How Not to Protect Your IP – An Industry-Wide Break of IEEE 1735 Implementations

Julian Speith*†, Florian Schweins†, Maik Ender*†, Marc Fyrbiak*, Alexander May†, Christof Paar*†
*Max Planck Institute for Security and Privacy, Bochum, Germany,
julian.speith@mpi-sp.org, maik.ender@mpi-sp.org, marc.fyrbiak@mpi-sp.org, christof.paar@mpi-sp.org
†Horst Görtz Institute for IT Security, Ruhr University Bochum, Bochum, Germany,
florian.schweins@rub.de, alex.may@rub.de

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833605

***Abstract*—Modern hardware systems are composed of a variety of third-party Intellectual Property (IP) cores to implement their overall functionality. Since hardware design is a globalized process involving various (untrusted) stakeholders, a secure management of the valuable IP between authors and users is inevitable to protect them from unauthorized access and modification. To this end, the widely adopted IEEE standard 1735-2014 was created to ensure confidentiality and integrity.**

In this paper, we outline structural weaknesses in IEEE 1735 that cannot be fixed with cryptographic solutions (given the contemporary hardware design process) and thus render the standard inherently insecure. We practically demonstrate the weaknesses by recovering the private keys of IEEE 1735 implementations from major Electronic Design Automation (EDA) tool vendors, namely Intel, Xilinx, Cadence, Siemens, Microsemi, and Lattice, while results on a seventh case study are withheld. As a consequence, we can decrypt, modify, and re-encrypt all allegedly protected IP cores designed for the respective tools, thus leading to an industry-wide break. As part of this analysis, we are the first to publicly disclose three RSA-based white-box schemes that are used in real-world products and present cryptanalytical attacks for all of them, finally resulting in key recovery.

***Index Terms*—Hardware IP Protection, IEEE Standard 1735-2014, Reverse Engineering, Key Extraction, White-Box RSA**

I. Introduction

The emerging digital society is based on a myriad of interconnected computers and embedded devices built from Integrated Circuits (ICs). Modern ICs are increasingly realized as Systems-on-Chip (SoCs), i.e., a single chip that incorporates a large number of different functional modules, referred to as IP cores. In particular, modern SoCs can consist of hundreds of IP cores that implement a wide range of functionalities, from simple periphery, to cryptographic co-processors, neural network accelerators, or even entire CPUs. Given both the increasing complexity of modern hardware and strict time-to-market requirements [6], hardware designers resort to the re-use of well-proven IP cores, often from third-party providers. However, the use of third-party IP cores provides several security challenges from both the IP author as well as the IP user perspective [6] due to the globalized hardware design and manufacturing process comprising numerous untrusted stakeholders. For example, the author wants to safeguard their valuable IP so that no other party can infringe their IP or its license. Simultaneously, the user wants to assure that the purchased IP is not modified by another stakeholder.

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany´s Excellence Strategy - EXC 2092 CASA - 390781972 and ERC grant 695022.

To address the secure management of hardware IP, IEEE standard 1735-2014 [31] was established to provide aforementioned confidentiality and integrity while ensuring interoperability between EDA tools. The standard is supported by all major tool vendors and is trusted throughout the industry [26].

At CCS'17, Chhotaray et al. [17] presented an in-depth analysis of IEEE 1735. The authors identified exploitable weaknesses with a focus on cryptographic flaws (e.g., a padding-oracle attack on non-authenticated symmetric encryption). They recommend using secure cryptographic algorithms, i.e., authenticated encryption, to protect against these attacks:

> "*From a cryptographic perspective, the solution is simple. Use a provably secure authenticated encryption scheme that supports associated data (AEAD) to encrypt the sensitive IP [...].*" [17]

Our work proves this statement to be insufficient in practice. We go considerably further and question the standard's general security (even assuming strong cryptographic primitives) due to its foundational assumptions about the trust model and execution environment:

> "*A decryption tool shall manage its secret or secrets in a private, secure manner. It may harden such secrets directly into its software or use an external persistent storage scheme.*" [31, p. 20]

Here, the crucial decryption module is embedded within an EDA tool that is typically executed in an untrusted, adversary-controlled environment. This raises the question of how such a module can be protected so that an adversary cannot exploit the decryption software and disclose valuable IP.

Goals and Contributions. In this paper, we analyze the security of the IEEE standard 1735-2014 and its implementations. Our goal is to assess the standard's trust model assumption with respect to the capabilities of real-world adversaries. To this end, we review the standard's recommendations on both the handling of cryptographic key material and the toolchain execution environment, with a particular focus on adversaries with static and dynamic software reverse-engineering capabilities. In the case studies listed in Table I, we investigate how the standard is realized in market-leading

TABLE I
OVERVIEW OF OUR CASE STUDIES ON IEEE 1735 IMPLEMENTATIONS OF MAJOR EDA TOOLS. FOR EACH TOOL, WE LIST WHETHER WE DEFEATED THE ENCOUNTERED SOFTWARE PROTECTIONS (✓) OR NO SUCH PROTECTIONS WERE PRESENT AT THE TIME OF ANALYSIS (-). ADDITIONALLY, WE PROVIDE THE NUMBER OF RECOVERED PRIVATE KEYS AND THE APPROXIMATED TIME REQUIRED TO PERFORM EACH CASE STUDY.

EDA Tool	**Defeated Software Protections**			**#Recovered**	**Time**
	Obfuscation	Anti-Debugging	White-Box	**Private Keys**	**Estimate**
Intel Quartus Prime	-	-	-	1	1 hour
Cadence Xcelium	-	-	-	2	1 hour
Lattice Radiant	-	-	-	1	3 hours
Microsemi Libero SoC	✓	✓	(✓)	1	1 week
Siemens ModelSim	-	-	✓	3	3 days
Xilinx Vivado Design Suite	✓	✓	✓	5	2 weeks

Field-Programmable Gate Array (FPGA) and Application-Specific Integrated Circuit (ASIC) design and verification tools. Based on software reverse engineering, we reveal *all* cryptographic private keys of the analyzed EDA tools within weeks, days, or sometimes even just hours. In particular, we bypass a wide variety of software protection measures, including three different white-box RSA implementations. Thereby, we provide the first-ever description of public-key white-box schemes used in real-world applications. We then analyze these schemes and provide multiple attacks, each of which is able to invalidate the standard's security properties of IP confidentiality and integrity. Our main contributions are:

- **Insecurity of IEEE 1735.** We describe flaws in the current IEEE standard 1735-2014 for hardware IP protection. Our investigation focuses on the foundational trust model assumptions, i.e., the *untrusted execution environment*, the recommendation of *hard-coded keys*, and the *lack of guidance* on a secure implementation (see Section III). We argue that these trust model assumptions cannot be addressed with cryptographic solutions given the current hardware design process and thus render the standard inherently insecure.
- **Case Studies on Market-Leading EDA Tools.** We analyze implementations of IEEE 1735 in six market-leading EDA tools from Xilinx, Intel, Cadence, Siemens, Lattice, and Microsemi. Thereby, we extract *all* cryptographic private keys from these tools leading to a full break of the confidentiality and integrity of the protected IP.
- **Insecure RSA White-Boxes.** We analyze three white-box RSA schemes and present distinct cryptanalytical attacks for each of them: a (1) code-lifting attack that yields a decryption oracle, and a (2) key extraction attack that recovers the hidden secret key within seconds. Our key extraction attacks exploit knowledge on the custom RSA structure to extract the secret key from the obfuscated decryption functions.

Responsible Disclosure. Following standard responsible disclosure principles, we reported the discovered security vulnerabilities to the affected vendors at least three months ahead of publication, through the vendors' vulnerability disclosure programs or technical support platforms. All vendors named in this paper have acknowledged our findings, are working towards (or have already implemented) mitigation improvements, and have agreed to publication. By arrangement with the vendors, we sometimes refrain from providing a detailed elaboration on our reverse engineering process, as this work is not meant to act as a guide to IP theft. Hence, some statements are *deliberately* kept vague.

II. BACKGROUND

In this section, we provide background on the hardware design flow and the role of IP cores. We also summarize key aspects of IEEE standard 1735-2014 and briefly review white-box cryptography.

A. Hardware Design Flow

Hardware design generally follows a multi-stage process distributed across a global supply chain. We distinguish between ASICs that are fixed in their functionality and FPGAs, i.e., reconfigurable ICs that are programmed using a so called *bitstream*.

Design Flow. First, a hardware design is specified and described in an Hardware Description Language (HDL) such as (System)Verilog or VHDL. Next, the design files are analyzed and transformed by a so-called *synthesizer* provided by the EDA vendor. This synthesis step yields a gate-level netlist that is composed of logic gates and their interconnections. Next, the netlist is implemented, i.e., technology-mapped, placed, and routed using tools appropriate for the selected technology. For FPGAs, the output of this implementation process is a bitstream file which is then programmed onto the FPGA. ASIC implementation yields a layout file (e.g., GDSII) describing the technology-mapped and placed-and-routed netlist then sent to a fab for manufacturing, see Figure 1.

Types of IP Cores. Third-party IP can be integrated into the hardware design throughout the entire design process. Three different types of IP are distinguished depending on their level of abstraction, cf. Figure 1. *Soft IP* cores are synthesizable descriptions of an IP core. They offer flexibility to the IP user and may be implemented on a wide range of architectures and technologies. *Firm IP* cores are synthesized gate-level netlists and therefore less flexible than soft IP. The reconstruction of their high-level description is related to the domain of netlist reverse engineering [2, 40, 52]. For FPGAs, *hard IP* cores refer to additional on-device circuitry that can merely be activated and configured by the bitstream, but cannot be retrofitted to the

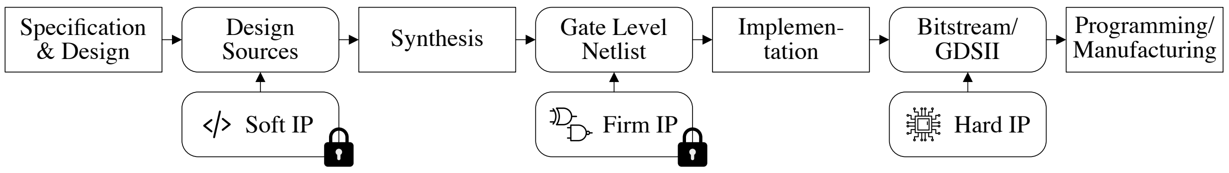

Fig. 1. High-level view of the hardware design flow for FPGAs and ASICs. Different types of third-party IP can be integrated into the design depending on the design stage. IEEE 1735 can be applied to protect both soft and firm IP.

device. In contrast, hard IPs in the ASIC context comprise a technology-mapped and placed-and-routed design commonly provided as a black-box by manufacturers.

B. IEEE Standard 1735-2014

The IEEE standard 1735-2014 [31], also referred to as "*IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)*", refers to a set of guidelines on how to manage and protect electronic design IP. It has been approved and published in late 2015 to promote security and improve interoperability throughout the hardware industry and remains in effect ever since. Note that, although never explicitly being mentioned in IEEE 1735, its setting is related to classical Digital Rights Management (DRM). A revision of the standard has been announced for 2020 [37], but has since been delayed.

Stakeholders. The standard defines three main stakeholders. An *IP author* is the creator and legal owner of an IP core. Safeguarding the IP is a central objective for the IP author in order to protect business interests and prevent potential losses from infringements. An *IP user* acquires rights to an IP core through interaction with the IP author. A key interest of the IP user is to minimize costs associated with a well-proven IP core. Hence, from a business perspective, there is a tension between an IP author and its users. For this reason, IEEE 1735 explicitly represents an IP user as a potential attacker with the goal of infringing on the IP author's design. The standard counters this threat by use of cryptographic protections to safeguard the IP core from illegitimate use. Lastly, the *tool vendor* provides design tools that enable the IP user to interact with the IP of an author.

Digital Envelope. IEEE 1735 dictates the use of a *digital envelope* to protect the IP itself as well as the rights granted by the IP author. Hence, the standard specifies a unified format for the protected IP to ensure security and compatibility between the stakeholders and their tools. It thereby resorts to the use of a mark-up format to customize properties of the IP core as well as the applied protection mechanisms. IEEE 1735 supports both VHDL and (System)Verilog. Hence, its scope is limited to soft and firm IP cores, as illustrated by the locks in Figure 1.

A simplified illustration of the digital envelope is provided in Figure 2. The envelope is split into two sections, the first of which specifies access rights and additional properties. It comprises a number of *right blocks* that specify the rights granted to the IP user by its author within each supported EDA tool. For example, such rights can be used to limit the

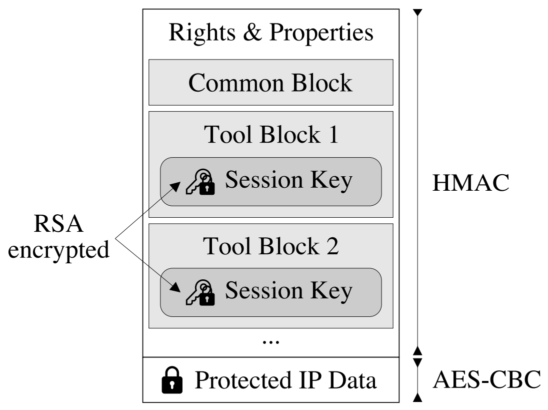

Fig. 2. General structure of the digital envelope as specified by IEEE 1735. The envelope comprises the common and tool-specific right blocks, the encrypted session keys, and the protected IP data.

visible information during simulation or synthesis. There are two different kinds of rights: those that are common across all supported tools denoted as *common block* and those that are specific to a particular tool denoted as *tool block*. The digital envelope comprises one such tool block for every supported EDA tool. Thereby, the standard allows each tool to introduce custom, tool-specific rights. The second section of the envelope contains the actual IP data in encrypted form.

Confidentiality and Integrity. As the IP data section contains the valuable IP sources, upholding its confidentiality is crucial. Therefore, the IP data is encrypted using AES-CBC and a unique 128- or 256-bit symmetric session key. This session key is stored within each vendor's tool block after being encrypted using the corresponding tool vendor's RSA public key. IEEE 1735 mandates the use of RSA keys comprising no less than 2048 bits, though this is not always strictly followed, as is evident from our case studies. Each tool block contains a unique identifier for the required RSA key pair to support multiple different keys per vendor and facilitate adding new keys over time.

For each tool, the session key is also used to compute an HMAC over the common block as well as its tool block using SHA-256 or SHA-512. Consequently, the session key needs to be decrypted before the integrity of the RSA key identifier is verified. The HMAC allows each tool to verify the integrity of the common rights and its own tool-specific rights, but not the IP data itself. Notably, IEEE 1735 does *not* enforce integrity of the encrypted IP data and offers *no* authenticity [17]. We emphasize that an exposed session key implies a break of the confidentiality and integrity of the protected IP.

Most IPs support multiple vendors, hence leakage of a single private key may directly compromise security of the entire scheme. Therefore, the secure handling of each and every private key is vital. As not only the private key but also the plaintext IP reside in program memory during decryption, recovery of the plaintext IP poses an additional threat.

C. RSA

In anticipation of our case studies in Sections IV to IX, we briefly recall RSA encryption and Miller's factorization algorithm [43].

Plain RSA. Let $N = pq$ be an RSA modulus. We denote by $\mathbb{Z}_N$ the ring of integers modulo N with the multiplicative group $\mathbb{Z}_N^*$ of order $\Phi(N) = (p-1)(q-1)$. Let $e \in \mathbb{Z}_{\Phi(N)}^*$ be a public encryption exponent with corresponding RSA private key d satisfying $ed = 1 \bmod \Phi(N)$. The RSA encryption function is the map $\mathbb{Z}_N \to \mathbb{Z}_N, m \mapsto m^e \bmod N$. The RSA decryption function is the map $\mathbb{Z}_N \to \mathbb{Z}_N, c \mapsto c^d \bmod N$.

Private Key Implies Factorization. Given public exponent e and private key d, Miller's well-known probabilistic reduction from RSA secret recovery to factoring [43] can be applied. Let $ed - 1 = 2^k t = 0 \bmod N$ with odd t. We take a random $g \in \mathbb{Z}_N^*$ until $g^t \neq 1 \bmod N$. Let us then square g^t until we obtain 1 as result. Let $c < k$ be minimal such that $g^{2^{c+1}t} = 1 \bmod N$. Since 1 has four square roots modulo N, we obtain $g^{2^c t} \neq \pm 1 \bmod N$ with probability $\frac{1}{2}$. In this case

$$\gcd(g^{2^c t} \pm 1, N) = \{p, q\}$$

directly reveals the factorization of $N = pq$.

D. White-Box Cryptography

We now recall standard notions of white-box cryptography in anticipation of the white-box case studies in Sections VII, VIII and IX. White-box cryptography and the associated white-box attacker model were introduced in the seminal work of Chow et al. [18, 19]. The goal of white-box cryptography is to protect a cryptographic secret such as a key within the implementation of a cryptographic algorithm itself.

Attacker Model. In traditional *black-box* scenarios, the attacker has access to the cryptographic algorithm, can adaptively choose input plaintexts and/or ciphertexts, and observe the algorithm outputs. However, the dynamic execution of the algorithm remains hidden from their eyes. In contrast, a *white-box* setting grants the attacker full access to the software encompassing the cryptographic algorithm and its execution environment. The attacker may thus perform static code analysis, arbitrarily execute the white-box algorithm, examine intermediate values in memory, and dynamically manipulate these values during execution. White-box cryptography aims to provide security even in such hostile environments.

Security Goals. The two primary security goals of white-box cryptography are formalized as *security against key-extraction* and *security against code-lifting* [18, 19]. A white-box cryptographic algorithm is secure against key extraction, if and only if an attacker cannot recover the embedded cryptographic secret from the implementation. However, a white-box scheme that is *only* secure against key extraction might not offer any security at all. Even without any knowledge of the secret key, an attacker that can extract the white-box algorithm itself can then execute the algorithm and thereby yield an encryption/decryption oracle. A white-box cryptographic algorithm that provides security against code-lifting attacks cannot be extracted, i.e., *lifted*, from its application while maintaining functionality.

Public-Key Cryptography. Most academic work on the topic deals with symmetric white-box techniques, the only exception – to the best of our knowledge – is the work by Barthelemy [4] based on a lattice-based scheme. Furthermore, the CHES 2021 WhibOx contest [23] aimed to obfuscate an ECDSA signature scheme to be secure in a white-box setting, however, all submissions have successfully been attacked after at most two days. While the symmetric white-box setting has extensively been formalized [3, 9, 24, 49], public-key white-box cryptography lacks any such formalization. Despite these shortcomings and the limited availability of public research, numerous companies offer public-key white-box solutions [25, 35, 54], and hold patents in that area [28, 29, 30, 41, 61].

III. Attacks on IEEE Standard 1735-2014

We now assess the security of IEEE standard 1735-2014. To this end, we first outline the classic Man-at-the-End (MATE) attacker model and then identify foundational security flaws in the trust model assumption and the key management of IEEE 1735. The case studies presented in Sections IV to IX then demonstrate the severe implications of the discovered flaws in practice.

A. Attacker Model

Attacker Capabilities. Our attacker model is inspired by the well-established MATE attack scenario [1]. We consider an attacker with full control over the EDA tool execution environment (that contains the secret decryption keys). The attacker is able to thoroughly analyze the EDA tool software by performing static and dynamic software analysis. In particular, the attacker is able to interact with the application at runtime and can thus manipulate and extract intermediate values during execution. Note that this setting directly corresponds to the white-box attacker model outlined in Section II-D.

Attacker Goal. The attacker's objective is to break the confidentiality and integrity of the protected IP to uncover and/or manipulate the plaintext IP. More precisely, the attacker aims to recover the decrypted session key provided within the EDA tool rights block of the digital envelope as specified by IEEE 1735 (see Section II-B). Extracting the hard-coded RSA private key from the EDA tool fulfills this goal as it allows for decryption of the session key.

Attacker Skill Set To contextualize the attack time per case study denoted in Table I, we briefly summarize our skill set. The reverse engineers conducting our case studies are a Master's degree student with eight years of experience in the field of binary analysis and a PhD student working in the area hardware security for five years. The attacks on the RSA white-box decryption algorithms were conducted in collaboration with a cryptography expert with 25 years of experience in cryptanalysis.

B. Principle Security Considerations

In light of an ongoing revision of the standard [37], we detail flaws in the protocol that break the alleged confidentiality and integrity of the protected IP. Despite not giving explicit strong security guarantees, the standard is trusted throughout the industry:

"*There are specific IEEE standards for evaluation and comprehension. The effort for cracking this encryption is so high that it is difficult even for large firms.*" [26, pp. 139-140]

Untrusted Execution Environment. Since EDA tools commonly run on the workstation(s) of IP users, IEEE 1735 operates within a notoriously untrusted execution environment, i.e., an adversary has full control over the execution of said EDA tools, see Section III-A. In this hostile setting, the standard attempts to provide confidentiality and (in parts) integrity of the protected IP by employing both public-key and symmetric cryptography. However, even in the presence of impeccable protection measures and analogously to DRM use cases, the plaintext IP will reside in memory during tool execution and thus can be recovered by an adversary. To counter this threat, the standard encourages tool vendors to legally prevent reverse engineering of the design tools altogether:

"*Between tool vendors and their users [...], an agreement for use of the tool should forbid tampering and reverse engineering without being granted explicit permission.*" [31, p. 11]

However, in reality, an attacker who is willing to commit IP theft or insert malicious circuitry into a protected IP is unlikely to be stopped by a legal agreement as both attacks are already illegal on their own.

Hard-Coded Keys. To our surprise, IEEE 1735 recommends the same RSA private key to be used across all instances of an EDA tool. The standard furthermore proposes to *hard-code* these keys into the design software itself:

"*One key should be used by a single product or a set of closely related products from one company that shares a common code base.*" [31, p. 25]

"*It may harden such secrets directly into its software or use an external persistent storage scheme.*" [31, p. 20]

As becomes evident by our case studies in Sections IV to IX, not a single vendor resorts to an "external storage scheme" (e.g., dongles) as it is regarded to be impractical and not scaleable for most use cases. An adversary who is able to recover a private key, e.g., by means of reverse engineering, can subsequently decrypt and manipulate each and every protected IP. Hence, this single point of failure constitutes a valuable target for attackers.

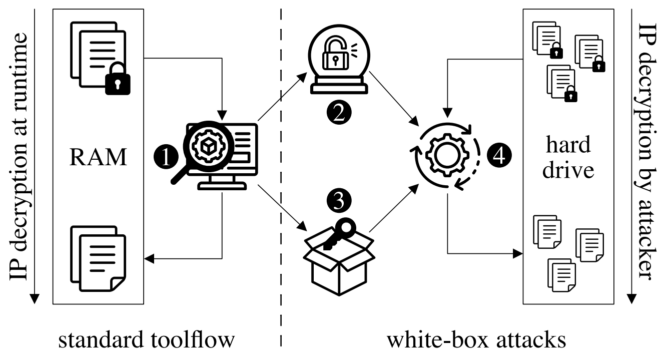

Fig. 3. High-level overview of the standard EDA tool flow for IP decryption as well as code-lifting and key-extraction attacks targeting a white-box implementation.

Lack of Guidance. Although the threat of key extraction is addressed within IEEE 1735, the standard omits a clear discussion on the importance of private key protection and regards the disclosure of IP as an implementation issue:

"*If a disclosure happens because a tool is hacked during execution by an untrusted IP user, using any combination of binary tampering and debugging techniques, that is an implementation issue. Each tool vendor needs to decide what, if any, anti-debug, anti-tamper, and anti-key-discovery technologies they will use here.*" [31, p. 11]

The standard mentions that it would be in the tool vendor's best business interest to invest in protecting its private keys, however, no guidance on a secure implementation of the key storage scheme is provided. Moreover there is a limited incentive to invest into proper software protections for the private keys as only few tool vendors sell and license their own IP. Just as in a DRM setting, the security of the entire scheme is founded upon the protection of the private keys, hence the failure to address these threats appropriately has devastating consequences in practice.

C. White-Box Implementations

White-box cryptography (see Section II-D), despite not being mentioned in the standard, appears to be a natural fit for the key management scheme introduced by IEEE 1735 due to its DRM-like nature. Hence, some vendors use white-box implementations to protect their private keys. Note that we adopt the framing of the discovered algorithms as *white-box* cryptography from the vendors to then evaluate their implementations within the white-box model in Sections VII, VIII and IX. Thereby, we reveal that these commercial white-box solutions do not fulfill the notion of white-box cryptography and fail to uphold its basic security requirements, i.e., *security against key extraction* and *security against code-lifting*.

White-Box Attack Strategies. Figure 3 gives a high-level overview of the standard tool flow for IP decryption and our overall attack strategy. By default, the EDA tool loads the protected IP into program memory ahead of decryption. It proceeds to decrypt the symmetric session key using the RSA private key specified within the IP's digital envelope. The session key is then used for the decryption of the protected IP data. After decryption, the plaintext IP resides within program memory awaiting operations such as synthesis or simulation.

Our attack strategy evolves around the two central security properties of white-box cryptography, i.e., *security against code-lifting* and *security against key extraction*. For every case study involving a white-box, ❶ we analyze the decryption process using static and dynamic software reverse-engineering techniques to locate the cryptographic subroutines. Having identified the white-box RSA decryption algorithm, ❷ we proceed to lift it from the main application by replicating it in a high-level language. This replica can be seen as a decryption oracle, as it carries the private key embedded within its implementation. Given the replicated white-box, we generate an abstract, formalized description of the white-box algorithm. Based upon this description, ❸ we develop key-extraction attacks to recover the hidden private key. From this point on, by keeping a record of either the replicated white-box algorithm or the recovered secret key, ❹ we can decrypt the symmetric session key and consequently uncover the plaintext of any arbitrary IP protected by IEEE 1735 and destined for use within the respective tool. Both attack strategies combined break the two crucial security properties of white-box cryptography.

IV. Case Study: Intel Quartus Prime

Since the acquisition of Altera, Intel has grown to be the second largest vendor of FPGAs in the market. Intel Quartus Prime is their main EDA tool for FPGA design. Our analysis deals with Intel Quartus Prime in version 21.1 for Windows. At first, we identify Dynamic Link Libraries (DLLs) of interest by searching for the documented [34] RSA key name in program memory at runtime. Thereby, we uncover two potential target DLLs, one of which only provides IEEE 1735 encryption capabilities and contains public keys of other vendors. The other DLL is (at least partially) equipped with symbols that, f.i., provide human-readable names to the library functions and thereby support the reverse engineering process. In order to identify cryptographic subroutines, we automatically scan for unique strings such as `key_keyname` using a static analysis tool. According to IEEE 1735, `key_keyname` precedes the actual key identifier within the digital envelope. By analyzing the string references, we uncover a function called `assemble_key`, which prepares the RSA private key for decryption. Note that the private key is solely protected by XORing of two hard-coded byte-arrays. Hence, revealing the RSA private key is straight-forward.

V. Case Study: Cadence Xcelium

Cadence Design Systems develops EDA tools for ASIC design that see widespread use throughout the industry. Cadence Xcelium is a verification tool based on logic simulation, which provides encryption and decryption capabilities in compliance with IEEE 1735. In this case study, we analyze Xcelium 21.0 on CentOS. By going through user documentation [13], we identify two target keys, one of which is an insecure (and by now deprecated) 512-bit RSA key that has been in active use until approximately 2015. Due to the availability of symbols and the absence of code obfuscation, automated scanning of the executable for the documented RSA key names reveals the target function. When this function is invoked, one of two other functions is called depending on the selected secret key. These functions both return a static memory address pointing to the private key stored in plaintext. Hence, reverse engineering and recovering the RSA private key is straightforward.

VI. Case Study: Lattice Radiant

Lattice Semiconductor is one of the four major FPGA vendors. They provide Lattice Radiant to develop hardware designs for their FPGAs. By investigating Lattice's implementation of IEEE 1735 within Lattice Radiant 2.2.1 for Windows, we identify the target library by again automatically searching for the documented key name [38]. Thereby, we discover a function that loads a secret key which is identified by name. In order to extract the RSA private key from the decryption context, we follow the references to callers of that function. We subsequently identify another function that takes as input a key name, the encrypted data, and an empty buffer apparently reserved for the decrypted data. The discovered decryption function then calls the RSA decrypt function provided by OpenSSL. Next, we inject custom code triggering the decryption on the targeted private key and subsequently extract the RSA private key from the OpenSSL decryption at runtime.

VII. Case Study: Microsemi Libero SoC

Microsemi is one of the four major FPGA companies and provides solutions for aerospace, defense, communications, and industrial markets. In this case study, we focus on Libero SoC v2021.1 for Windows, their current FPGA design tool.

A. Reverse Engineering

Similar to the first three case studies, automatic scanning for the RSA key name [42] reveals a valid 2048-bit RSA private key in program memory. However, since we do not have access to an encrypted IP or the corresponding RSA public key, we could not verify its correctness on our own. After consultation with Microsemi, this key turned out to be a decoy.

We proceed by dynamically analyzing the executable. To this end, we trigger the RSA decryption routine and observe its operation at runtime by executing Microsemi Libero SoC and importing a specifically crafted IP. We observe that Libero SoC first feeds the encrypted soft IP into a synthesizer which in turn yields a synthesized and re-encrypted IP at netlist-level. Next, we turn our attention to the subprocess calls taking place on this re-encrypted netlist, as this is the first time Libero SoC itself interacts with the encrypted files via a dedicated executable. We observe that this executable is dynamically linked against a DLL equipped with code obfuscation which hints at its valuable content.

We again detect instances of the RSA key name within this library, but since static analysis appears to be cumbersome due to obfuscation, we opt for a dynamic approach instead. However, the binary is equipped with debugger detection mechanisms and terminates standard execution when attaching a debugger. We defeat (almost) all anti-debugging measures protecting the DLL using ScyllaHide [14]. Moreover, we discover the use of integrity checks throughout the library, preventing us from inserting software breakpoints, but evade this protection by the use of hardware breakpoints. We discover the ciphertext, i.e., the RSA-encrypted session key, in memory as it is passed as an input to a function that is called in close proximity to the discovered RSA key name. Hence, we place a (hardware-based) memory access breakpoint on the ciphertext, which holds execution upon the first access to the monitored memory address. In our case, the breakpoint is triggered at the entry point to the cryptographic computations. Thereby, we skip the obfuscated parts of the binary and reveal the (unobfuscated) functions that are responsible for decryption of the session key.

B. Private Key Recovery

The RSA decryption is implemented using Montgomery modular multiplications [46], as revealed by static analysis of the unobfuscated cryptographic functions. Note that this technique is typically used to speed up subsequent multiplications as required for the modular exponentiation of RSA. By identifying the forward and backward transformations into and from Montgomery domain, we discover the main RSA decryption routine between these two function calls. We observe that the control flow of the decryption differs between executions, utilizing multiple distinct exponentiation algorithms. By tracing through a single decryption and dumping inputs and outputs for each exponentiation, we reconstruct an entire decryption run. Our investigation shows an extended version d_ℓ of Microsemi's 7680-bit RSA private key d to be split into four hard-coded shares, i.e., two 7680-bit numbers d_1, d_2 and the two 32-bit values d_3, d_4. Further analysis reveals that d_ℓ is computed as

$$d_\ell = d_1 \cdot d_2 + d_3 \cdot 2^{32} + d_4 = d + k \cdot \varPhi(N),\ k \in \mathbb{N}.$$

Note that Microsemi refers to this implementation as a white-box, hence we have indicated this within Table I using braces.

For recovery of the original 7680-bit private RSA key $d = d_\ell \bmod \varPhi(N)$, we factor $N = pq$ to compute $\varPhi(N) = (p-1)(q-1)$. To this end, we leverage the well-known factorization approach outlined in Section II-C, which also works for d_ℓ.

VIII. Case Study: Siemens ModelSim

Ever since incorporating Mentor Graphics in 2016, ModelSim is Siemens' tool for Register Transfer Level (RTL) as well as netlist-level simulation and comes with support for IEEE 1735 [50]. Siemens ModelSim is commonly bundled with other EDA tools and IP encrypted for these tools often includes an additional tool block for ModelSim, see Section II-B. Hence, extracting the private RSA keys from Siemens ModelSim impacts multiple vendors at once.

A. Reverse Engineering

We target ModelSim Pro version 2020.4 for Windows. In order to trigger and inspect the decryption at runtime, we load a protected IP into the EDA tool similar to previous case studies. As we could not identify cryptographic functions within the main ModelSim executable using static analysis, we attach a debugger and track all sub-process calls. Thereby, we identify a sub-process that handles decryption of the protected IP, as indicated by its console output. Using static analysis, we verify the use of a well-known cryptographic library within this sub-process, but are unable to identify all cryptographic functions right away. However, the binary is still equipped with assert statements, which include the file path and line number of the original asserts within the source code. Thus, these asserts help us to identify the missing functions by matching file names and line numbers with the open-source cryptographic library implementation.

B. Code-Lifting Attack

Through dynamic analysis, we discover that the standard RSA decryption is not called during execution. Instead, a custom decryption routine is implemented using the Application Programming Interface (API) provided by the cryptographic library. Another assert statement discovered within the binary hints at an RSA white-box implementation using the Chinese Remainder Theorem (CRT).

In order to recover the session key, we first opt for a code-lifting attack that reconstructs the white-box decryption to be used as an oracle, see Section II-D. To this end, we reconstruct the white-box algorithm in a high-level language. We verify correct functionality by decrypting an arbitrary encrypted IP using our implementation. During analysis we discover support for three distinct RSA key pairs, two 1024-bit and one 2048-bit pair. The same decryption routine is called for all three private RSA keys. Given the white-box data associated with each key, we can then decrypt protected IP even without knowledge of the keys.

This reconstruction does not only break the essential white-box security notion of *security against code-lifting attacks*, but also serves as the basis for the understanding of the underlying white-box algorithm as well as the extraction of the secret keys in the next section.

C. Private Key Extraction

CRT-RSA. Let us recall standard CRT-RSA decryption as shown in Algorithm 1. Let $d_p = d \bmod p-1$ and $d_q = d \bmod q-1$. We compute $c^d \bmod N$ via $c^d = c^{d_p} \bmod p$ and $c^d = c^{d_q} \bmod q$, and eventually combine both results using the CRT.

Algorithm 1 CRT-Exp($c, d, d_p, d_q, p, q, \gamma, N$)

```
1: $m_p = c^{d_p} \bmod p$.
2: $m_q = c^{d_q} \bmod q$.
3: $m = (m_p - m_q)\gamma + m_q \bmod N$          ▷ apply CRT
4: return $m$
```

The CRT-parameter γ satisfies

$$\gamma = \begin{cases} 1 \bmod p, \\ 0 \bmod q. \end{cases} \tag{1}$$

Thus, in Step 3 of Algorithm 1 we compute $m \in \mathbb{Z}_N$ satisfying

$$m = \begin{cases} m_p = c^{d_p} = c^d \bmod p, \\ m_q = c^{d_q} = c^d \bmod q. \end{cases}$$

Computing CRT-Exp is four times faster than computing $c^d \bmod N$ directly, hence it is commonly applied in practice to speed up RSA exponentiations. However, CRT-Exp is also much harder to obfuscate, since *all five* parameters d_p, d_q, p, q, γ directly reveal the factorization of N. This is obvious for p, q. For d_p we have $c^{d_p} - c = 0 \mod p$ for any c, and therefore $\gcd(N, c^{d_p} - c) = p$, analogous for d_q. And last, by Equation (1) we have $\gcd(N, \gamma - 1) = p$ and $\gcd(N, \gamma) = q$.

Obfuscated CRT-RSA. We now outline how Siemens obfuscates the critical values d_p, d_q, p, q, γ, where the first four are 512-bit each.

d_p, d_q**:** Additively split $d_p = d_{p,1} + d_{p,2}$ and $d_q = d_{q,1} + d_{q,2}$.

p, q**:** Use moduli $\tilde{p} = kp$, $\tilde{q} = \ell q$ with 8-bit integers k and ℓ. Notice that if we compute $m_p = c^{d_p} \bmod \tilde{p}$, then m_p also holds modulo p. Thus, the correctness of Algorithm 1 is maintained. However, $\tilde{p}$ and $\tilde{q}$ still reveal the factorization by computing $p = \gcd(\tilde{p}, N)$ or $q = \gcd(\tilde{q}, N)$. Therefore, Siemens ModelSim additively splits $\tilde{p} = p_1 - p_2$ and $\tilde{q} = q_1 - q_2$, where p_2, q_2 are 256-bit numbers. Siemens ModelSim's sophisticated obfuscation technique Obf-Mod now performs a reduction modulo $\tilde{p}$ using only the split p_1, p_2, see Algorithm 3.

γ**:** The parameter is split multiplicatively and additively, thereby yielding $\gamma = \gamma_1(\gamma_2 - \gamma_3)$.

Altogether, this results in the obfuscated CRT-RSA exponentiation shown in Algorithm 2.

Algorithm 2 Obf-CRT-Exp(c, $d_{p,1}$, $d_{p,2}$, $d_{q,1}$, $d_{q,2}$, p_1, p_2, γ_1, γ_2, γ_3, N)

```
1: $m_p = \text{Obf-Mod}(c^{d_{p,1}} \cdot c^{d_{p,2}}, p_1, p_2)$
2: $m_q = \text{Obf-Mod}(c^{d_{q,1}} \cdot c^{d_{q,2}}, q_1, q_2)$
3: $m = (m_p - m_q)\gamma_1(\gamma_2 + \gamma_3) + m_q \bmod N$  ▷ apply CRT
4: return $m$
```

Let us now have a closer look at Obf-Mod in Algorithm 3 that performs the obfuscated modular reduction of some parameter a modulo $\tilde{p} = p_1 - p_2$ using only p_1, p_2.

Algorithm 3 Obf-Mod(a, p_1, p_2)

```
1: Set $q = \lfloor \frac{a}{p_1} \rfloor$.
2: Set $r = a \bmod p_1$.
3: return $q \cdot p_2 + r \bmod N$
```

Notice that Obf-Mod maps a to the value

$$\begin{aligned} q \cdot p_2 + r &= \lfloor \frac{a}{p_1} \rfloor \cdot p_2 + (a \bmod p_1) \\ &= \lfloor \frac{a}{p_1} \rfloor \cdot p_2 + a - \lfloor \frac{a}{p_1} \rfloor \cdot p_1 \\ &= a - \lfloor \frac{a}{p_1} \rfloor \cdot (p_1 - p_2) \\ &= a - \lfloor \frac{a}{p_1} \rfloor \cdot \tilde{p}. \end{aligned}$$

Thus, Obf-Mod indeed reduces a by a multiple of $\tilde{p}$. However, notice that Obf-Mod *does not* realize the standard Euclidean reduction $a - \lfloor \frac{a}{\tilde{p}} \rfloor \cdot \tilde{p}$ with a remainder in $[0, \tilde{p})$. Nonetheless, since $p_2 \ll p_1$ we have $\tilde{p} \approx p_1$, and therefore a gets sufficiently reduced.

Private Key Recovery. We extract $\gamma = \gamma_1(\gamma_2 + \gamma_3)$ and compute the factorization of N as $\gcd(\gamma - 1, N) = p$ and $\gcd(\gamma, N) = q$. For verification we check whether for $\tilde{p} = kp = p_1 - p_2$ and $\tilde{q} = \ell q = q_1 - q_2$ it holds that $\gcd(N, \tilde{p}) = p$ and $\gcd(N, \tilde{q}) = q$. Eventually, we compute $d_p = d_{p,1} + d_{p,2}$ and $d_q = d_{q,1} + d_{q,2}$ to verify that for random $c \in \mathbb{Z}_N$ we have $\gcd(N, c^{d_p} - c) = p$ and $\gcd(N, c^{d_q} - c) = q$. Recovering d now is as simple as $d = e^{-1} \bmod \Phi(N)$.

IX. Case Study: Xilinx Vivado Design Suite

Xilinx is the world's largest manufacturer of FPGAs. *Xilinx Vivado Design Suite* is their current EDA tool for FPGA development. We target *Xilinx Vivado* version 2020.2 for Windows, but have verified applicability to 2019.2 and 2020.1.

Remark. Xilinx makes use of white-box cryptography to protect (some of) its RSA private keys. After concluding our work on Xilinx Vivado in August 2020, we discovered another white-box attack [55] on Google's Widevine DRM solution published on October 31, 2020. We verified that Widevine used the same white-box scheme as Xilinx, hence indicating that the white-box is supplied by a third-party vendor.

A. Reverse Engineering

As in previous case studies, we begin by exploring the Xilinx Vivado executable using static analysis. We locate implementations of cryptographic primitives by scanning the program memory for cryptographic constants during decryption. Thereby, we determine that Xilinx Vivado uses a well known cryptographic library to handle its private key. Moreover, Xilinx secures Vivado using control-flow obfuscation, debugger detection, and exception-based anti-debugging techniques among others, see [15, 16, 20] for a comprehensive overview on software protection techniques. However, the discovered cryptographic functions do not exhibit any such software protections.

The targeted Xilinx Vivado installation supports five different 2048-bit RSA key pairs [58]. Our analysis subsequently reveals one of the five RSA private keys using dynamic analysis. We verify correctness by decryption of an openly-available IP encrypted using the corresponding public key.

B. Code-Lifting Attack

Using dynamic analysis, we observe that the discovered RSA decryption routine is no longer triggered for the four other RSA private keys. In order to identify the decryption routine used for these keys, we statically follow the call graph of the discovered decryption, aiming to locate the dispatcher calling the respective decryption functions for the different keys. However, large parts of the functions within the call-graph (excluding cryptographic primitives) are protected by software obfuscation. By getting around these protective measures, we reveal a reference to another presumably cryptographic function. Observing the inputs and outputs of the newly discovered function using debugging reveals that it indeed performs an RSA decryption and is only invoked for the remaining four RSA private keys. Due to the non-standard RSA decryption implementation that is heavily based on custom precomputed tables, we conclude that the function implements an RSA white-box. By triggering the RSA decryption on different private keys, we observe that the same code is executed for all four available white-box keys, but different data structures are accessed depending on the selected key.

To obtain a decryption oracle that allows to recover the RSA-encrypted session keys, we perform a code-lifting attack by replicating the white-box RSA decryption in Python. The replication of the white-box decryption breaks the white-box property of *security against code-lifting attacks*.

C. White-Box Simplification

In anticipation of the private key extraction attack in the next section, we briefly describe high-level implementation concepts and our simplifications thereof. Note that manipulations to the implementation of the RSA white-box are in line with our attacker model defined in Section III.

Montgomery Multiplications. First, we identify the use of Montgomery multiplications [46] by identifying Montgomery-specific constants such as $R = 2^{|d|} \bmod N$ for private key bit-length $|d|$ and modulus N within the extracted data structures. To simplify the algorithmic description of the white-box implementation, we replace all Montgomery transformations and multiplications with isomorphic standard modular operations.

Ciphertext Dependency. The white-box algorithm utilizes pre-computed values, some of which are selected based on the input ciphertext. This ciphertext-dependency is introduced by first evaluating a hash function on the ciphertext and then applying multiplicative masking to the ciphertext based on the hash function output. We observe that these values do not impact the decryption as verified by fixing the hash function output to zero and observing the output over multiple decryption runs. Therefore, we conclude that we can safely remove this ciphertext-dependency.

D. White-Box Description

This section gives a formal description of the white-box RSA decryption routine we uncovered from within Xilinx Vivado. Based upon this description, we subsequently present two sophisticated attacks that instantly lead to a full recovery of the protected RSA secret key in Section IX-E.

Notation. We denote vectors as bold-face, lowercase letters such as $\mathbf{v}$ and matrices as bold-face, uppercase letters like $\mathbf{A}$. We address vector and matrix entries as $\mathbf{v}[i]$ and $\mathbf{A}[i][j]$ respectively.

Ordinary RSA Decryption. Let us first consider an ordinary RSA decryption. Our goal is to recover the plaintext m by computing the exponentiation $c^d = m \bmod N$. We introduce a window technique whose usual purpose is to provide a time-memory trade-off for fast exponentiation, but in our case additionally lies at the heart of Xilinx Vivado's white-box implementation. Consider a secret decryption key d of length n that is split into $k = \lceil \frac{n}{5} \rceil$ chunks of five bits each, i.e.,

$$d = d_0 + d_1 \cdot 2^5 + d_2 \cdot 2^{10} + \ldots + d_k \cdot 2^{5k} \text{ with } d_i \in \mathbb{Z}_{32}.$$

We thus write $\mathbf{d} = (d_0, d_1, \ldots, d_k) \in \mathbb{Z}_{32}^{k+1}$ for the vector containing these secret key chunks. Then it follows that

$$m = c^d = c^{d_0 + d_1 \cdot 32 + d_2 \cdot 32^2 + \ldots + d_k \cdot 32^k} = \prod_{i=0}^{k} \left(c^{d_i}\right)^{32^i}$$
$$= \left(\left((c^{d_k})^{32} \cdot c^{d_{k-1}}\right)^{32} \cdot \ldots \cdot c^{d_1}\right)^{32} \cdot c^{d_0} \bmod N. \quad (2)$$

Note that we can precompute the ciphertext-dependent vector $\mathbf{c} = (c^0, c^1, \ldots, c^{31}) \in \mathbb{Z}_N^{32}$ once at the beginning and use it as a lookup table throughout the decryption process. Thus, the computation of Equation (2) is realized by procedure EXP($\mathbf{c}, \mathbf{d}$) in Algorithm 4 comprising only k multiplications and exponentiations in $\mathbb{Z}_N$.

Algorithm 4 EXP($\mathbf{c}, \mathbf{d}$)

```
1: $m = \mathbf{c}[\mathbf{d}[k]] = c^{d_k}$
2: for $i = k - 1$ downto 0 do
3:     $m = m^{32} \cdot \mathbf{c}[\mathbf{d}[i]] = m^{32} \cdot c^{d_i} \bmod N$
4: end for
5: return $m$
```

Hiding d. Notice that in its current form, Algorithm 4 directly operates on (and thereby reveals) $\mathbf{d} = (d_0, \ldots, d_k) \in \mathbb{Z}_{32}^{k+1}$ as input. Hence, an attacker could straightforward recover the secret key $d = \sum_{i=0}^{k} d_i \cdot 32^i$ from memory.

The core idea behind the discovered white-box approach is to hide all original 5-bit values $d_i \in \mathbb{Z}_{32}$ of $\mathbf{d}$ via some secret *permutation* $\pi : \mathbf{Z}_{32} \to \mathbf{Z}_{32}$. The bijection π is hard-coded within the implementation in an obfuscated manner. Each of the four available secret keys using the white-box utilizes a different π.

Notice that π as well as the inverse permutation π^{-1} can efficiently be represented by providing a lookup table with values $\pi(0), \ldots, \pi(31)$ requiring only $32 \cdot 5 = 160$ bits.

As a result, instead of $\mathbf{d}$ itself, only the obfuscated key vector $\hat{\mathbf{d}} = (\pi^{-1}(d_0), \ldots, \pi^{-1}(d_k))$ is revealed during execution of the white-box algorithm. Since the computation $\pi(\hat{\mathbf{d}}) = \mathbf{d}$ immediately yields the original secret key, an attacker may find considerable value in trying to recover the secret permutation π. However, the number of permutations on $\mathbf{Z}_{32}$ is given as $32! > 2^{117}$. Therefore, trying to guess π using a brute-force approach is infeasible. This also provides reasoning for the choice of a chunk size of $k = 5$ bits, since it is the smallest k that provides sufficient security against brute-force attacks.

Now, let us precompute the vector $\bar{\mathbf{c}} = (c^{\pi(0)}, \ldots, c^{\pi(31)})$. When subsequently executing $\text{EXP}(\bar{\mathbf{c}}, \hat{\mathbf{d}})$ on the permuted inputs $(\bar{\mathbf{c}}, \hat{\mathbf{d}})$ instead of $(\mathbf{c}, \mathbf{d})$, the intermediate values

$$\bar{\mathbf{c}}[\hat{\mathbf{d}}[i]] = \bar{\mathbf{c}}[\pi^{-1}(d_i)] = c^{\pi(\pi^{-1}(d_i))} = c^{d_i}.$$

are computed. Thus, $\text{EXP}(\bar{\mathbf{c}}, \hat{\mathbf{d}})$ outputs the same plaintext $m = c^d$ as $\text{EXP}(\mathbf{c}, \mathbf{d})$ does on the original inputs.

Randomization of $\bar{\mathbf{c}}$. Notice that an attacker knows the ciphertext c and thus can compute $c^0, \ldots, c^{31}$ himself. As a result, $\bar{\mathbf{c}} = (c^{\pi(0)}, \ldots, c^{\pi(31)})$ would immediately reveal the secret permutation π. In order to prevent this disclosure of π, a randomized and obfuscated precomputation vector $\hat{\mathbf{c}}$ has to be defined. To this end set

$$\begin{aligned} &\mathbf{r} = (r_0, \ldots, r_{31}) \text{ for some } r_i \in_R \mathbb{Z}_N^* \text{ and} \\ &\hat{\mathbf{c}} = \mathbf{r} \cdot \bar{\mathbf{c}} = (r_{\pi(0)} \cdot c^{\pi(0)}, \ldots, r_{\pi(31)} \cdot c^{\pi(31)}). \end{aligned} \tag{3}$$

Since the r_i are chosen uniformly at random, $\hat{\mathbf{c}}$ does not reveal any information about π. Looking at the output of an execution of $\text{EXP}(\hat{\mathbf{c}}, \hat{\mathbf{d}})$, we notice that the intermediate values are now of the form

$$\hat{\mathbf{c}}[\hat{\mathbf{d}}[i]] = \hat{\mathbf{c}}[\pi^{-1}(d_i)] = r_{\pi(\pi^{-1}(d_i))} \cdot c^{\pi(\pi^{-1}(d_i))} = r_{d_i} \cdot c^{d_i}.$$

Given Equation (2), it is not hard to see that $\text{EXP}(\hat{\mathbf{c}}, \hat{\mathbf{d}})$ therefore produces the output

$$\begin{aligned} &\left(\left((r_{d_k} \cdot c^{d_k})^{32} \cdot r_{d_{k-1}} c^{d_{k-1}}\right)^{32} \cdot \ldots \cdot r_{d_1} c^{d_1}\right)^{32} \cdot r_{d_0} c^{d_0} \\ &= c^d \cdot \prod_{i=0}^{k} r_{d_i}^{32^i} = m \cdot \prod_{i=0}^{k} r_{d_i}^{32^i} \bmod N. \end{aligned}$$

Now define the constant term $r = \prod_{i=0}^{k} r_{d_i}^{-32^i} \bmod N$. In order to reconstruct the desired RSA plaintext m one has to multiply the output of $\text{EXP}(\hat{\mathbf{c}}, \hat{\mathbf{d}})$ by r. Subsequently we obtain

$$c^d = \text{EXP}(\mathbf{c}, \mathbf{d}) = r \cdot \text{EXP}(\hat{\mathbf{c}}, \hat{\mathbf{d}}). \tag{4}$$

During the precomputation step an attacker can not directly compute $\hat{\mathbf{c}} = (r_{\pi(0)} \cdot c^{\pi(0)}, \ldots, r_{\pi(31)} \cdot c^{\pi(31)})$, since the implementation hides the values of r_i and $\pi(i)$ for $i \in \{0, \ldots, 31\}$ from the attacker. To this end, the discovered cryptographic algorithm presents a clever approach towards realizing a white-box RSA decryption.

Realization of r and π. As revealed by the reverse engineering efforts described in Section IX-B, Xilinx Vivado executes the white-box RSA precomputation shown in Algorithm 5 on any given ciphertext $c \in \mathbb{Z}_N$. Algorithm 5 computes a vector $\mathbf{s} \in \mathbb{Z}_N^{33}$ comprising powers $s_i = (\alpha c + \beta)^i$ of the linearly transformed ciphertext c for all $0 \leq i \leq 32$ using some hard-coded constants $\alpha, \beta \in \mathbb{Z}_N$. As we will show in the following, the fixed matrix $\mathbf{A} \in \mathbb{Z}_N^{32 \times 33}$ utilized in Algorithm 5 realizes (and hides) the secret permutation π on $\mathbb{Z}_{32}$ and the randomization vector $\mathbf{r} = (r_0, \ldots, r_{31})$ from Equation (3).

Algorithm 5 OBFUSC-PRECOMP(c)

```
1: Use hard-coded $\mathbf{A} \in \mathbb{Z}_N^{32\times 33}, \mathbf{t'} = (t_0, \ldots, t_{31}) \in \mathbb{Z}_N^{32}, \alpha, \beta \in \mathbb{Z}_N$.
2: $\mathbf{s} = (s_0, \ldots, s_{32})$ with $s_i = (\alpha c + \beta)^i \bmod N$
3: $\mathbf{t} = -c^{32} \cdot \mathbf{t}'$
4: return $\mathbf{As} + \mathbf{t} = \hat{\mathbf{c}} = (r_{\pi(0)} \cdot c^{\pi(0)}, \ldots, r_{\pi(31)} \cdot c^{\pi(31)})$
```

Given just the description of Algorithm 5, it remains unclear how the output $\mathbf{As} + \mathbf{t}$ realizes an obfuscated vector $\hat{\mathbf{c}}$. Using the *binomial theorem*, we can write s_i as

$$s_i = (\alpha c + \beta)^i = \sum_{j=0}^{i} \binom{i}{j} (\alpha c)^j \beta^{i-j} \quad \forall i : 0 \leq i \leq 32.$$

Thus, within the sum making up s_i the coefficient of c^j with $j \leq i$ is given as $\binom{i}{j}\alpha^j\beta^{i-j}$. Let us subsequently define the following lower-triangular matrix $\mathbf{M} \in \mathbb{Z}_N^{33\times 33}$ with entries

$$\mathbf{M}[i][j] = \binom{i}{j}\alpha^j\beta^{i-j} \quad \forall i,j : 0 \leq j \leq i \leq 32,$$

and $\mathbf{M}[i][j] = 0$ otherwise:

$$\mathbf{M} = \begin{pmatrix} 1 & & & & & \\ \beta & \alpha & & & & \\ \beta^2 & 2\alpha\beta & \alpha^2 & & & \\ \beta^3 & 3\alpha\beta^2 & 3\alpha^2\beta & \alpha^3 & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ \beta^{32} & 32\alpha\beta^{31} & \binom{32}{2}\alpha^2\beta^{30} & \binom{32}{3}\alpha^3\beta^{29} & \ldots & \alpha^{32} \end{pmatrix}.$$

Note that we omit the zeros from the depiction of $\mathbf{M}$ for ease of presentation.

Let us define the vector $\mathbf{c} = (1, c, c^2, \ldots, c^{32})$ made up of powers of the ciphertext c. Then by definition it follows that

$$\mathbf{M} \cdot \mathbf{c} = \mathbf{s}.$$

The following theorem shows that Algorithm 5 indeed computes the obfuscated vector $\hat{\mathbf{c}}$ if and only if $\mathbf{AM}$ directly reveals the secret permutation π as well as the randomization vector $\mathbf{r}$.

Theorem 1. *Let $\mathbf{e}_i \in \{0,1\}^{32}$ be the i^{th} unit vector. We have $\mathbf{As} + \mathbf{t} = \hat{\mathbf{c}} = (r_{\pi(0)} \cdot c^{\pi(0)}, \ldots, r_{\pi(31)} \cdot c^{\pi(31)})$ if and only if*

$$\mathbf{AM} = \begin{pmatrix} r_{\pi(0)}\mathbf{e}_{\pi(\mathbf{0})} & t_0 \\ r_{\pi(1)}\mathbf{e}_{\pi(\mathbf{1})} & t_1 \\ \vdots & \vdots \\ r_{\pi(31)}\mathbf{e}_{\pi(\mathbf{31})} & t_{31} \end{pmatrix}.$$

In other words, for $0 \leq i < 32$ the i^{th} row vector of $\mathbf{AM}$ is entirely made up of 0's, except for the last and the $\pi(i)^{th}$ position, at which we get entries t_i and $r_{\pi(i)}$ respectively.

Proof. Let $\mathbf{a}_i$ be the i^{th} row of $\mathbf{A}$. We then have to show that if for all $0 \leq i \leq 33$ we have $\mathbf{a}_i \cdot \mathbf{s} + \mathbf{t}[i] = r_{\pi(i)} c^{\pi(i)}$ it follows that

$$\mathbf{a}_i \cdot \mathbf{M} = (r_{\pi(i)} \mathbf{e}_{\pi(\mathbf{i})},\ t_i), \tag{5}$$

and vice versa. Hence, assume that $\mathbf{a}_i \cdot \mathbf{s} + \mathbf{t}[i] = r_{\pi(i)} c^{\pi(i)}$. Since we have $\mathbf{s} = \mathbf{M} \cdot \mathbf{c}$ and $\mathbf{t}[i] = -t_i c^{32}$, we obtain

$$\mathbf{a}_i \cdot \mathbf{M} \cdot \mathbf{c} = r_{\pi(i)} c^{\pi(i)} + t_i c^{32} = (r_{\pi(i)} \mathbf{e}_{\pi(\mathbf{i})},\ t_i) \cdot \mathbf{c}. \tag{6}$$

As the identity from Equation (6) has to hold independent of the ciphertext c and thus for all $\mathbf{c}$, this already implies the desired identity from Equation (5).

For the opposite direction, assume that Equation (5) holds. This immediately implies Equation (6), from which we conclude that $\mathbf{a}_i \cdot \mathbf{s} + \mathbf{t}[i] = r_{\pi(i)} c^{\pi(i)}$. □

E. Private Key Extraction

In the following, we propose two attack strategies to recover the secret permutation π from the white-box cryptographic algorithm. Hence, we show that the recovered white-box algorithm does not hold up to the notion of *security against key extraction* as outlined in Section II-D.

Chosen Ciphertext Attack. Extracting the key via a chosen ciphertext attack is straightforward once the ciphertext-dependency has been removed as outlined towards the end of Section IX-B. Given the modified white-box algorithm, we first run Obfusc-Precomp(1) from Algorithm 5 using ciphertext $c = 1$ as input. By the correctness property of Algorithm 5, we obtain the output

$$\pi(\mathbf{r}) = (r_{\pi(0)}, \ldots, r_{\pi(31)}).$$

Next, we run Obfusc-Precomp(2), resulting in output

$$(r_{\pi(0)} 2^{\pi(0)}, \ldots, r_{\pi(31)} 2^{\pi(31)}).$$

After dividing each element of the second output by the respective r_i, we receive vector $\mathbf{v} = (2^{\pi(0)}, \ldots, 2^{\pi(31)}) \in \mathbb{Z}_N^{32}$ comprising powers of 2. Since $\pi(i) \leq 31$ and $2^{31} < N$, no entry is reduced modulo N such that $\mathbf{v} \in \mathbb{Z}^{32}$. Therefore, computing the $\log_2$ of $\mathbf{v}$ element-wise yields the values of $\pi(0), \ldots, \pi(31)$ and thus discloses the secret permutation π.

Using Matrix A. We now present a second attack that only requires access to the hard-coded values $\mathbf{A}$, α, β defining the two matrices $\mathbf{A}$ and $\mathbf{M}$. We can directly read off $\mathbf{A}$ from Algorithm 5. Using Theorem 1, we can subsequently compute

$$\mathbf{AM} = \begin{pmatrix} r_{\pi(0)} \mathbf{e}_{\pi(\mathbf{0})} & t_0 \\ \vdots & \vdots \\ r_{\pi(31)} \mathbf{e}_{\pi(\mathbf{31})} & t_{31} \end{pmatrix},$$

which reveals the permuted randomization $\pi(\mathbf{r}) = (r_{\pi(0)}, \ldots, r_{\pi(31)})$ and the secret permutation π.

Private Key Recovery. Obfusc-Precomp(c) computes $\hat{\mathbf{c}}$ on ciphertext c as input. Recall from Equation (4) that the RSA decryption can be computed on obfuscated vector $\hat{\mathbf{c}}$ as

$$m = c^d = \text{Exp}(\hat{\mathbf{c}}, \hat{\mathbf{d}}) \cdot \prod_{i=0}^{k} r_{d_i}^{-32^i},$$

using the obfuscated secret key $\hat{\mathbf{d}} = (\pi^{-1}(d_0), \ldots, \pi^{-1}(d_k))$. Consequently, we can extract $\hat{\mathbf{d}}$ from $\text{Exp}(\hat{\mathbf{c}}, \hat{\mathbf{d}})$ and recover the original RSA secret key by computing

$$\pi(\hat{\mathbf{d}}) = \mathbf{d} = (d_0, \ldots, d_k) \text{ and } d = \sum_{i=0}^{k} d_i \cdot 32^i.$$

Given public key e and private key d, we retrieve the factorization $N = pq$ using the approach described in Section II-C.

X. Discussion

In the following, we discuss implications of our industry-wide security analysis, reflect on potential defenses, elaborate on existing literature, and finally outline future research directions.

A. Implications

We now discuss implications of our attacks from different perspectives.

EDA Tools. Although with varying complexity, each case study results in a full break of the confidentiality and integrity of the respective IEEE 1735 implementation. In particular, we extract all RSA private keys used within the analyzed EDA tools for the decryption of protected IP. As explicitly permitted by IEEE 1735, three out of the six analyzed vendors exhibit no (noteworthy) private key protections at all. Reverse engineering of these unprotected tools is sometimes even less time-consuming than their installation process. Since most IP is protected using at least one of the targeted vendors' RSA public keys, we can decrypt almost *all* protected IPs used throughout the industry. Consequently, the affected IP does not only present a target for IP theft, but is also susceptible to malicious manipulations such as hardware Trojans [5, 60].

IEEE Standard 1735-2014. Since the execution environment of EDA tools is typically untrusted, they are naturally susceptible to software reverse-engineering and manipulation. In this context, the recommendation of hard-coded vendor keys within IEEE 1735 is inherently insecure. Countering this critical threat by declaring it "*out-of-scope*" [31, p. 11] within IEEE 1735 and encouraging the prohibition of reverse engineering by adoption of end-user license agreements [31, p. 11], rather than providing guidance for a secured implementation, demonstrates a lack of a realistic risk assessment. Moreover, even if the IP decryption could be performed securely, the decrypted IP core would still be stored in Random-Access Memory (RAM) at runtime and could thus be extracted. In this regard, the standard mentions that "*Each tool vendor needs to decide what, if any, anti-debug, anti-tamper, and anti-key-discovery technologies they will use here.*" [31, p. 11]. However, the standard provides a false sense of security since in practice its soundness is entirely founded on the protection of the private keys and the plaintext IP. As in other DRM applications, *security-by-obscurity* constitutes the *only* available protection layer in this case. Additionally, an IP may contain session keys encrypted for multiple different tool vendors, therefore a single unprotected private key within one EDA tool can compromise the entire scheme. Hence, the central security assumptions of IEEE 1735 are invalid in practice. Future iterations of IEEE 1735 should at least *explicitly* emphasize the importance of software protection, outline limitations of cryptography in a real-world setting, and *clearly* communicate the threats arising thereof.

Public-Key White-Box Cryptography. While symmetric white-box cryptography has received significant attention in recent years [7, 9, 11, 18, 36], public-key white-box schemes have been analyzed scarcely in the open literature so far. However, (presumably insecure) proprietary public-key white-box schemes are already deployed in practice to protect high-value IP. In its current state and due to limited public scrutiny, we claim that public-key white-box cryptography lacks the security assurances required to be deployed on its own, i.e., without any additional software protections. Our case studies demonstrate proprietary cryptography to (again) be ineffective in practice. On the basis of our recovery of two commercial public-key white-box decryption schemes, we demonstrated its weaknesses in regard to both central security notions of white-box cryptography.

B. Thoughts on IP Protection in Practice

We now reflect on general implications and the potential mitigation of threats arising from untrusted execution environments used for IP protection schemes.

Integrity and Authentication. Some use-cases demand integrity and authentication of the IP rather than confidentiality, as most protected IP implement well-known interfaces or algorithms that typically do not provide a significant edge over competitor product portfolios. This is especially true for security-sensitive and safety-critical applications, f.i., in military or critical infrastructure systems. In these use-cases, the ability to inspect third-party IP before integration may even be required. Such applications primarily benefit from tamper-resistance (e.g., enforced by integrity protections) and authenticity of the IP vendor to prevent supply chain attacks.

Integrity and authentication of an IP can be achieved using digital signatures in combination with a Public Key Infrastructure (PKI) or (in offline-settings) by manual comparison of hash-values computed over the encrypted IP. Since integrity and authenticity are primarily in the interest of the IP user, there is typically no incentive for them to bypass these protections on their machines. Nonetheless, we want to emphasize that this cannot defend against software manipulations of the EDA tool that would enable an attacker to interfere with the IP after integrity checks have been performed. As a result, cryptographic integrity protections can defend against Man-in-the-Middle (MITM) attackers trying to intercept and manipulate the IP during shipment, but still fail to prevent attacks within the MATE attacker model.

Confidentiality. Introducing a PKI for the management and distribution of individual session keys at first seems to be an obvious choice. However, the private keys will still end up in the EDA tool's program memory and are therefore vulnerable to extraction. Hence, for the sake of our argument, let us assume that a perfectly secure key storage solution exists and all cryptographic operations can be performed in a trusted execution environment (e.g., Intel SGX or ARM TrustZone). Following our attacker model, a reverse engineer would still be able to tamper with the design software itself and read out its memory at runtime. As soon as the EDA tool performs any kind of operation on the protected IP core (e.g., run a synthesis or implementation step), it must decrypt the IP and keep its plaintext representation in memory during the entire operation. At this point, an attacker can recover the decrypted IP by dumping and inspecting the memory during operation. To counter this threat, the *entire* EDA tool would have to run in a trusted execution environment, which is not desirable for performance reasons.

Even if all EDA tool computations were to be performed in such a trusted environment, the result of the hardware design flow is either a bitstream file that configures an FPGA or a layout description for an ASIC. Crucially, both a bitstream [53, 56, 59] and a layout [47] contain all information about an IP core as they are simply a different gate-level netlist representation. A determined adversary can extract the high-level functionality of an IP core through netlist analysis [2, 39, 40, 52]. We note that even though this approach arguably requires considerable efforts and specialized knowledge on hardware reverse engineering, it always succeeds given a motivated adversary. Note that this is similar to other DRM settings, as the protected data must be handed to the user eventually.

In summary, perfect confidentiality of protected IP cannot be achieved, even when using strong cryptographic solutions and tools that are running in a trusted execution environment. Thus so far, the only viable defenses are raise-the-bar countermeasures based on strong software defense mechanisms.

Raise-the-Bar Countermeasures. Similar to DRM use-cases, cryptography alone cannot resolve the aforementioned issues to achieve confidentiality, hence the goal must be to increase the economic effort such that the costs of an infringement outweigh any potential profit. In our case studies, potent software obfuscation proves to be the one countermeasure that significantly increases attack time, while most anti-debugging measures were defeated using publicly available tooling. Cryptography and even white-box implementations may play a vital part, but, in their current state must always be combined with strong software obfuscation techniques to suppress any attempt on static or dynamic analysis.

Another approach to accomplish confidentiality in such untrusted execution environments may be to outsource all computations that require access to a decrypted IP core to a trusted (cloud-)server and employ a *thin-client* solution. This mitigates the threat of IP disclosure and manipulation to some extend as the protected IP itself is never decrypted on the IP user workstation. While thin-client solutions may be a viable for some use-cases, others such as the hardware development for military or critical infrastructure applications require the entire design process to be performed on *air-gapped* machines located on manufacturer premises. Despite all efforts, the IP user (and potential attacker) still receives a bitstream or layout in the end, which may then be targeted by netlist reverse-engineering to defeat confidentiality and integrity.

C. Related Work

Our work relates to aspects of both hardware IP protection and white-box cryptography.

IP Protection. The threat of hardware IP infringement and the development of techniques to achieve IP protection is widely covered in the literature [12, 45]. While works commonly focus on hardware obfuscation, IEEE 1735 [31] presents the first approach to an industry-wide solution based on well-known cryptographic algorithms. In 2017, Chhotaray et al. [17] identify flaws in the choice of cryptographic primitives of IEEE 1735. They mount both a padding-oracle and a syntax-oracle attack on an EDA tool for FPGAs. Subsequently, they recover the plaintext IP and apply meaningful manipulations to insert a hardware Trojan. However, their attacks did not question the fundamental security assumptions of IEEE 1735. In response to findings of Chhotaray et al., the IEEE issued a statement asking IP owners and users to secure their supply chain and implement end-user agreements [32]. The standard itself remains unchanged to this day. Mirian et al. [44] discuss flaws within the implementation of Xilinx ISE, which was discontinued in 2012. They extract the gate-level netlist of an IP protected by IEEE 1735 by exploiting the insecure design flow of Xilinx ISE, but do not recover a high-level description.

White-Box Cryptography. White-box cryptography was first introduced by Chow et al. [18, 19] in 2002. In the last decades numerous efforts to formalize symmetric white-box security properties have been published [9, 24, 49]. For now, white-box cryptography resembles an arms race with many such propositions [18, 19, 36] being broken shortly after publication [7, 8, 11]. In 2017, 2019, and 2021, white-box competitions were held at CHES [21, 22, 23] to stimulate research of white-box implementations and their security. The first competition was subsequently evaluated by Bock et al. [10]. Apart from classical attacks, gray-box approaches that use side-channel information to extract information from the white-box present another critical threat [11, 27, 48]. In academia, white-box research is typically focused on symmetric encryption schemes. To the best of our knowledge, the work of Barthelemy [4] is the only (academic) publication on public-key white-box cryptography based on a novel lattice-based approach. The latest white-box competition at CHES [23] set out the target to develop a secure ECDSA white-box implementation. At the time of writing, each and every contestant has been broken within two days of publication.

D. Future Work

In light of our discussion in Section X-B, we argue that an all-encompassing solution for IP protection is out of reach given the contemporary hardware design process. Nonetheless, IEEE standard 1735-2014 is in dire need for a timely revision that not only takes its choice of cryptographic primitives into account, but pays respect to its foundational security assumptions in untrusted execution environments as well. Furthermore, additional precautions for integrity assurance of the *entire* IP are essential to restore trust in a future iteration of IEEE 1735. Hence, we provide an incentive for standardization bodies and researchers alike to develop sound solutions for the protection of hardware IP.

Our work practically demonstrates the discrepancy between commercially-available public-key white-box solutions and the limited understanding of public-key white-box security in academia. Hence, both academia and industry can build upon our work to take the presented cryptanalytic attacks into consideration and build the foundations for a secure usage of public-key white-box cryptography in the future. Moreover, we hope that our work sparks interest in developing (and analyzing) public-key white-box algorithms to improve the understanding of such schemes.

XI. Conclusion

IEEE standard 1735-2014 aims to provide recommended practices for the secure management of electronic design IP [31]. Our work presented structural weaknesses of IEEE 1735 that allow for extraction of the EDA tool vendors' private keys. Consequently, our attacks enable to decrypt, maliciously modify, and re-encrypt all allegedly protected IP. In particular, our case studies uncovered that three vendors use no software protection to safeguard their *hard-coded* private keys, two vendors employ code obfuscation and anti-debugging, and three vendors utilize a public-key white-box cryptographic approach. We analyzed the white-box schemes and presented cryptanalytical attacks on all three schemes to effectively extract their hidden keys within seconds.

Our insights on realistic software analysis capabilities in untrusted execution environments demand a rethinking of the use of third-party IP cores in its current form as both IP authors and IP users are at risk with limited prospect of improvement.

References

[1] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. "Man-At-The-End attacks: Analysis, taxonomy, human aspects, motivation and future directions". In: *Journal of Network and Computer Applications* 48 (2015), pp. 44–57.

[2] Nils Albartus, Max Hoffmann, Sebastian Temme, Leonid Azriel, and Christof Paar. "DANA Universal Dataflow Analysis for Gate-Level Netlist Reverse Engineering". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 4 (2020), pp. 309–336. URL: https://tches.iacr.org/index.php/TCHES/article/view/8685/8244.

[3] Estuardo Alpirez Bock, Chris Brzuska, Marc Fischlin, Christian Janson, and Wil Michiels. “Security Reductions for White-Box Key-Storage in Mobile Payments”. In: *26th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Vol. 12491. Springer, 2020, pp. 221–252. URL: https://eprint.iacr.org/2019/1014.pdf.

[4] Lucas Barthelemy. “Toward an Asymmetric White-Box Proposal”. In: *IACR Cryptology ePrint Archive* 2020 (2020), p. 893. URL: https://eprint.iacr.org/2020/893.pdf.

[5] Shivam Bhasin, Jean Luc Danger, Sylvain Guilley, Xuan Thuy Ngo, and Laurent Sauvage. “Hardware trojan horses in cryptographic IP cores”. In: *10th Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)* (2013), pp. 15–29. URL: https://eprint.iacr.org/2014/750.pdf.

[6] Swarup Bhunia, Sandip Ray, and Susmita Sur-Kolay. *Fundamentals of IP and SoC Security: Design, Verification, and Debug*. 1st. Springer, 2017.

[7] Olivier Billet and Henri Gilbert. “A traceable block cipher”. In: *9th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*. Vol. 2894. 2003, pp. 331–346. URL: https://www.iacr.org/cryptodb/archive/2003/ASIACRYPT/31/31.pdf.

[8] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. “Cryptanalysis of a White Box AES Implementation”. In: *11th International Workshop on Selected Areas in Cryptography (SAC)* 3357 (2004), pp. 227–240. URL: https://bo.blackowl.org/s/papers/waes.pdf.

[9] Estuardo Alpirez Bock, Alessandro Amadori, and Chris Brzuska. “On the Security Goals of White-Box Cryptography”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 2 (2020), pp. 327–357. URL: https://tches.iacr.org/index.php/TCHES/article/view/8554/8119.

[10] Estuardo Alpirez Bock and Alexander Treff. “Security Assessment of White-Box Design Submissions of the CHES 2017 CTF Challenge”. In: *IACR Cryptology ePrint Archive* 2020 (2020), p. 342. URL: https://eprint.iacr.org/2020/342.pdf.

[11] Joppe W. Bos, Charles Hubain, Wil Michiels, and Philippe Teuwen. “Differential computation analysis: Hiding your white-box designs is not enough”. In: *18th International Conference on Cryptographic Hardware and Embedded Systems (CHES)*. Springer, 2016, pp. 215–236. URL: https://www.iacr.org/archive/ches2016/98130106/98130106.pdf.

[12] Lilian Bossuet and Lionel Torres. *Foundations of Hardware IP Protection*. 1st ed. Springer, 2017.

[13] Cadence Design Systems. *Using the IEEE 1735 protection mechanism with a Public key to protect Verilog code or VHDL code, and how models can share between vendor tool sets, DECERR or CORRPD error*. Aug. 11, 2021.

[14] Carbon, cypher, mrexodia, and Mattiwatti. *ScyllaHide*. 2021. URL: https://github.com/x64dbg/ScyllaHide (visited on 12/09/2021).

[15] Ping Chen, Christophe Huygens, Lieven Desmet, and Wouter Joosen. “Advanced or not? A comparative study of the use of anti-debugging and anti-VM techniques in generic and targeted malware”. In: *IFIP International Conference on ICT Systems Security and Privacy Protection* 471 (2016), pp. 323–336. URL: https://core.ac.uk/download/pdf/34612995.pdf.

[16] Xu Chen, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. “Towards an Understanding of Anti-virtualization and Anti-debugging Behavior in Modern Malware”. In: *38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2008, pp. 177–186. URL: https://web.eecs.umich.edu/~zmao/Papers/DCCS-xu-chen.pdf.

[17] Animesh Chhotaray, Adib Nahiyan, Thomas Shrimpton, Domenic Forte, and Mark Tehranipoor. “Standardizing Bad Cryptographic Practice: A Teardown of the IEEE Standard for Protecting Electronic-design Intellectual Property”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017, pp. 1533–1546. URL: https://acmccs.github.io/papers/p1533-chhotarayA.pdf.

[18] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. Van Oorschot. “A white-box DES implementation for DRM applications”. In: *ACM CCS-9 Workshop on Security and Privacy in Digital Rights Management (DRM)*. Springer, 2002, pp. 1–15. URL: https://crypto.stanford.edu/DRM2002/whitebox.pdf.

[19] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. “White-box cryptography and an AES implementation”. In: *9th Annual International Conference on Selected Areas in Cryptography (SAC)*. 2595. Springer, 2002, pp. 250–270. URL: https://home.cs.colorado.edu/~jrblack/class/csci7000/f03/papers/oorschot-whitebox.pdf.

[20] Christian Collberg, Clark Thomborson, and Douglas Low. *A taxonomy of obfuscating transformations*. Tech. rep. 1997.

[21] Conference on Cryptographic Hardware and Embedded Systems (CHES). *Whib0x Contest 2017*.

[22] Conference on Cryptographic Hardware and Embedded Systems (CHES). *Whib0x Contest 2019*.

[23] Conference on Cryptographic Hardware and Embedded Systems (CHES). *Whib0x Contest 2021*.

[24] Cécile Delerablée, Tancrède Lepoint, Pascal Paillier, and Matthieu Rivain. “White-Box Security Notions for Symmetric Encryption Schemes”. In: *20th Annual International Conference on Selected Areas in Cryptography (SAC)*. Vol. 8282. Springer, 2013, pp. 247–264. URL: https://eprint.iacr.org/2013/523.pdf.

[25] Digital.ai. *White-Box Cryptography*. URL: https://digital.ai/glossary/whitebox-cryptography (visited on 12/09/2021).

[26] Dominic Distel. "Markets for Technology in the Semiconductor Industry – The Role of Ability-Related Trust in the Market for IP Cores". PhD thesis. Technical University of Munich, 2017. URL: https://mediatum.ub.tum.de/doc/1357158/1357158.pdf.

[27] Louis Goubin, Matthieu Rivain, and Junwei Wang. "Defeating State-of-the-Art White-Box Countermeasures with Advanced Gray-Box Attacks". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 3 (2020), pp. 454–482. URL: https://tches.iacr.org/index.php/TCHES/article/view/8597/8164.

[28] Jan Hoogerbrugge and Wil Michiels. "Protecting the Input/Output of Modular Encoded White-Box RSA". U.S. Patent 10,726,108 B2. 2020.

[29] Jan Hoogerbrugge and Wil Michiels. "White-Box Modular Exponentiation". U.S. Patent 10,235,506 B2. 2019.

[30] Jan Hoogerbrugge, Wil Michiels, and Pim Vullers. "White-Box Elliptic Curve Point Multiplication". U.S. Patent 10,068,070 B2. 2018.

[31] IEEE Design Automation Standards Committee (DASC). *IEEE 1735-2014 - Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP)*. IEEE, 2015.

[32] IEEE Design Automation Standards Committee (DASC). *Security Statement*. Tech. rep. IEEE, 2017. URL: http://standards.ieee.org/about/ieee_1735.pdf.

[33] Intel. *Intel Quartus Prime Pro Edition: Version 21.3 Software and Device Support Release Notes*. 2021. URL: https://www.intel.com/content/www/us/en/programmable/documentation/ewa1443722509979.html.

[34] Intel. *Introduction to Intel® FPGA IP Cores*. Version 2020.11.09. Sept. 11, 2020. URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_intro_to_megafunctions.pdf.

[35] Irdeto. *Whitebox Cryptography*. URL: https://irdeto.com/whitebox-cryptography/ (visited on 12/09/2021).

[36] Mohamed Karroumi. "Protecting white-box AES with dual ciphers". In: *13th International Conference on Information Security and Cryptology (ICISC)*. Vol. 6829. Springer, 2010, pp. 278–291.

[37] Stan Krolikoski. *Year End 2020 DASC Report*. Tech. rep. IEEE Design Automation Standards Committee (DASC), 2020. URL: https://www.dasc.org/DASC%20Report%2011.2020.pdf.

[38] Lattice Semiconductor. *Lattice Radiant Software 3.0 User Guide*. Version 3.0. June 14, 2021. URL: https://www.latticesemi.com/view_document?document_id=53229.

[39] Max Planck Institute for Security and Privacy. *HAL - The Hardware Analyzer*. 2019. URL: https://github.com/emsec/hal (visited on 12/09/2021).

[40] Travis Meade, Kaveh Shamsi, Thao Le, Jia Di, Shaojie Zhang, and Yier Jin. "The Old Frontier of Reverse Engineering: Netlist Partitioning". In: *Journal of Hardware and Systems Security* 2.3 (2018), pp. 201–213. URL: http://jin.ece.ufl.edu/papers/HASS2018_netlist.pdf.

[41] Wil Michiels and Paulus Gorissen. "Exponent Obfuscation". U.S. Patent 8,600,047 B2. 2013.

[42] Microchip Technology. *Secure IP Flow for IP Vendors and Libero SoC Users*. Version A. May 2021. URL: https://www.microsemi.com/document-portal/doc_download/133573-libero-soc-secure-ip-flow-user-guide.

[43] Gary L. Miller. "Riemann's Hypothesis and Tests for Primality". In: *Journal of Computer and System Sciences* 13.3 (1976), pp. 300–317. URL: https://dl.acm.org/doi/abs/10.1145/800116.803773.

[44] Vincent Mirian and Paul Chow. "Extracting designs of secure IPs using FPGA CAD tools". In: *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)* (2016), pp. 293–298.

[45] Prabhat Mishra, Swarup Bhunia, and Mark Tehranipoor. *Hardware IP Security and Trust*. 1st ed. Springer, 2017.

[46] Peter L. Montgomery. "Modular Multiplication Without Trial Division". In: *Mathematics of Computation* 44.170 (1985), p. 519. URL: https://www.ams.org/journals/mcom/1985-44-170/S0025-5718-1985-0777282-X/S0025-5718-1985-0777282-X.pdf.

[47] Rachel Selina Rajarathnam, Yibo Lin, Yier Jin, and David Z. Pan. "ReGDS: A Reverse Engineering Framework from GDSII to Gate-level Netlist". In: *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 2020, pp. 154–163. URL: https://ieeexplore.ieee.org/abstract/document/9300272.

[48] Eloi Sanfelix, Cristofaro Mune, and Job de Haas. "Unboxing the White-Box: Practical attacks against Obfuscated Ciphers". In: *Proceedings of the 2015 Black Hat Europe Conference* (2015). URL: https://www.blackhat.com/docs/eu-15/materials/eu-15-Sanfelix-Unboxing-The-White-Box-Practical-Attacks-Against-Obfuscated-Ciphers-wp.pdf.

[49] Amitabh Saxena, Brecht Wyseur, and Bart Preneel. "Towards security notions for white-box cryptography". In: *12th International Conference on Information Security (ISC)*. Vol. 5735. Springer, 2009, pp. 49–58. URL: https://www.esat.kuleuven.be/cosic/publications/article-1260.pdf.

[50] Siemens. *ModelSim® User's Manual*. Version v10.5c. URL: https://www.microsemi.com/document-portal/doc_download/136662-modelsim-me-10-5c-user-u-s-manual-for-libero-soc-v11-8.

[51] Siemens. *SSA-400332: Insufficient Design IP Protection in Questa and ModelSim*. (available latest on 12/14/2021). 2021. URL: https://cert-portal.siemens.com/productcert/pdf/ssa-400332.pdf.

[52] Pramod Subramanyan, Nestan Tsiskaridze, Wenchao Li, Adrià Gascón, Wei Yang Tan, Ashish Tiwari, Natarajan

Shankar, Sanjit A. Seshia, and Sharad Malik. “Reverse engineering digital circuits using structural and functional analyses”. In: *IEEE Transactions on Emerging Topics in Computing* 2.1 (2014), pp. 63–80. URL: https://sites.cs.ucsb.edu/~nestan/pdf/TETCSI.pdf.

[53] SymbiFlow Team. *Project X-Ray*. 2021. URL: https://github.com/SymbiFlow/prjxray (visited on 12/09/2021).

[54] Thales. *White Box Cryptography*. URL: https://cpl.thalesgroup.com/software-monetization/white-box-cryptography (visited on 12/09/2021).

[55] tomer8007. *Widewine L3 Decryptor*. 2020. URL: https://github.com/tomer8007/widevine-l3-decryptor (visited on 12/09/2021).

[56] Claire Wolf and Mathias Lasser. *Project IceStorm*. 2018. URL: http://bygone.clairexen.net/icestorm/ (visited on 12/09/2021).

[57] Xilinx. *Findings on IEEE-1735-2014 recommended practice from Max Planck Institute for Security and Privacy (MPI-SP) and Ruhr University Bochum (RUB)*. 2021. URL: https://support.xilinx.com/s/article/000033507.

[58] Xilinx. *Vivado Design Suite User Guide – Creating and Packaging Custom IP (UG1118)*. Version v2021.1. June 30, 2021. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug1118-vivado-creating-packaging-custom-ip.pdf.

[59] Hoyoung Yu, Hansol Lee, Sangil Lee, Youngmin Kim, and Hyung Min Lee. “Recent advances in FPGA reverse engineering”. In: *Electronics* 7.10 (2018), pp. 1–14. URL: https://www.mdpi.com/2079-9292/7/10/246.

[60] Xuehui Zhang and Mohammad Tehranipoor. “Case study: Detecting hardware Trojans in third-party digital IP cores”. In: *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)* (2011), pp. 67–70.

[61] Yongxin Zhou and Stanley T. Chow. “System and Method of Hiding Cryptographic Private Keys”. U.S. Patent 7,634,091 B2. 2009.

APPENDIX
IEEE AND VENDOR RESPONSES

The vendors covered by our case studies have been cooperative throughout the entire responsible disclosure process and (at the point of writing) have either updated their tools already or are working towards fixing the disclosed vulnerabilities. Moreover, several affected vendors publicly acknowledge our findings in coordination with our publication [33, 51, 57] and guide users through the necessary steps to improve the protection of their IP. So far, the IEEE has declined multiple offers to participate in discussions on our attacks and the future directions of IEEE 1735. We note that within their correspondence, the IEEE insisted on referring to IEEE 1735 as a “recommended practice” instead of a “standard”.