\mathcal{L}$ with $\mathcal{L}'$: Due to the interactions/protocol between M and M', the leakage from M together with leakage from M' may break security because M and M' may share state (e.g., a key is shared and the different leakage functions together simply leak too much for the composition of M and M' to be secure). This argument also hold for composing layers – again leakage at the transistor level can possibly be combined with leakage from digital information from a software level to break security.

Notice, however, that the composition theorem makes the composition of leakage sources more manageable, and in fact modular: Given that $\exists_S\ P_M \approx F_M \circ S$, and $\exists_{S'}\ P_{M'} \approx F_{M'} \circ S'$, the composition theorem tells us that there exists a simulator S^ρ for the ideal functionality ρ which combines F_M and $F_{M'}$ such that the physical realization P_ρ of ρ realizes ρ with calls to F_M and $F_{M'}$ substituted by P_M and $P_{M'}$. This latter proof shows that even an adversary with access to both $\mathcal{L}$ and $\mathcal{L}'$ cannot break security. For example, assume the adversary is strong enough to do a power side channel analysis and can therefore use this to learn a leakage function for module M as well as a leakage function for module M'. In this sense $\mathcal{L}$ and $\mathcal{L}'$ are similar in nature but different with respect to their input space (as these describe the different states of the different modules). As another example, at the processor architecture level we wish to provide 'hardware isolation' as a security guarantee – here, the interactions between the CPU (i.e., module M) and DRAM (i.e., module M') are implemented as a hardware access control system which isolates untrusted computing (by e.g. the higher layer OS which uses the processor architecture layer) away from accessing a secure memory compartment in DRAM.

Similarly, in order to show that a collection of counter-measures "compose securely" we would need to show how these counter-measures can be "sequentialized" so that each counter-measure can be treated as a layer that builds on the guarantees of the layers below, and provides an additional set of guarantees (or, abstractions) that can be used by the higher layers. There is a caveat when composing counter measures as each counter measure CM is specific to a certain class of leakage functions $\mathcal{L}_{CM}$ and counter measures are integrated as they are not implemented as separate 'modules'. The latter does not allow us to apply the composition theorem. Instead, one needs to argue the security of a physical realization of

[1]The UC framework posits an adversarial execution environment and models the environment by way of a single computational entity, named *environment*. To avoid confusion, in this paper we will often use the term adversary instead of environment.

multiple counter measures in one universal approach. And in such a universal approach we need to model adversary Π as an adversary who can learn all leakage functions L_{CM} together at the same time. Indeed, applying protection mechanisms together without taking care of potential circularity, may lead to breaking security: For example, we may add redundancy to thwart a fault injection attack and this is in conflict with the masking added to thwart a power side channel attack. As a hardware security engineer we must first define the adversary with its leakage function from which we wish to protect ourselves and next find a holistic counter measure approach in order to reach this goal of being secure against such an adversary. The decision of what kind of (strong) adversary we need to take into account is often given to us after doing a (business) risk analysis with respect to security breaches.

III. The Environment, Also Known As: The Adversary

When reasoning from the hardware perspective of an electronic system, we need to understand how the interaction between the environment and functionality (as depicted in Figure 2) represents all the ways in which the physical, actual external environment interacts with the system. In fact UC modeling invites us to formalize in a mathematical language the capabilities of the actual physical adversarial environment and its interactions with the system.

Recall that the UC modeling separates between the input and output interfaces, which represent the desired input/output behavior of the system, and "side channel", or "adversary interfaces" which allow the environment to gain information on the internal state of the system, and also influence the behavior of the system. Specifically, the adversary interface allows the environment to obtain the result of a *leakage function* L, applied to the internal state of the system. This function can represent physical observations such as a heat map, or other observable for which the adversary has designed a suitable monitoring tool.

This modeling allows the adversary to be the same across all layers; the different capabilities or monitoring tools per layer are captured by the code of the respective ideal functionalities and systems. In other words the interaction with the environment in the UC model consists of several layers ranging from leakage coming from the micro (transistor) level to leakage from higher levels (for example, cache side channel).

The above discussion has so far missed an important point: Indeed, the UC model makes explicit in a leakage function $L(x)$ what kind of additional information can be observed by an adversary. This describes a passive adversary. Also possible is to have an active adversary who is able to *extend* the API by being able to *modify* the physical system other than what the digital API describes. For example, the CPU clock can be made to run faster and this can lead to power glitches which will change the state of the functionality. Now the abstract notation of a UC model falls short in that the analyst needs to think ahead and formalize both an extended API (which properly reflects the adversary) as well as how the functionality behaves given the extra inputs over the API extension. Thus capabilities comes in two forms:

- Communication from Functionality to Adversary: The adversary has a fine tuned tool which can be used to monitor interactions and extract information from interactions.
- Communication from Adversary to Functionality: The adversary has a means to influence the functionality beyond what the digital API specifies.

Example – Cold-Boot Attack: The Cold-Boot Attack illustrates how difficult it can be to deal with a multi-layered adversary. This attack is used to break trusted-boot configuration based on disk encryption or trusted platform modules [7]. Volatile memory-modules containing sensitive content and encryption keys are the final outcome of a controlled trusted-boot operation. When a computer's power is removed, the volatile sensitive content disappears, leaving only the hardened encrypted disk and trusted-platform module to defend. However, by cooling the memory chips, the data remnence can be significantly extended, to the point where the chips can be manually removed and inserted into a different computer under control of the attacker. This completely breaks the assumption of the disk-encryption mechanism. A logical mechanism that prevents leakage $L(x)$ of an in-memory secret x is disabled by building a new physical pathway that enables that leakage. In other words, the Cold-Boot Attack initiates a new type of communication; a pathway at a low physical abstraction based on the physical environment of the trusted computing system. Such attacks and countermeasures can be captured in UC analysis by incorporating the adversary's ability to modify the physical behavior of memory under temperature change within the standard model of computation (that usually ignores physical aspects of computations.)

Example – Speculative Execution Attacks: In recent years, we have witnessed a broad range of attacks that exploit micro-architecture features in high-performance processors as leakage channels [8]. When processors execute instructions speculatively, the micro-architecture state is temporarily different from the logical state of the program. The difference is exploited as a leakage function, such as by observing execution time of branch instructions and memory accesses. The observations, in turn, are available from standard built-in performance monitoring infrastructure. Micro-architecture speculation, and performance monitoring are both instrumental at building fast computers and high-performance software. The combination of both, on the other hand, has enabled a leakage model $L(x)$ that has affected every major processor architecture in the past two years. Again, such attacks and countermeasures can be captured by incorporating within the model of computation the ability of the adversary to measure the duration of certain short execution snippets.

One main principle arises from these examples: *it is dangerous to reason about security in too abstract high levels of environment-functionality interaction*.

IV. (Ideal) Functionalities versus (Real) Physical Processes

Strictly speaking, real electronic systems do not solely implement the abstract Boolean functions they are designed to compute: Only at the higher layer, when looking only at digital information, we may see an input and output behavior that can be modeled as a (deterministic) function. At the lower layer, physical noise in the form of jitter, glitches, etc. transforms the input-output behavior into a physical statistical process.

The physical realization of a Boolean functionality $f(.)$ hence does not only implement $f(.)$; instead, it implements an analog function that contains enough information to deduce $f(.)$ — and perhaps much more. To illustrate this, assume for simplicity that the physical realization can be modeled as taking x as input, computing $y = f(x)$, and adding noise by computing $n_p(y)$, where p is a statistical parameter (e.g., with bias p a bit in y is flipped). If p itself is a function of x, then the physical realization outputs

$$z = n_{p(x)}(y) \text{ with } y = f(x).$$

Here, y may be a binary string while z is an element of a much richer structure, e.g., as a vector of real values. Clearly, z embeds y but adds additional information about x as a result of $n_{p(x)}(.)$. An adversary, who observes z, can exploit this to extract information about x which the ideal functionality $f(.)$ does not want to reveal. When we talk about an ideal random process (rather than ideal functionality) we want p to be independent of x such that $x \rightarrow y \rightarrow z$ represents a Markov model and z does not leak more information about x than y leaks information about x. The UC framework allows capturing and representing this additional information by way of formulating the appropriate internal states for the relevant physical modules, and then letting the Boolean outputs be one derivative of the analog internal state, and the "leakage function" be another derivative of the same internal state.

Example – Reliability based PUF Attack: In this example the functionality represents a PUF that takes a binary string, called challenge, as input and generates a binary bit, called response, as output. In theory and at first glance, this purely digital and deterministic model of a PUF is fine. However, in practice, the inevitable measurement noise transforms this input-output behavior into a statistical process; the PUF in reality does *not* represent a deterministic function. This fact initially sounds like a merely practical nuisance that can be overcome by suitable error correction. However, as some recent attacks have shown [9], it is actually more than that, and is highly security relevant: By repeating the measurement of a response to the same challenge for several times, the adversary can obtain "reliability" information, in the form of an estimate of the probability the response bit is equal to 0. This is a finer grained type of information about how the PUF computes its response bit, and can boost the machine learning performance from exponential to polynomial in certain PUF architectures, for example in the XOR Arbiter PUF (compare [9], [10], [11], [12]). The overall reason that enables these novel polynomial attacks, and that initially perhaps has lead to the incomplete belief that there are no such attacks, is the inaccurate and incomplete modeling of a PUF as a deterministic mathematical function on the logical level. Once more, we stress that the above attacks obviously need to be distinguished from a PUF-attack on a pure protocol level (see, e.g., [13], [14]), or from a purely physical PUF-attack (such as [15], for example). Instead, it lives right at the boundary between physics and mathematics, so to speak. Again, this shows the relevance of accurate modeling physical security primitives across different layers of abstraction.

In truth 'noise' is more complex in that it is interwoven with functionality f (represented as an interactive protocol or algorithm) in a more complex way. Another principle is that *one should model a functionality as a random/statistical interactive process.*

V. Transferring UC Composition and Proofs to a Hardware Context

The UC composition theorem makes the following statement: Assume we have two functionalities F_A and F_B, a protocol P_A that realizes F_A with the assistance of making calls to F_B, and another protocol P_B that realizes F_B. Stated more compactly we have that $P_A^{F_B}$ realizes P_A, and P_B realizes F_B. Then, the composition theorem says that $P_A^{P_B}$ realizes F_A. More specifically, assume there is a simulator S_A such that $P_A^{F_B} \approx F_A \circ S_A$, and a simulator S_B such that $P_B \approx F_B \circ S_B$. Then there is a composite simulator S_{AB} such that $P_A^{P_B} \approx F_A \circ S_{AB}$.

This is a powerful statement which can help the security analyst in the following ways. For example, a module with counter measure P_B needs to assume that it has access to an ideal functionality F_A which offers a certain property which the counter measure is able to amplify in making the whole module P_B being able to resist the (modeled) adversary. In a way the root of trust is reduced to only F_A, i.e., there is a secure physical realization P_A for the smaller module with ideal functionality F_A.

Example – Masking as a Side-Channel Countermeasure: In order to resist an adversary with access to the power side channel, circuitry (P_B) can be implemented using masking or secret sharing. A sensitive signal a is split into secret shares $\{r_0, r_1, .., r_k, a \oplus r_0 \oplus r_1 \oplus ... \oplus r_k\}$ using k random numbers r_i. As each share is statistically independent from the others, the side-channel analysis becomes significantly more difficult for the adversary. Regardless of the complexity of such a higher-order side-channel attack, we observe that secret sharing requires multiple independent and identically distributed (iid) random bits for each masked bit. In practice, random bits are generated by a combination of a True Random Number Generator (TRNG) and a deterministic Pseudorandom Generator (PRNG) [16]. The latter of these two, the PRNG, requires protection from side-channel analysis since disclosure of its internal state makes the shares predictable. Leakage-resilient design techniques are applicable to the design of such

a PRNG [17], [18]. While leakage-resilient design is hard to achieve in general algorithms, those general algorithms can be masked using secret-sharing, with the shares created by a leakage-resilient RNG. This illustrates how the security of a masked algorithm F_B can be built using leakage-resilient random-number generation F_A.

Another interpretation of the same 'AB' argument is that an adversary can have several leakage function that are specific to different abstraction layers. The higher layer description of a module implementation P_B uses lower layer machinery – in a sense P_B calls lower layer implementations P_A. The adversary observes the lower layer P_A differently than the higher layer P_B. Thus, we only need to prove the security of P_A with respect to the leakage function that corresponds to P_A's level and we need to prove the security of P_B with respect to the leakage function that corresponds to P_B's level. By the UC theorem this is sufficient to conclude that $P_B^{P_A}$ is secure with respect to an adversary with access to both leakage functions.

VI. Conclusions & Future Directions

In sum, the UC framework may well enable modular design and analysis of hardware and software components. However, care must be taken in order to compose components in "the right way" so that the composition theorem can be applied, Indeed, arbitrarily combining counter-measures might not lead, in general, to an overall mechanism that provides the combination of the individual protections.

A natural way to combine protection mechanisms in a security-preserving manner is to apply these mechanisms in a layered fashion. That is, logically order the protection mechanisms, and then show that (a) The first mechanism realizes an ideal functionality F_1 that provides some initial protection guarantees, and (b) each subsequent mechanism M_i, using the protections provided by ideal functionality F_i, realizes an ideal functionality F_{i+1}, where F_{i+1} provides stronger guarantees than F_i. Indeed, thinking about security along the lines of the UC framework *does* seem to allow us understanding the subtleties of the effect of module and layer interactions, and the development of effective countermeasures, better.

A caveat of UC analysis for large complex electronic systems is that comprehensive analysis appears to be prohibitively complicated. Several approaches for mitigating this complexity appear possible in practice. On approach is to analyze only relatively small components with limited functionality, and then building one's way slowly to deal with more complex components. This "bottom-up" approach might, for instance, start with modeling of individual transistors or other components, and build up to larger circuits. Alternatively, one could take a "top-down" approach which starts by capturing only higher-level components and abstractions, and then works its way down to further refine component structure. Either way, developing (or adapting) techniques from formal verification is bound to be extremely useful. Ideally, one would wish to reason about security in a Higher Level Coding (HLC) language, and to have compilers that implement countermeasures in secure ways. This means that the complexity of analysis is pushed towards proving that the compilers properly translate functionality (correctness), as well as introduce countermeasures in order to guarantee security (resilience and safety). Of course, designers will still need to explain through annotation in their HLC code what is expected from the compiler. As another example, one may construct a library of gate implementations at the transistor level together with realistic leakage functions. A compiler which translates from circuit level to transistor level can use this library to provide security guarantees. In this example, the difficulty lies in how to obtain realistic leakage functions.

It is clear that reasoning about security of electronic hardware systems is difficult, and will at least in parts always remain so. Nevertheless, there do seem clear paths ahead of us in terms of further applying UC analysis, and of designing appropriate HLC and compilers. These steps might help substantially improving the security of future generations of complex, multi-layered electronic devices.

Acknowledgements

This note was inspired by a Dagstuhl workshop on Secure Composition for Hardware Systems, organized by Divya Arora, Ilia Polian, Francesco Regazzoni and Patrick Schaumont. We thank them and all the workshop participants.

References

[1] U. Erlingsson, Personal communication at ASHES 2017.

[2] R. Canetti, "Universally composable security: A new paradigm for cryptographic protocols," in *FOCS 2001*.

[3] J. Katz, "Universally composable multi-party computation using tamper-proof hardware," in *CRYPTO 2007*.

[4] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tamper-proof hardware tokens," in *TCC 2010*.

[5] C. Brzuska, M. Fischlin, H. Schröder, and S. Katzenbeisser, "Physically uncloneable functions in the universal composition framework," in *CRYPTO 2011*.

[6] N. Bitansky, R. Canetti, and S. Halevi, "Leakage-tolerant interactive protocols," in *TCC 2012*.

[7] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX 2008*.

[8] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *USENIX 2019*.

[9] G. T. Becker, "The gap between promise and reality: On the insecurity of xor arbiter pufs," in *CHES 2015*.

[10] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *DAC 2007*.

[11] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber, "Modeling attacks on physical unclonable functions," in *CCS 2010*.

[12] U. Rührmair and J. Sölter, "Puf modeling attacks: An introduction and overview," in *Conference on Design, Automation & Test in Europe, 2014*. European Design and Automation Association, 2014, p. 348.

[13] U. Rührmair and M. van Dijk, "Practical security analysis of puf-based two-player protocols," in *CHES 2012*.

[14] ——, "On the practical use of physical unclonable functions in oblivious transfer and bit commitment protocols," *JCEN 2013*.

[15] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert, "Cloning physically unclonable functions," in *HOST 2013*.

[16] V. Fischer, "Random number generators for cryptography, design and evaluation," URL: https://summerschool-croatia.cs.ru.nl/2014, 6 2014.

[17] Y. Yu, F. Standaert, O. Pereira, and M. Yung, "Practical leakage-resilient pseudorandom generators," in *CCS 2010*.

[18] M. M. I. Taha, A. Reyhani-Masoleh, and P. Schaumont, "Stateless leakage resiliency from nlfsrs," in *HOST 2017*.