Standardizing Bad Cryptographic Practice

A teardown of the IEEE P1735 standard for protecting electronic-design intellectual property

Animesh Chhotaray
University of Florida

Adib Nahiyan
University of Florida

Thomas Shrimpton
University of Florida

Domenic Forte
University of Florida

Mark Tehranipoor
University of Florida

ABSTRACT

We provide an analysis of IEEE standard P1735, which describes methods for encrypting electronic-design intellectual property (IP), as well as the management of access rights for such IP. We find a surprising number of cryptographic mistakes in the standard. In the most egregious cases, these mistakes enable attack vectors that allow us to recover the entire underlying plaintext IP. Some of these attack vectors are well-known, e.g. padding-oracle attacks. Others are new, and are made possible by the need to support the typical uses of the underlying IP; in particular, the need for commercial system-on-chip (SoC) tools to synthesize multiple pieces of IP into a fully specified chip design and to provide syntax errors. We exploit these mistakes in a variety of ways, leveraging a commercial SoC tool as a black-box oracle.

In addition to being able to recover entire plaintext IP, we show how to produce standard-compliant ciphertexts of IP that have been modified to include targeted hardware Trojans. For example, IP that correctly implements the AES block cipher on all but one (arbitrary) plaintext that induces the block cipher to return the secret key.

We outline a number of other attacks that the standard allows, including on the cryptographic mechanism for IP licensing. Unfortunately, we show that obvious "quick fixes" to the standard (and the tools that support it) do not stop all of our attacks. This suggests that the standard requires a significant overhaul, and that IP-authors using P1735 encryption should consider themselves at risk.

CCS CONCEPTS

• **Security and privacy** → *Digital rights management*; *Hardware security implementation*; • **Hardware** → *Best practices for EDA*;

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134040

KEYWORDS

syntax oracle attack, padding oracle attack, IP encryption, IP piracy, hardware Trojan, P1735

1 INTRODUCTION

A System on Chip (SoC) is a single integrated circuit that incorporates all of the digital and analog components necessary to implement a target system architecture, e.g., a radio-frequency receiver, an analog-to-digital converter, network interfaces, a digital signal processing unit, a graphics processing unit, one or more central processing units, a cryptographic engine, memory, and so on. The vast majority of mobile and handheld devices contain a SoC, as do many embedded devices. The complexity and cost of modern SoC processors, amplified by time-to-market pressure, makes it infeasible for a single design house to complete an entire SoC without outside support. Instead, they procure electronic design intellectual property (IP) for various SoC components and integrate them with their own in-house IP. An IP is a collection of reusable design specifications that may include — a chip layout, a netlist, a set of fabrication instructions, etc [13]. These IP cores are intellectual property of one party, and could be licensed to other parties as well. A modern SoC can include tens of IPs from different vendors distributed across the globe. This approach to SoC design has become the norm for a large portion of the global IP market.

The current semiconductor IP market is valued at $3.306 billion, and is estimated to reach $6.45 billion by 2022 [30] with the emergence of IoT devices. Thus, IP developers have a clear economic incentive to protect their products and their reputations. Profit is lost if the IP is used by parties who have not paid for it, if it divulges trade secrets, or if it is used to produce so-called "clone" chips. Company reputations are damaged if the IP does not perform as advertised. And if security critical design features are leaked, or backdoors uncovered by users, the damage can be long-lasting.

In order to protect confidentiality of IP and provide a common mark-up syntax for IP design that is interoperable across different electronic design and automation (EDA) tools and hardware flows, the IEEE SA-Standards Board developed the P1735 standard [13]. This standard has been adopted by Synopsys, Xilinx, and other leaders of the semiconductor IP industry.

IEEE P1735 is broken (and potentially dangerous). We expose a number of cryptographic mistakes in the P1735

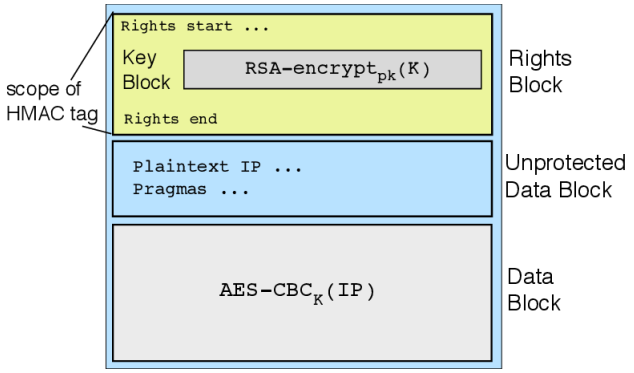

Figure 1: A P1735 ver. 2 *Digital Envelope*. The Rights Block contains the RSA-encryption of an AES key, which is used to encrypt the sensitive portions of the IP with AES-CBC mode. Note that only the Rights Block is covered by the authentication mechanism.

standard, be they explicit mistakes, mistakes of omission, or failure to address important attack vectors. We show that commercial EDA tools that comply with this standard can actually enable attacks that allow *full recovery of the plaintext IP* without the key. We also demonstrate that, given the encryption of an IP, we can forge a standard-compliant encryption of that IP modified to *contain targeted hardware trojans of our choosing*. For example, we turn encrypted IP for an AES implementation into one that can be induced to leak its secret key. This ability to insert HW trojans results from the fact that, despite surface appearance to the contrary, the cryptographic methods standardized in IEEE P1735 provide no integrity protections whatsoever to the encrypted IP.

We use the Synopsys Synplify Premier EDA tool (Version L-2016.09) to make our attacks concrete and to analyze their performance. Synopsys is one of the main EDA tool vendors, with a market share of 37% [1]. To be clear, we are not finding fault with the tool: it is the standard that bears the blame.

Let us give a very brief summary of what P1735 gets wrong, from a cryptographic perspective, and how we exploit these mistakes.

No confidentiality protection. Figure 1 gives a slightly simplified view of the P1735 "digital envelope." It implements a kind of hybrid public-key encryption scheme: it transports an AES key K that is encrypted under the RSA public-key of the EDA tool, and then the sensitive portion of the IP is encrypted using AES (under key K) in CBC-mode. While the Data Block contains the AES encrypted IP, the Key Block holds the encryption of the AES key. We stress that CBC-mode is the only symmetric-key encryption scheme discussed in the standard.[1]

First of all, the P1735 standard provides no guidance as to how plaintexts should be padded prior to CBC-mode encryption. Thus, tools wishing to support P1735 are left to make a choice that is known to be security critical [29, 35, 37]. As an example, the Synopsys Synplify Premier tool implements the commonly used PKCS#7 scheme; it also reports a distinguishable padding error upon decryption. The combination of these leads to well-known padding-oracle attacks (POA), which we exploit to recover full plaintexts without knowledge of the key.

An informed "quick fix" to stop the padding-oracle attack might be to employ a different padding scheme, e.g. OZ or AByte padding [9, 28]. Or to switch from CBC-mode to a block-cipher mode that requires no padding, e.g., counter-mode (CTR). But none of these would stop our new *syntax-oracle attack* (SOA) from recovering plaintext. In this attack, we exploit the fact that EDA tools may provide a wealth of observable syntax-error messages, once the encrypted IP has been decrypted and the tool begins to process the plaintext. Indeed, the standard recommends this:

> "all tools do error checking and report errors and warnings to the user. The quality of those error messages reflects on the usability of those tools, and by extension, the quality of protected IP." [13, Section 10]

Moreover, the standard suggests that such error messages are not useful to attackers:

> "... for IP of more than trivial complexity, it is highly unlikely that information in error messages will fundamentally compromise the IP and allow essential information to be stolen. Therefore, there is a good argument that protected IP is more usable with error messages that are transparent and the risk of loss of value will be little to none." [13, Section 10]

Our SOA attack shows this thinking is entirely wrong-headed.

No integrity protection. In addition to providing no actual confidentiality guarantees to the underlying IP, a P1735 digital envelope provides no integrity protection. To be fair, the standard does not call out integrity protection of the digital envelope (or even the IP) as an explicit goal. One might even argue that there is no need for integrity protection. After all, the standard states that the EDA tool is assumed to be trusted, and there is no incentive for an *honest* IP user to maul the digital envelope it receives from the IP author. Our position is that this viewpoint is too narrow. Rogue entities do exist in the modern SoC design-flow, and the existence of P1735 is evidence that the semiconductor industry acknowledges the real (and costly) threat they represent.

To highlight the danger of not addressing integrity protection, we give an attack that succeeds to embed targeted hardware-Trojans into an IP that is encrypted via a P1735 digital envelope. In fact, the standard admits such an attack trivially, because the creator of the digital envelope selects the AES key K, and the standard provides no mechanism for authenticating the party who selected it. But our attack

[1] The standard allows for DES- and 3DES-based CBC-mode (although these are deprecated), and requires that AES128- and AES256-based CBC-mode be supported.

works *even if the key K is unknown and bound to the IP author.*

Broken licensing-proxy mechanism. The standard also includes a mechanism for EDA tools to communicate with an IP-author-provided licensing proxy. Loosely, the tool sends an AES-CBC encrypted "license request" message on behalf of the user, and the proxy responds with an AES-CBC encrypted "license granted" or "license denied" message. Although we did not have available a commercial tool that implements this protocol, P1735 appears to admit multiple attacks on it. Here, the culprits are the use of the same initialization vector (IV) for all messages sent within a single connection (and there may be multiple license requests and responses within a connection), and the fact that the "license granted" and "license denied" messages both echo the "license request" message.

There are a number of other cryptographic errors that are not as obviously damaging, and numerous places where the standard is vague or silent on security critical matters. A broader summary is found in Appendix A.

Summary of our contributions. At a high level, our work makes contributions along multiple dimensions. First, it analyzes an international standard that has been adopted by major commercial EDA tools and is likely to impact the development of future tools. Second, while our attacks are not technically deep from a cryptographic perspective, they demonstrate that complying with the standard provides no real security. We optimize these attacks to make them quite efficient, especially when one considers the amount of time (and money) that IP developers expend to develop their products. Third, we bring to the attention of the security community a facet of the supply-chain attack surface that is badly in need of principled protections. We hope our work will encourage others to examine standards that aim to protect other pieces of this surface.

Concretely, the main results of this paper are:

- Two attacks (POA and SOA) that extract the plaintext from standard-compliant ciphertexts without knowledge of the key. We also provide optimizations suitable for both attacks that reduce their complexity from a naive $O(N^2)$ to $O(N)$, where N is the number of ciphertext blocks.
- Application of the POA and SOA attacks on nine IP benchmarks of various sizes and content. We quantitatively compare them according to their execution time and accuracy.
- Two integrity-violating attacks that require only partial knowledge of the IP plaintext; this can be gained using POA, SOA or other attacks that may yet be discovered. We show how to insert a targeted hardware Trojan into any IP without knowledge of the key.
- Analysis of potential vulnerabilities in the licensing scheme described by the standard, which can result in unauthorized access and denial of service.

We also provide recommendations for addressing the mistakes we identify and exploit. 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 and produce the Data Block, treating everything that is not the sensitive IP (but still is to be transmitted) as the associated data (AD). For example, the standard could mandate CTR-mode encryption of the IP for the Data Block, with an attached HMAC whose scope covers everything to be included in the digital envelope. That is, use CTR-mode encryption and HMAC in an "encrypt-then-MAC" style of generic composition [8],[27], appropriately modified to admit AD. Using CTR-mode removes concerns about padding (hence padding-oracle attacks), and using encrypt-then-MAC style AEAD prevents (in theory) any sort of syntax-oracle attack because digital envelopes would be rejected as invalid before any plaintext from the Data Block was released for further processing.

However, we note that this conceptually straightforward change would require substantial changes in the standard, and the EDA tools that support it. Minimally, the IEEE would need to: deprecate previous versions of the P1735 standard immediately with no support for backward compatibility, define standard specific variables (or "pragmas") for an AEAD scheme, define the revamped mark-up format of the digital envelope, explicitly define the behavior of the tool when decryption fails (due to any reason), and create a set of standard error messages that the tool can output during processing of the digital envelope (e.g., a version error to identify digital envelopes complying with a previous version of the P1735 standard.) Likewise, EDA tool providers would need to: identify deprecated versions of the standard and report version error, add new APIs that the IP authors could use to create the digital envelope using the standardized AEAD scheme, add error flags/messages in its compiler to catch errors due to the AEAD scheme, and avoid conflating cryptographic error messages with Verilog/VHDL error messages.

2 BACKGROUND

2.1 SoC Design Flow

Figure 2 shows a typical SoC design. In the first step, the SoC integrator (design house) specifies the high-level requirements and blocks of the SoC. It then identifies a list of IPs necessary to implement the given specification. These "IP cores" are either developed in-house or purchased from third party IP (3PIP) developers. In the latter case, the cores may be of the following forms:

- "Soft" IP cores are delivered as synthesizable register transfer level (RTL) specifications written in a high-level hardware description language (HDL) such as Verilog or VHDL. These IP cores are human-readable by virtue of being written in a high-level language.
- "Firm" IP cores are delivered as gate-level implementations of the IP, i.e., sets of registers and logic gates

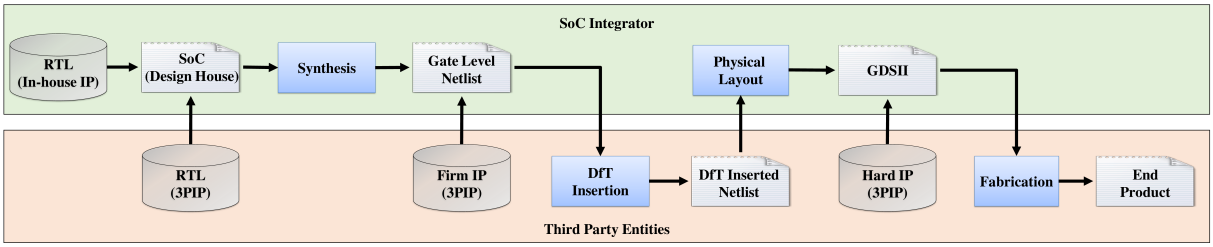

Figure 2: System-on-chip (SoC) design flow.

connected by wires. They are often visualized as gate-level schematics or human-readable netlists, but do not expose the underlying IP. Reverse engineering the RTL specification (even approximately) from the gate-level implementation is considered a non-trivial problem, akin to recovering source code from machine code.

- "Hard" IP cores are delivered as GDSII representations of a design, i.e., a set of planar geometric shapes representing transistors and interconnects. These are human readable (with some effort), and are easily converted to gate-level implementations. Like firm IP, it is non-trivial to recover the original RTL from which it was generated (if any).

Soft IPs provide greater flexibility and enable easier integration with other IPs in the SoC. Therefore, soft IP is the most common form of 3PIP by a large margin [38]. After developing/procuring all the necessary soft IPs, the SoC design house integrates them to generate the RTL specification of the whole SoC. The RTL design goes through extensive functional/behavioral testing to verify the functional correctness of the SoC and also to identify bugs. The SoC integrator then synthesizes the RTL description into a gate-level netlist based on a target technology library. (They may also integrate firm IP cores from a vendor into this netlist.) The gate-level netlist then goes through formal equivalence checking to verify that the netlist is equivalent to the RTL representation.

Next, specific design-for-test (DFT) and design-for-debug (DFD) structures are integrated into the netlist. As the names suggest, these make it easier to test and debug a SoC design later on in the fabrication process. (We note that DFT and DFD structures may be integrated into the netlist in-house, or by third parties, further complicating the security surface.) The DFT inserted netlist then goes through static timing analysis to analyze if the implemented design conforms to the timing requirement.

After this, the gate-level netlist is translated into a physical-layout design. At this stage, it is also possible to import and integrate hard IP cores from vendors. After performing static timing analysis and power closure, the SoC integrator generates the final layout in GDSII format and sends it out to the foundry for fabrication.

The flow discussed above is for application-specific integrated-circuit (ASIC) designs. An SoC can also be implemented in a field-programmable gate array (FPGA). The FPGA design flow is similar to ASIC flow until synthesis. After the synthesis in FPGA flow, the design goes through "place-and-route" process for the targeted FPGA chip and a bit-stream is generated which implements the design on FPGA.

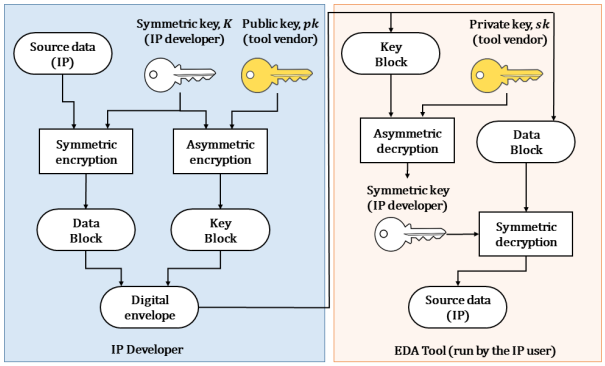

Figure 3: Work flow of the P1735 standard.

In the SoC design flow, for either ASIC or FPGA, the P1735 standard is mainly used by developers of soft and firm IP-cores, who wish to keep their technology confidential. This standard is also used by SoC designers who want to ensure that the design is not tampered by rogue employees (i.e., insider attack) or by third party entities present in the SoC design flow.

2.2 IEEE P1735 Standard

The IEEE SA-Standards Board developed the P1735 standard to provide guidance on protection of electronic design intellectual property (IP) [13]. It defines three stakeholders: IP author, IP user, and tool vendor. The *IP author* is the producer and legal owner of the IP. The *IP user* is the party who will use the IP author(s) product(s) to develop its SoC. The *tool vendor* provides an EDA tool to the IP user. The tool should simultaneously enable the IP user to develop its SoC, and protect the rights of the IP author. Note that the EDA tool is run on a platform that the IP user controls.

From an economic perspective, the IP author and IP user have competing interests. The former wants to maximize the return on its (often signficant) research and developement investment; the latter wants to use various pieces of IP at

minimal cost. The P1735 standard effectively adopts the viewpoint that the IP user is the adversary. A malicious IP user would like to recover the plaintext IP, and possibly find and exploit holes in the access control mechanism. The EDA tool is considered to be trusted, and is thus permitted by the IP author to carry out decryption. Also, it is the EDA tool that provides code for IP encryption to the IP author, and this code is trusted. The working assumption is that the EDA tool will not leak to the IP user anything beyond what the IP author deems acceptable, this being specified in the Rights Block of the protected IP.

The P1735 standard provides recommended practices for using encryption in order to ensure confidentiality of IP. To support interoperability and broad adoption, it also specifies a common mark-up format to represent an encrypted IP. The mark-up format uses standard-specific variables, or *pragmas*, to identify and encapsulate different portions of the protected IP. It also uses these pragmas to specify the encryption algorithms, digest algorithms, etc.

The standard also provides mechanisms to support rights management and licensing; together these enable IP authors to assert fine-grained access control. With the rights management functionality, an IP author can assert which output signals are accessible to the IP user when the EDA tool simulates the IP for the latters benefit. The licensing functionality allows access to authorized users only, e.g., companies that have paid for the rights to use the IP.

The basic work flow of the standard is shown in Figure 3. The standard mandates AES-CBC (but allows for other blockciphers) and RSA (≥ 2048) for symmetric and asymmetric encryption, respectively. For AES it recommends a key size of 128 or 256. We note that while the tool may perform simulation, synthesis, and other processes on the IP, it never reveals the IP in its plaintext format to the IP user [13].

2.3 Hardware Trojans

Due to the globalization of the semiconductor design and fabrication process, SoCs are increasingly becoming vulnerable to malicious modifications often referred to as hardware Trojans [16] [25]. These hardware Trojans can create backdoors in the design, through which sensitive information can be leaked, and other possible attacks (e.g., denial of service, reduction in reliability, etc.) can be performed.

The basic structure of a hardware Trojan consists of two main parts: trigger and payload. A Trojan trigger is an optional part that monitors various signals and/or a series of events in the SoC. Once the trigger detects an expected event or condition, the payload is activated to perform a malicious behavior. Typically, the trigger is expected to be activated under extremely rare conditions, so the payload remains inactive for most of the time. When the payload is inactive, the SoC acts like a Trojan-free circuit, making it difficult to detect the Trojan [26]. A Trojan can have a variety of possible payloads. In this paper, we will focus on payloads which leak secret information [34].

3 CONFIDENTIALITY ATTACKS

In general, IP authors price in a *risk premium* to compensate for the risk of revenue loss should their IP be used in an unauthorized manner. The P1735 standard aims to mitigate this risk, and to establish trust in the semiconductor IP market, by mandating cryptographic mechanisms meant to provide confidentiality (at least) for IP. Reducing the risk should reduce the cost of the IPs; increasing trust should enable IP authors to engage in transactions with more prospective IP users. To this end, the standard states [13, Section 4.3]

> "in its encrypted form, and in the absence of the decryption key, the data is secure both in transmission and at rest in a file ... There are no independent means to decrypt and access it at the IP user premises"

but we show that this claim is completely false.

We present two different attacks to break the confidentiality of an encrypted IP. The first is a standard padding-oracle attack (POA), and the other is a new, related, syntax-oracle attack (SOA). These attacks extract the plaintext of an encrypted IP without the knowledge of the key. (Readers who are very familiar with padding-oracle attacks may wish to skip directly to the syntax-oracle attack in Section 3.2.) Moreover, in Section 4 we show that once the confidentiality of the IP is broken, the adversary can insert any targeted hardware Trojan into the original IP ciphertext.

3.1 Padding-Oracle Attack

The P1735 standard mandates CBC-mode for symmetric encryption. CBC-mode operates on strings whose length is a multiple of the blocksizse of blockcipher being used, e.g. 128-bits when using AES-CBC as recommended by the standard. Therefore, one must attend to padding of plaintexts to make them block-aligned. The standard makes no recommendation for any specific padding scheme, leaving the tool vendors to decide what to do. (Recall that the EDA tool provides code for encryption of IP intended for use with that tool.)

The Synplify Premier tool supports PKCS#7 padding. In this scheme, if the last block of plaintext is block-aligned, a new block is added and filled with the padding byte (PB) which is equal to the block-size in bytes. Otherwise, the last block is padded with PB till the block gets full. In this case, PB is equal to the difference of block size in bytes and the number of bytes in the last block. For example, if the last block is short by 2 bytes, it is padded with 0x02 0x02. During decryption, if the last plaintext block has incorrect padding, a padding error is reported.

In the classic padding-oracle attack [35], Vaudenay used this error as an oracle (**PAD**) to recover the plaintext (P) without knowing the key. In this attack, when the oracle is given a ciphertext (C) as input, it returns 1 if there is a padding error, and 0 otherwise. Suppose the target ciphertext is $C = IV \,\|\, C_1 \,\|\, C_2 \,\|\, C_3$, where IV is the initialization vector, and all blocks are 16 bytes long. Letting $C_j[i]$ and $P_j[i]$ denote the i^{th} byte in the j^{th} block of the ciphertext and plaintext, respectively, the attack proceeds as follows. The

adversary starts guessing bytes in the last block (C_3) in the reverse order, i.e., she first guesses the 16^{th} byte. Let the guess byte be g. She xors $C_2[16]$ with the guess byte and padding byte, PB (= 0x01), i.e $C_2'[16] = C_2[16] \oplus g \oplus 0x01$, where C_2' is modified C_2. She concatenates the ciphertext blocks and IV as shown earlier, and queries the padding oracle. If the oracle returns 0 (no padding error), she repeats the process with a new guess byte. When 1 is returned, she stops, initializes $P_2[16]$ with the value of g, and xors $C_2[16]$ with 0x01 to remove the padding. The adversary then repeats the process for the 15^{th} byte, with pad as 0x02. Note that she has to xor $C_2[16]$ with 0x02, so that the last two bytes in P_3 become 0x02 0x02 (valid padding) when the adversary correctly guesses $P_2[15]$. She repeats this process to guess all the bytes in C_3. Then, she truncates the last block to make C_2 as the current target block, and repeats the attack to recover plaintext from C_2.

In the case of AES and a plaintext alphabet of ASCII bytes, the attack takes a maximum of $256 \times 16 \times N$ attempts to find all of the plaintext, where N is the number of ciphertext blocks. In each attempt, the tool performs N decryptions. Therefore, the algorithmic complexity of the attack is $O(N^2)$.

Defense. The current versions of the standard have no means to protect against the POA. Simple ways to fix this include

- Changing the padding scheme to AByte or OZ padding. Since these schemes have no invalid padding, decryption never fails due to incorrect padding [9, 28].
- Changing to AES-CTR mode, which requires no padding of the plaintext.

The above two modifications require minimal changes to the mark-up format mandated by the current version of the standard, although both would require tool-specific modifications. Anyway, neither of these simple defenses actually prevent recovery of the plaintext, as we will see in the next section.

Using a proper authenticated encryption (AE) scheme *would* prevent the POA and the new attack that we are about to give. From a cryptographic perspective, we recommend mandating an AE scheme with support for associated data (AEAD) [31]. The associated data (AD) should be all of the digital envelope that is not the Data Block, so that there is a proper binding between AD and Data Block. Our recommendation would be achieved with the least number of changes by demanding (1) that the HMAC computation always is carried out, (2) that the scope of the HMAC computation is the entirety of the digital envelope, specifically including the encrypted Data Block, and (3) that every decryption failure results in a single error signal. For the last, this means that the padding must be checked even if the HMAC check fails, to avoid enabling the POA via a timing-channel [11] Moreover, no processing of the digital envelope beyond these checks should occur if decryption fails.

That said, supporting *any* AEAD scheme would require significant changes to the standard and the EDA tools. So it is worth evaluating other provably secure AEAD schemes with respect to their efficiency and operational considerations.

3.2 Syntax-Oracle Attack

EDA tools need to provide an extensive debugging environment so that any SoC design issues can be swiftly identified. This applies to encrypted IPs as well, since IP users need the ability to detect potential design errors and systhesis issues in the purchased IPs. The P1735 standard highlights these needs, as we noted in the Introduction with quotes from [13, Section 10].

Our SOA exploits the syntax errors reported by EDA tools in a manner similar (but not identical) to the POA. The main strategy is to inject into the decrypted plaintext, via manipulations of the ciphertext, a particular character that will elicit a unique *syntax-error message* when the plaintext is processed by the tool.[2] In Verilog grammar, we have found that the ` (backquote) character has these unique properties. The ` symbol is a Verilog keyword that indicates preprocessor directives such as "define", "include", and "ifdef". For example, "` define S0 1" defines a macro S0 that is replaced by 1 during preprocessing of the plaintext IP. If the backquote character is followed by any token other than the supported directives, the EDA tool reports one or more syntax errors. (This is a property of Verilog parsers.) These errors can be used akin to the padding oracle to recover the plaintext IP.

For example, when Synplify encounters a misplaced backquote symbol, it throws one of the following two errors: "expecting identifier immediately following back-quote" or "Unknown macro". In our attack, we use these two error messages to affect a syntax oracle (**SO**). When the oracle is given a ciphertext (C) as input, it returns 1 if either of these two errors occurs, and 0 otherwise.

We use the same example as the POA to explain our SOA. Let the ciphertext message be $C = IV \,\|\, C_1 \,\|\, C_2 \,\|\, C_3$. In SOA, the attacker can target *any* ciphertext block. (In padding oracle attack, the target block should be the last block of ciphertext; the ciphertext can be truncated to make the target block, the last block.) Let the target ciphertext block be C_2. The attacker can guess the plaintext characters of the target block in *any* order.

Let's suppose the attacker is interested in learning the 5^{th} byte of the 2^{nd} block, i.e., $P_2[5]$. The SOA attack for this case is illustrated in Figure 4. We first replace $C_1[5]$ with the guess byte, g, i.e., $C_1'[5] = g$, where $C_1'[5]$ is the modified value of $C_1[5]$. We then query the **SO**. If the oracle returns 0, the same process is repeated with a new guess byte. When **SO** returns 1, we stop because it indicates that the ` character is present at $P_2'[5]$, the modified value of $P_2[5]$. We extract the $P_2[5]$ value by $C_1[5] \oplus 0x60 \oplus g$. (0x60 is the ASCII value of `). The same process is repeated to find the rest of the plaintext.

To see that the attack works, consider the following. Before the attack, $P_2[5] = C_1[5] \oplus y[5]$, where $y = E_K^{-1}(C_2)$. When the **SO** returns 1,

- $P_2'[5] = C_1'[5] \oplus y[5]$,
- $P_2'[5] = 0x60$, and

[2]We define a unique syntax error as an error that is caused only by presence of a particular character in the IP.

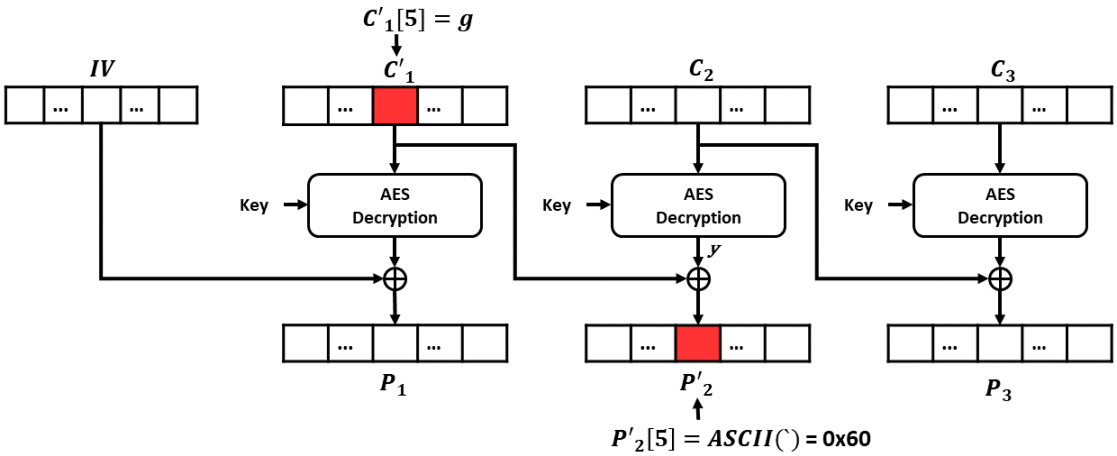

Figure 4: Syntax-oracle attack to extract the plaintext, $P_2[5]$. Before the attack, $P_2[5] = C_1[5] \oplus y[5]$. When $C'_1[5] = g$, suppose **SO** returns 1, i.e., $P'_2[5] = 0x60$. Since, $P'_2[5] = C'_1[5] \oplus y[5]$, so $P_2[5] = C_1[5] \oplus 0x60 \oplus g$. ($P'_2$ and C'_1 represent the modified plaintext and ciphertext block.)

- $C'_1[5] = g$.

So, $P_2[5] = C_1[5] \oplus 0x60 \oplus g$.

In case of AES and a plaintext alphabet of ASCII bytes, it would require at most $(256 \times 16 \times N)$ attempts to extract the entire IP, where N is the number of ciphertext blocks. Each attempt requires N AES decryption. So, the algorithmic complexity of this attack is $O(N^2)$.

Defense. Our SOA relies on modifying the ciphertext to inject specific syntax errors in the decrypted plaintext. Our attack works because the P1735 standard does not provide any integrity protection for the Data Block, and encourages the return of descriptive syntax errors. As the latter seems crucial for facilitating SoC design, we recommend a cryptographic solution. As noted in the discussion of POA defenses, we recommend using a proper, provably secure AEAD scheme, and treating all of the digital envelope that is not the Data Block as associated data.

3.3 Optimizing the syntax-oracle attack

In the worst case, the SOA requires $256 \times 16 \times N$ attempts to extract the plaintext IP consisting of N ciphertext blocks. For $N = 10,000$, the SOA would require roughly 40 million attempts to recover the plaintext. For each attempt, the EDA tool must decrypt the IP and run a syntax check. Our experimental results show that a single attempt takes around 0.25 seconds, on average. Therefore, for a 10,000 block IP, the SOA would take nearly 40 months to extract the entire plaintext. In short, the basic SOA may not be practical for large scale industrial IPs. In this section, we provide optimizations for the SOA that significantly reduce the run time of the attack.

Reduce sample space of guess byte (RSSGB). Consider the example introduced in the previous section. In the first step of the attack, $C'_1[5] = g$. Instead, the adversary could set $C'_1[5] = g_1$, where $g_1 = C_1[5] \oplus 0x60 \oplus g$. This optimization improves the attack efficiency by reducing the number of attempts to extract the plaintext. To see why, observe that

$$\begin{aligned} P_2[5] &= C_1[5] \oplus 0x60 \oplus g_1 \\ &= C_1[5] \oplus 0x60 \oplus (C_1[5] \oplus 0x60 \oplus g) \\ &= g \end{aligned}$$

Note that the ciphertext is an encryption of valid Verilog code. Since the guess byte is now equal to $P_2[5]$, it would be a valid Verilog character, and hence its range would be between 1 and 128. The maximum number of attempts to find all of the plaintext therefore reduces from $256 \times 16 \times N$ to $128 \times 16 \times N$. This optimization also works for the POA.

Reducing AES decryptions (RAD). In AES-CBC, a plaintext block is a function of two ciphertext blocks, namely $P_N = D_K(C_N) \oplus C_{N-1}$, where $D = E^{-1}$. The Synplify tool reports errors after it decrypts the entire ciphertext, and performs a syntax check on the resulting plaintext. This adds a lot of latency as the tool has to decrypt extra $N-2$ blocks of ciphertext to recover each targeted plaintext block. It is possible to parse and modify the ciphertext so that it only contains the target block and the block before the target block. However, any target block other than the last block will not have proper padding, and the tool does not report syntax errors if it finds a padding error. We counter this problem by using the following preprocessing — discard all ciphertext blocks except the last two blocks, the target block, and the block before it. The last two blocks prevent concealing of syntax errors due to padding errors. For example, consider a 100-block ciphertext, $C = IV \,\|\, C_1 \,\|\, \cdots \,\|\, C_{99} \,\|\, C_{100}$. To recover C_3, we could give $C' = C_2 \,\|\, C_3 \,\|\, C_{99} \,\|\, C_{100}$ as input to the tool, instead of C. Now, in each attempt, the tool has to decrypt just 4 blocks of ciphertext to get the plaintext instead of N. Owing to this optimization, the algorithmic

complexity reduces from $O(N^2)$ to $O(N)$. To be precise, the tool can *save* up $128 \times 16 \times N \times (N-4)$ AES decryption operations in each attack. For a 1,000-block Verilog code IP, the attack can be 250x faster than the RSSGB optimization. This optimization also works for POA.

All-blocks-at-once attack (ABAO) . The syntax-oracle attack can be independently applied to extract a character from any particular position. Also, instead of aiming to inject the backquote character/symbol at one position, we can aim to inject it at multiple position at the same time. The EDA tool will report the respective locations (in the decrypted IP) where it encounters errors due to the backquote symbol. These properties make the SOA inherently parallelizable and we can exploit it to gain a massive speedup.

The optimized attack needs some pre-processing similar to the previous optimization. This is shown in Figure 5. We first break the Data Block of the encrypted Verilog code into groups, where each group consists of a target block, its preceding block, and the last two blocks in the Data Block. A module is the basic unit of hierarchy in Verilog. So, each group is given a unique module name and is written on to a separate file. For example, for target block C_1, we write $IV \parallel C_1 \parallel C_{N-1} \parallel C_N$ in the Data Block of module1; for target block C_2, we write $C_1 \parallel C_2 \parallel C_{N-1} \parallel C_N$ in the Data Block of module2, and so on. It can be easily seen that the number of files that needs to be created is equal to the number of encrypted blocks in the Data Block of the original Verilog module. We then write a main module ("top" in Figure 5) in a separate file that can invoke the modules that we just constructed. Next, we modify (xor with 0x60 and the guess byte) all characters of the target block in each module and pass all files (module1, module2, ..., module100 and top) to the EDA tool. The tool checks for syntax errors in all the files. Notice that, in this case all instances of the guess byte (if present in the target block of a module) and their relative position to the start of the file will be known in a single attempt. So, after *128* attempts, which is the sample space of valid Verilog characters, we find all the characters in the original encrypted Data Block.

For this optimization, the algorithmic complexity of the attack is $O(N)$ where N is the number of ciphertext blocks. To be more precise, the attack takes a maximum of 128 attempts to find all of the plaintext. The maximum number of AES operations that the tool performs is equal to $128 \times N$, as compared to $256 \times 16 \times N \times N$, in case of no optimization. For a Data Block that contains 1,000 AES-128 encrypted blocks, this optimization reduces the worst case by more than 4 billion AES operations. Note that, the previous optimization was sequential in nature, while the current one is highly parallelized, as not only can we target all blocks at once, we can also find all instances of a single guess byte in a single run.

This gigantic stride in efficiency comes at a loss of accuracy. The ABAO optimization can introduce characters like EOF, double-quote and comment symbols in the decrypted plaintext. These characters also cause syntax errors which

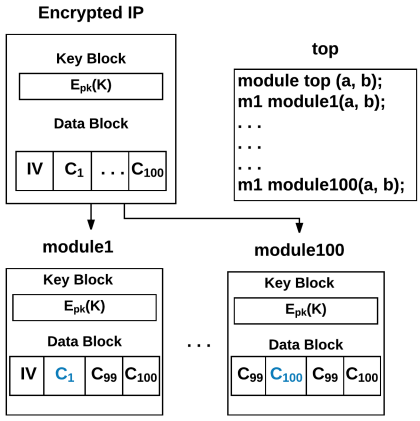

Figure 5: Modules creation in SOA - ABAO optimization. The blue ciphertext block is the target block in each module.

can mask the target (the backquote symbol). Therefore, the SOA with this optimization will fail to extract some plaintext characters. Our experimental results show that the SOA with ABAO optimization can extract around 85% of the total plaintext. For an adversary with subject matter expertise, it is feasible to infer the rest of the plaintext of the overall encrypted IP.

There is a clear trade-off between the accuracy and run-time of this attack. An adversary could use this optimization to recover a substantial portion of the plaintext in a short time. If there is spare bandwidth, they could run the basic SOA/POA to find the missing characters. A tweaked version of this optimization can be applied to the POA. We discuss this in Section 6.

Exploit frequency distribution. The attacker can also exploit the frequency distribution of characters in Verilog grammar to select guess bytes instead of making randomized guesses. For more efficiency, they could use a Markov model to make adaptive guesses based on partially decrypted plaintext.

Run parallel instances of the tool. Another trivial optimization involves the IP user running multiple instances of the Synplify tool in parallel to recover separate portions of the plaintext. The number of instances that could be run in parallel are controlled by the EDA tool. One could argue that the adversary could procure multiple licenses of the Synplify tool and recover the plaintext IP in a short time. But, these licenses are very expensive and cost upwards of $100,000.

3.4 POA vs. SOA

Table 1 shows the trade-off of each optimization with respect to the accuracy of the attack, for a 1,000 block ciphertext. Since run time of an attack is directly proportional to the number of AES operations, we use the maximum number of AES operations as an approximation of the run-time. The POA and the basic SOA can extract $\sim 100\%$ of the encrypted plaintext. While POA is restricted to AES-CBC

Table 1: Trade-off of approximate accuracy v/s number of AES operations (in worst case) in all the confidentiality attacks. The analysis is over an encrypted IP whose Data Block consists of 1,000 ciphertext blocks.

SN	Attack	#AES-decryptions	Approximate accuracy
1	Basic POA	4.096×10^9	100%
2	Basic SOA	4.096×10^9	98%
3	(2) + RSSGB	2.048×10^9	98%
4	(3) + RAD	8.192×10^6	98%
5	(2) + ABAO	1.28×10^5	85%

with a padding scheme that reports padding errors, the SOA relies on a unique syntax error due to a specific plaintext character in the underlying HDL (there could be more than one such character in the HDL). Both of these attacks have high run time owing to a large number of AES decryption operation. The optimizations on the SOA improve the run-time of the attack but also cause a loss in accuracy.

4 INTEGRITY ATTACKS

The modern SoC design flow involves third parties and even in-house teams that are located across the globe. For such a distributed design process, the trustworthiness of entities, their IP, and their actions are difficult to verify. For example, an SoC integrator may have different design teams located in different parts of the world. The design team that is responsible for designing security critical IPs for the SoC (e.g., Trusted Platform Modules) may not trust other design teams as they could tamper with the IP surreptitiously and avoid detection during functional verification and testing of it. In these scenarios, these IPs are encrypted using the P1735 standard to protect against malicious tampering.[3]

The standard does not consider any authenticity check on the identity of IP authors. However, it purports to provide integrity protection for the digital envelope by providing an HMAC computation over the Key Block (or the entire Rights block, if this includes more than just the Key Block). In this section, we demonstrate that the P1735 standard cannot ensure integrity protection of an encrypted IP based on two different attacks.

In the first attack (see Section 4.1), we present a way to maul the ciphertext so that our desired modification appears in the resultant plaintext *without* causing any syntax errors. The second attack (see Section 4.2) allows an adversary to insert ciphertext blocks in the encrypted IP such that no syntax errors are raised and exploit the lack of authentication of the IP author. Both of these attacks are performed without the knowledge of the decryption key, and can be applied to various security critical IPs with disastrous consequences. Moreover, these attacks work with any IP and any set of RTL instructions that the adversary wants to insert in the IP.

In these attacks, we insert a hardware Trojan in an encrypted crypto-accelerator IP that implements the AES algorithm in hardware [2]. To the best of our knowledge, *this is the first demonstration of hardware Trojan implementation in an encrypted IP.* When it observes one specific plaintext, our Trojan leaks the on-chip secret key used by the AES IP. The schematic of the Trojan (T) is shown in Figure 7(a). To implement it, an adversary needs to insert the code shown in Figure 7(b) into the encrypted RTL. Here, PT and CT are the plaintext and ciphertext ports of the AES module and T_j is the triggering condition. When PT is equal to T_j, the key is leaked. Note that detecting this Trojan is extremely difficult because it delivers its payload only when it observes a specific 128-bit plaintext. In addition, it is worth noting that traditional Trojan detection techniques [32], [12], [36] cannot be applied on encrypted IPs. We reiterate the fact that the P1735 standard does not facilitate any authenticity check on the encrypted IP and therefore, any modification in the encrypted IP is not detected.

4.1 Trojan Insertion in Crypto-accelerator - I

In this attack, we first recover the plaintext IP using one of our prior confidentiality attacks. Then, we manipulate the initialization vector (IV') to insert a start-comment directive in the first block of the plaintext (since $P'_1 = D_K(C_1) \oplus IV'$). Next, we insert two additional blocks —the attack block A_1, and the victim block V_1— after the first cipher text block. Each attack block is modified in an adaptive manner; all the victim blocks are same as C_1. Our aim is to modify the attack block so that the desired changes are reflected in the plaintext-block (P_{V_1}) corresponding to V_1. This is explained with the following equation,

$$P_{V_1} = D_K(V_1) \oplus A_1$$

Note that, we know $D_K(V_1)$ from the confidentiality attack. Therefore, we can change A_1 to make any specific changes in P_{V_1}. However, the plaintext block, P_{A_1} corresponding to

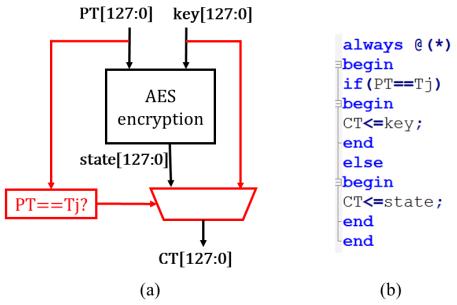

Figure 7: (a) Schematic of a Trojan which leaks the on-chip private key used by the AES IP. (b) The Verilog code which implements this Trojan.

[3] Anecdotally, representatives of the semiconductor industry have stated that the community is adopting the recommendations of the P1735 standard to ensure the *integrity* of IPs, too.

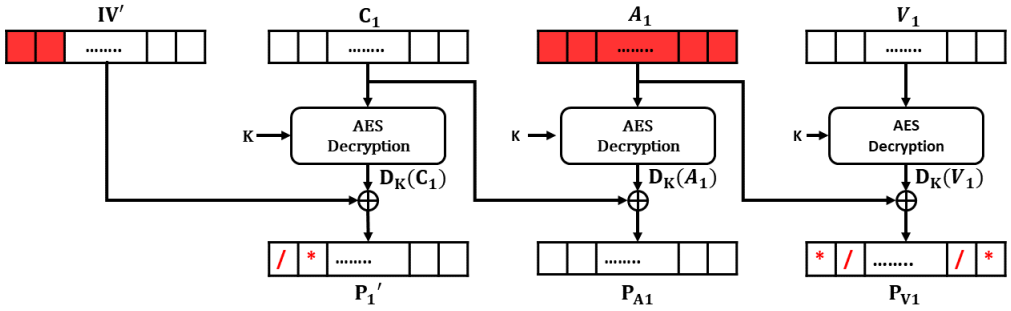

Figure 6: Integrity attack-I on the P1735 standard. The IV is modified to insert the start-comment directive in P_1. A random block (A_1) is inserted after C_1. The resulting plaintext text (P_{A_1}) is not checked by the tool for syntax errors as it is treated as commented characters. The attack block (A_1) is appended to the victim block ($V_1 = C_1$). A_1 is tampered to insert the end-comment directive in the first two bytes of P_{V_1}, and the desired Trojan's first twelve characters in the next twelve bytes. The process is repeated by tampering A_1 to insert the start-comment directive in the last two bytes of P_{V_1}. Precondition: the attacker knows at least one block of the plaintext.

A_1 would consist of random characters, which in turn would cause syntax errors with very high probability. We counter this by commenting out the P_{A_1} block. As mentioned in the previous paragraph, we have modified IV' to insert a start-comment directive in P_1'. We now modify A_1 in such a manner that it introduces a end-comment directive in P_{V_1}. Therefore, the P_{A_1} block is encapsulated inside a comment section, and the EDA tool does not check for syntax errors in commented sections of the IP.

Figure 6 illustrates our proposed integrity attack. The $/*$ is the Verilog directive for start of comment and $*/$ is the directive for end of comment. Notice that, the P_{A_1} block is encapsulated inside a comment section. Also, we modify the last two bytes in A_1 to insert a start-comment directive in the last two bytes of P_{V_1}. This allows us to insert the subsequent attack blocks, A_i, and victim blocks, V_i, where $i > 1$. All the victim blocks are identical in our attack. Since each of the victim blocks has a end-comment directive in the first two bytes and start-comment directives in the last two bytes, it allows us to insert any Verilog code in the rest of the twelve bytes. We insert the Verilog code for the Trojan in these twelve bytes in an incremental manner until the entire code is inserted. After that, we append the original ciphertext blocks. Note that, we use the same C_1 block for all victim blocks. Instead, we could use any ciphertext block, provided we know its corresponding plaintext. The Trojan-inserted ciphertext is given as follows. $C = IV' \| C_1 \| A_1 \| V_1 \| A_2 \| V_2 ... \| A_m \| V_m \| C_2 \| ... C_n$. Here, $V_1 = ... = V_m = C_1$, and m, n represent the number of attack-/victim blocks and original ciphertext blocks, respectively.

Defense. The defenses recommended for the SOA provide integrity checks on the Data Block in particular, and the entire IP in general. Hence, these defenses would stop the integrity attacks.

4.2 Trojan Insertion in Crypto-accelerator - II

In a global design process, authentication of participating IP authors is of paramount importance. The P1735 standard does not address this issue. Thus, it is trivial for an adversarial IP author (a rogue employee of the SoC integrator) to insert a Trojan in its own IP. However, it can also target security-critical IP's belonging to non-adversarial IP authors.

In this attack, we first extract the plaintext IP, P using one of our confidentiality attacks. Then we insert the Trojan, T in the plaintext IP at any desired position. We then chose a random session key, K', and encrypt P' (trojan-inserted IP) under the session key to get the encrypted Data Block. After that, we encrypt the session key under the public key of the tool to get the Key Block. The Data Block and the Key Block are bundled together as per the standard to get the digital envelope.

To defend against this, it would be sufficient to prevent the recovery of the plaintext IP, i.e., apply the suggestions from Section 3.

5 LICENSING ATTACKS

The standard specifies a rights management mechanism that can control the amount of information the tool outputs during processing of the encrypted IP, such as names and location of protected variables during error reporting, output signals during simulation, etc. It also describes a licensing mechanism that controls such rights based on whether the IP user has the corresponding license or not. The standard describes a protocol to implement the licensing scheme. The protocol consists of 4 sub-protocols. Some of these sub-protocols are vague, and give way to trivial attacks due to use of AES-CBC

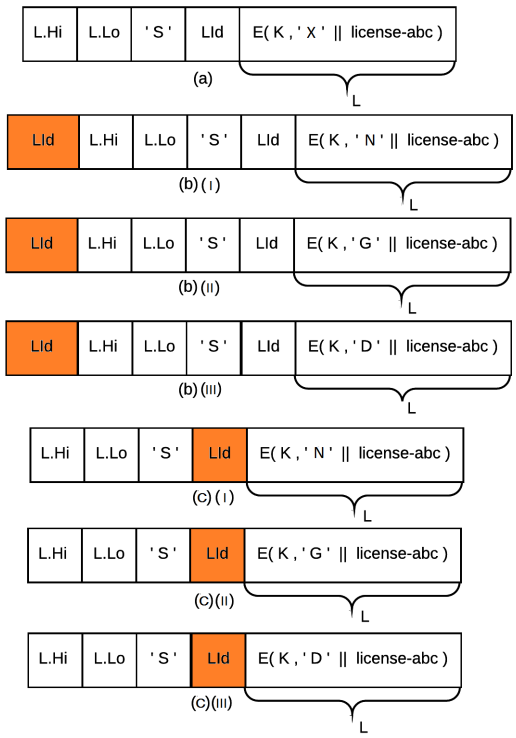

Figure 8: Format of different messages in the licensing protocol. (a) Format of a symmetric key encrypted message. 'X' represents the command byte, and can have values like 'N', 'G', or 'D'. L represents the symbolic encryption of license-abc. (b) and (c) show possible formats for (I) new license-request, (II) license-grant, and (II) license-deny messages. While in (b), the license id is prepended to the encrypted message, in (c), it is prepended to the symbolic encrypted message.

with fixed IVs, and a poor authentication mechanism of the license-request and license-response messages.

Threat model. The principals in this threat model are the IP user, a proxy server controlled by the IP author, and the tool that parses the IP for licenses to make license requests on behalf of the IP user. The IP author and the tool are trustworthy entities; the IP user is an adversary who does not have the valid license(s) and tries to get access to protected sections of the IP cores.

The licensing protocol. The licensing protocol can be divided into four sub-protocols: key exchange, license request, license response, and heartbeat. The protocol as whole proceeds as follows. The tool creates a socket connection with the proxy server and runs a key exchange protocol to establish a shared key (K). For each new license, it creates a new license id (LId), encrypts the license request under the shared secret key (K) and license id as the initialization vector, prepends the *encrypted message* with the license id, and sends the message to the proxy server. The proxy performs license validation and sends back a license grant/deny response. The license id, which is unique for each license request, is used by the proxy, as well as the tool, to distinguish between multiple licenses.

The license-request, license-grant and license-deny messages are prepended with the command byte 'N', 'G' and 'D' respectively, before carrying out the encryption. In plaintext, these messages are identical, except the *optional* application-specific string or denial message that is appended to the grant/deny message. The resultant encrypted text is referred to as "symbolic encrypted message" (denoted by L in Figure 8). The tag ('N', 'G' and 'D') in the first byte of the plaintext makes the symbolic encrypted messages different, despite being identical in the rest of the plaintext bytes. The standard cites authentication of the request message in the grant or deny response as the reason behind this design choice [13, Section 8].

A potential attack. In all protocols for license management, the (symmetric-key encrypted) messages that are exchanged between the proxy server and the tool have a specific format. See Figure 8(a). As per the standard, the license id should be prepended to the "encrypted message" in each license-request, license-deny or license-grant message. For example, the license-request message (LR) for license-abc can be syntactically represented as $\text{LR} \leftarrow \text{LId} \,\|\, \mathcal{E}_{\text{K}}^{\text{LId}}(\text{N} \,\|\, \text{license-abc})$, which can either be the symbolic encrypted message (L) or the entire encrypted message (L.Hi || L.Lo || 'S' || Lid || L). This is shown in Figure 8(b), and 8(c). A similar situation holds for license-grant and license-deny messages, too. Keep in mind tha $\mathcal{E}$ is CBC-mode over a particular blockcipher (likely AES).

In Figure 8(b), the license id is prepended to the entire encrypted message. As mentioned earlier, the license id is used to distinguish between multiple licenses. So, while processing the license-request message, the tool could parse the first two bytes to get the license id, check if it is a replay, and close the socket in that case. This prevents further processing of the "symbolic encrypted message" (L). Otherwise, if it is a new request, the proxy would call its decryption API for symmetric messages. The API decrypts L, and returns the plaintext ($\text{N} \,\|\, \text{license-abc}$) to the caller function. Next, the proxy checks whether license-abc is valid, and sends an LG/LD message back to the tool. Note that this decryption API for symmetric messages can be used in other protocols as well.

Since, the symmetric message format ensures that the IV (= license id) is always prepended to the "symbolic encrypted message", one could avoid prepending the IV to the entire encrypted message. See Figure 8(c). However, the license id is used for detecting replays and validating license requests. In this case, the decryption API for symmetric messages has to be overloaded to return the decrypted text and the IV. The IV, which is same as the license id, would be used by the calling function to check for replays and validation.

While the two formats might seem alike in terms of securing the socket communication, it is not the case. The format in Figure 8(b) is susceptible to a simple man-in-the-middle

attack, where the IP author could intercept an LD message from the proxy, and convert (xor first byte of license id in the IV field of the LD message with $D \oplus G$) it into an LG message, and hence get through the license check without actually having the particular license. Note that this is a simple exploitation of CBC with fixed IV. On the contrary, the format in Figure 8(c), inadvertently enforces an integrity check on the IV (by checking the license-id for replays or modifications).

More attacks. The standard is also vague in the processing of license grant/deny response sent by the proxy. It allows the proxy server to send optional application-specific strings concatenated with the license grant/deny response. But, it does not specify security checks that need to be performed on these strings. Since there is no integrity check on the CBC encrypted messages, an adversary (say, a competing IP user) can append any number of random cipher text blocks with the LG/LD responses. If the tool does not check the length of the LG/LD messages, the tool could crash due to memory overflow.

The standard requires that the proxy and the tool send periodic heartbeats to each other to know whether the receiver is alive or not. But, it does not specify how the heartbeat protocol behaves after the tool sends a license request. If the proxy and the tool send periodic heartbeats till the tool receives an LG/LD response from the proxy, and the proxy gets back a license release or a new license request, an adversary can cause a denial-of-service by just dropping these response messages.

Defense. The standard could recommend the use of TLS 1.2 (or higher version numbers) to exchange license requests and responses. Also, it must explicitly define protocols for all stages - handshake, license request, license response (grant and denial), and heartbeat.

6 EVALUATIONS

In this section, we evaluate the efficiency and accuracy of the padding-oracle attack and the syntax-oracle attack on the P1735 standard. We used open source semiconductor IPs from OpenCores [2], which is the largest site/community for the development of open source semiconductor IPs. We chose the following IPs for our benchmark — flipflop (FF), square-root arithmetic core (SQRT), SHA-256 digest core (SHA), Fast Fourier Transform DSP core (FFT), AES-128 crypto core (AES), Reed Solomon ECC core (RS), memory controller core (MC), and CISC processor (CISC). These IPs have different functionalities and range from small to industrial scale in size. Some of these IPs are generally procured/licensed from third party IP developers. We also selected the c7552 ISCAS benchmark which represents a firm IP. Note that the POA and SOA work on any semiconductor IP encrypted using the P1735 standard.

In the first step, we encrypted these IPs using the P1735 standard with an encryption script provided by Synopsys [3]. We then execute our padding-oracle and syntax-oracle attacks. We ran our experiments with Synopsys' Synplify Premier EDA tool (Version L-2016.09) installed on a CentOS virtual machine with 4 Intel core processors (each with a clock speed of 2.2GHz), and 8GB RAM.

Padding-oracle attack. We ran the padding-oracle attack with the optimizations that are inspired by the syntax-oracle attack (see Section 3.3). The aim of this attack was to decrypt the Data Block of the encrypted IP, which acts as the ciphertext in this case. The ciphertext was broken down into modules similar to the ABAO optimization process in the SOA, except each module consisted of two encrypted blocks instead of four. Since decryption precedes syntax-check, padding errors are never concealed due to any syntax errors. So, the target ciphertext block and its previous block are sufficient to generate appropriate padding in the targeted plaintext blocks.

In our experiment, the Synplify tool did not report any padding error when the ciphertext blocks were tampered as per the padding-oracle attack. But, it gives the warning "encrypted data mangled". We use this warning message to design our padding oracle. For each guess, the ciphertext is modified and fed to the Synplify tool for a syntax check. The tool writes all errors and warnings in the "syntax.log" file. The presence/absence of the warning "encrypted data mangled" in the log file is used as the padding oracle.

Table 2 shows the summary of the optimized POA on the 9 benchmark IPs. Figure 9(a) (red plot) shows the evaluation of time as a function of the number of ciphertext blocks in a loglog plot, whereas, Figure 9(b) (red plot) shows the accuracy of the POA. With the ABAO optimization, the algorithmic complexity of the attack is $O(N)$, where N is the number of ciphertext blocks in the encrypted IP. When the number of ciphertext blocks increases beyond 1,000 the tool seems to slow down, and this can be seen as a change in slope of the plot after 1,000 blocks. We can overcome this partially by breaking down a large IP into chunks of say, $< 2,000$ blocks and running the attack multiple times with these chunks. We did this for the CISC and c7522 benchmark (last two points in the plot). The accuracy, on the other hand, is nearly 100% for all the IPs.

Syntax-oracle attack. We ran our SOA with the ABAO optimization on the nine benchmark IPs. Table 2 summarizes the result. Figure 9(a) (blue plot) shows the evaluation of time as a function of the number of ciphertext blocks. The plot has similar attributes as the POA, except it is around 13 times faster. This can be observed as the nearly constant width between the two plots in Figure 9.

Figure 9(b) shows the accuracy as a function of lines of code (LOC) for SOA with ABAO optimization. In this case, the average accuracy is 85.3% with a standard deviation of 4.4%. We note that if we do not apply the ABAO optimization, then the average accuracy increases to 98%, while execution time increases by 16x. For example, the accuracy for FF, SQRT, FFT, AES, and RS increases to 100%, 95.9%,

Table 2: Results for SOA and POA attacks.

	# of Blocks	4	53	467	653	958	1268	2225	5071	9183
	# of Lines	7	51	614	440	712	1374	1854	2083	3858
SOA	Time (sec)	54.7	60.0	130.3	165.9	228.4	287.3	439.2	1065	2524
	Accuracy (%)	76.6	87.9	85.1	87.3	91.5	84.6	82.3	83.1	89.9
POA	Time (sec)	706.7	798.8	1677.4	1888.6	2484.3	3203.1	5575.0	12990.163	25454.234
	Accuracy (%)	100	100	100	100	100	100	100	100	100

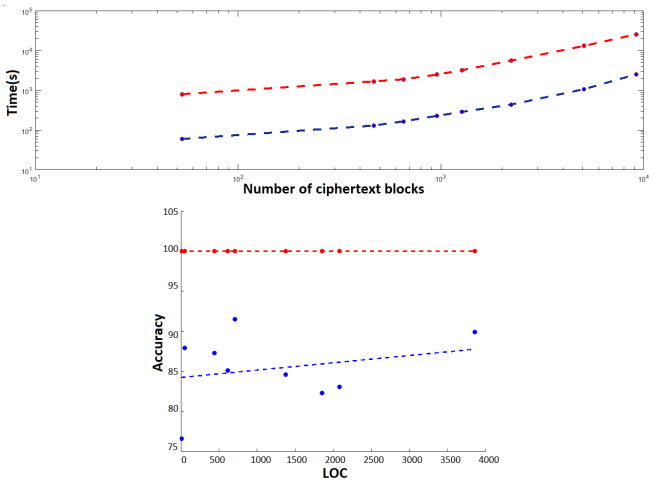

Figure 9: Top: Time vs number of ciphertext blocks for SOA (blue) and POA (red). **Bottom:** Accuracy vs LOC for SOA (blue) and POA (red) .

99.5%, 98%, and 97.2%, respectively without the ABAO optimization.

Comparison between the padding-oracle attack and the syntax-oracle attack. From Figure 9, it is evident that with the ABAO optimization, the POA is around 13 times (mean 12.6 with a standard deviation of 1.3) slower than the SOA. This is because in the former attack, for $j < 16$, the j^{th} plaintext character can be guessed only when the $(j+1)^{\text{th}}$ character has already been found. There is no such restriction on the latter attack. We find all instances of the guessed character in the entire ciphertext in a single guess.

Though the SOA is fast, it loses out (some) on accuracy with the ABAO optimization. Its accuracy has an average of 85.3% with standard deviation of 4.4%, whereas the POA is always 100% accurate. But, without the ABAO optimization, the accuracy of the syntax-oracle attack shoots to nearly 98%. We reiterate that the POA works only with AES-CBC and padding schemes which distinguish between a valid and invalid padding. On the other hand, the SOA has no such restrictions.

7 RELATED WORK

To the best of our knowledge, Myrian and Chow [23] provide the only work that presents any attack on the IEEE P1735 standard. The authors show how an encrypted IP from FPGA technology can be mapped to an ASIC technology. The proposed technique takes the encrypted RTL code and synthesizes it to the plaintext netlist using FPGA primitive. This FPGA implementation is then mapped to an ASIC implementation. This technique does not reveal any weakness of the P1735 standard as the authors did not consider the fact that the IEEE P1735 standard has guidelines to encrypt the synthesized netlist as well. Major FPGA vendors like Synplify and Vivado support this feature. Also, this technique never recovers the highi-level RTL code which is of main interest for IP piracy.

There have been numerous attacks on various protocols and standards that use CBC mode for encryption. In [35], Vaudenay showed that padding errors in CBC mode can be used as an oracle to get the decrypted text without knowing the key. Canvel et al. extended this idea in [11] by exploiting timing difference between errors due to bad MAC and those due to improper padding, to intercept the password of an email client that connects to an IMAP server over SSL/TLS.

While Vaudenay exploited the RC5-CBC-PAD algorithm [7], Paterson and Yau demonstrated efficient attacks on the ISO CBC Mode Encryption standard to recover plaintext [29]. These attacks required IVs to be public. The same group of researchers came up with new padding-oracle attacks against the revamped ISO CBC Mode Encryption standard that recommended private and random IVs instead of public IVs [37]. Joux et al. in [15] gave attacks on CBC mode of encryption by adversaries that can adaptively choose chunks (one or more blocks) of plaintext bytes to find the original message. They termed these adversaries as block-wise-adaptive adversaries. In [9], Black et al. studied Vaudenay's attack with different padding schemes - XY-pad, OZ-pad, AByte-pad, to name a few. They found that padding methods that have no invalid paddings are immune against padding-oracle attacks, which was corroborated by Paterson and Watson in their provable security analysis of CBC mode against padding-oracle attacks [28]. One such padding scheme is AByte-pad. In [19], Klíma et al. used ASN.1 encoding errors in PKCS#7 with AByte padding as an oracle to invert the ciphertext. Most of these attacks can be thwarted by enforcing integrity checks on the ciphertext.

Prior to Vaudenay's attack on CBC mode, Bleichenbacher presented an adaptive chosen-ciphertext attack that exploits padding errors in PKCS#1 v1.5 [10]. This attack was extended by Klíma et al. in [18]. They used errors due to the version number check in PKCS#1 as a side-channel. A

plaintext-aware encryption scheme, RSAES-OAEP was proposed to make it immune against such chosen ciphertext attacks. But, Manger in [20] exploited side channels in implementations of RSAES-OAEP as specified in PKCS#1 v2.0, to recover the plaintext message. This attack is based on the fact that the adversary can distinguish between errors during decoding and those due to incorrect integer to octet conversion; this is possible as the standard is vague on error conditions, such as unsupported MGF, handling of timing difference between the two errors, etc.

Apart from the above attacks, there has been a plethora of side channel attacks in the last 15 years - padding error attacks on RSA [14, 17, 22], timing attacks on AES-CBC implementations (MAC-encode-encrypt) in SSL/TLS [4], timing attacks on SSH [5, 6], and other side-channel attacks [9, 21, 24, 33, 39].

8 CONCLUSION AND FUTURE WORK

The P1735 IP encryption standard is widely used in the EDA community to protect confidentiality of high-value IPs. It also enforces fine-grained access control via rights management and licensing mechanisms. We have presented confidentiality and integrity attacks on the P1735 standard as implemented by the widely used Synplify Premier tool, a Synopsys EDA tool that provides an advanced FPGA design and debug environment. While the confidentiality attacks can reveal the entire plaintext IP, the integrity attack enables an attacker to insert hardware Trojans into the encrypted IP. This not only destroys any protection that the standard was supposed to provide, but also *increases* the risk premium of the IP. We also proposed various optimizations of the basic confidentiality attacks that reduce the complexity from $O(N^2)$ to $O(N)$.

The design flaws in P1735 are troubling considering the fact that it is susceptible to the classical POA that was reported in 2002, and it is disappointing that an international organization like the IEEE would mandate a brittle encryption mode (CBC) without any authentication, when there has been more than a decade of published research on AEAD schemes.

The standard also recommends PKCS#1 v1.5 as a padding scheme for RSA. As discussed in the related work section, there are many side-channel attacks on this padding scheme. In future work, we plan to attack the Key Block of the encrypted IP, which holds the encryption of the symmetric key used to create the Data Block, using RSA PKCS#1 v1.5 encryption scheme. We will also extend our cryptanalysis to other EDA tools (e.g. Xilinx), and evaluate license-proxy implementations complying with the P1735 standard as they become available.

9 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable feedback. This research was supported in part by Cisco Systems, Inc., in part by NSF grants CNS-1564444 and CNS-1564446, and in part by the National Institute of Standards and Technology grant 60NANB16D248.

REFERENCES

[1] EDACafe. EDA Industry Update September 2008. http://www10.edacafe.com/nbc/articles/. (EDACafe). Accessed: 2017-08-21.
[2] IP. OpenCores. http://opencores.org. (IP). Accessed: 2017-05-14.
[3] Synplify. Premier. https://www.synopsys.com/implementation-and-signoff/fpga-based-design/synplify-premier.html. (Synplify). Accessed: 2017-01-30.
[4] Nadhem J. Al Fardan and Kenneth G. Paterson. 2013. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 526–540. https://doi.org/10.1109/SP.2013.42
[5] Martin R. Albrecht, Jean Paul Degabriele, Torben Brandt Hansen, and Kenneth G. Paterson. 2016. A Surfeit of SSH Cipher Suites. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 1480–1491. https://doi.org/10.1145/2976749.2978364
[6] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. 2009. Plaintext Recovery Attacks Against SSH. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP '09)*. IEEE Computer Society, Washington, DC, USA, 16–26. https://doi.org/10.1109/SP.2009.5
[7] Robert Baldwin and Ronald Rivest. 1996. *The rc5, rc5-cbc, rc5-cbc-pad, and rc5-cts algorithms*. Technical Report.
[8] Mihir Bellare and Chanathip Namprempre. 2008. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. *J. Cryptol.* 21, 4 (sep 2008), 469–491. https://doi.org/10.1007/s00145-008-9026-x
[9] John Black and Hector Urtubia. 2002. Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption. In *Proceedings of the 11th USENIX Security Symposium*. USENIX Association, Berkeley, CA, USA, 327–338. http://dl.acm.org/citation.cfm?id=647253.720297
[10] Daniel Bleichenbacher. 1998. Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1. In *Proceedings of the 18th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '98)*. Springer-Verlag, London, UK, UK, 1–12. http://dl.acm.org/citation.cfm?id=646763.706320
[11] Brice Canvel, Alain Hiltgen, Serge Vaudenay, and Martin Vuagnoux. 2003. *Password Interception in a SSL/TLS Channel*. Springer Berlin Heidelberg, Berlin, Heidelberg, 583–599. https://doi.org/10.1007/978-3-540-45146-4_34
[12] Matthew Hicks, Murph Finnicum, Samuel T King, Milo MK Martin, and Jonathan M Smith. 2010. Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically. In *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, 159–172.
[13] IEEE. 2014. 1735-2014 - IEEE Recommended Practice for Encryption and Management of Electronic Design Intellectual Property (IP). (2014). http://standards.ieee.org/findstds/standard/1735-2014.html
[14] Tibor Jager, Jörg Schwenk, and Juraj Somorovsky. 2015. On the Security of TLS 1.3 and QUIC Against Weaknesses in PKCS#1 V1.5 Encryption. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2810103.2813657
[15] Antoine Joux, Gwenaëlle Martinet, and Frédéric Valette. 2002. Blockwise-Adaptive Attackers: Revisiting the (In)Security of Some Provably Secure Encryption Models: CBC, GEM, IACBC. In *Proceedings of the 22Nd Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '02)*. Springer-Verlag, London, UK, UK, 17–30. http://dl.acm.org/citation.cfm?id=646767.704309
[16] Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. 2010. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer* 43, 10 (2010), 39–46.
[17] Vlastimil Klíma, Ondrej Pokorný, and Tomáš Rosa. 2003. *Attacking RSA-Based Sessions in SSL/TLS*. Springer Berlin Heidelberg, Berlin, Heidelberg, 426–440. https://doi.org/10.1007/978-3-540-45238-6_33
[18] Vlastímil Klíma and Tomáš Rosa. 2003. *Further Results and Considerations on Side Channel Attacks on RSA*. Springer Berlin Heidelberg, Berlin, Heidelberg, 244–259. https://doi.org/10.1007/3-540-36400-5_19

[19] Vlastimil Klima and Tomáš Rosa. 2003. Side Channel Attacks on CBC Encrypted Messages in the PKCS# 7. (2003).
[20] James Manger. 2001. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) As Standardized in PKCS #1 V2.0. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '01)*. Springer-Verlag, London, UK, UK, 230–238. http://dl.acm.org/citation.cfm?id=646766.704143
[21] Christopher Meyer and Jörg Schwenk. 2013. SoK: Lessons learned from SSL/TLS attacks. In *International Workshop on Information Security Applications*. Springer, 189–209.
[22] Christopher Meyer, Juraj Somorovsky, Eugen Weiss, Jörg Schwenk, Sebastian Schinzel, and Erik Tews. 2014. Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 733–748. http://dl.acm.org/citation.cfm?id=2671225.2671272
[23] Vincent Mirian and Paul Chow. 2016. Extracting designs of secure IPs using FPGA CAD tools. In *Great Lakes Symposium on VLSI, 2016 International*. IEEE, 293–298.
[24] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. 2014. This POODLE bites: exploiting the SSL 3.0 fallback. *Security Advisory* (2014).
[25] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo Contreras, Domenic Forte, and Mark Tehranipoor. 2017. Hardware Trojan detection through information flow security verification. In *International Test Conference*. IEEE.
[26] Adib Nahiyan and Mark Tehranipoor. 2017. Code Coverage Analysis for IP Trust Verification. In *Hardware IP Security and Trust*. Springer, 53–72.
[27] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton. 2014. Reconsidering generic composition. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 257–274.
[28] Kenneth G. Paterson and Gaven J. Watson. 2008. Immunising CBC Mode Against Padding Oracle Attacks: A Formal Security Treatment. In *Proceedings of the 6th International Conference on Security and Cryptography for Networks (SCN '08)*. Springer-Verlag, Berlin, Heidelberg, 340–357. https://doi.org/10.1007/978-3-540-85855-3_23
[29] Kenneth G. Paterson and Arnold Yau. 2004. *Padding Oracle Attacks on the ISO CBC Mode Encryption Standard*. Springer Berlin Heidelberg, Berlin, Heidelberg, 305–323. https://doi.org/10.1007/978-3-540-24660-2_24
[30] Research and Markets. 2016. *Global Semiconductor IP Market - Global forecast to 2022*. Technical Report.
[31] Phillip Rogaway. 2002. Authenticated-encryption with Associated-data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS '02)*. ACM, New York, NY, USA, 98–107. https://doi.org/10.1145/586110.586125
[32] Hassan Salmani and Mohammed Tehranipoor. 2013. Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on*. IEEE, 190–195.
[33] Y Sheffer, R Holz, and P Saint-Andre. 2015. *Summarizing Known Attacks on Transport Layer Security (TLS) and Datagram TLS (DTLS)*. Technical Report.
[34] Mohammad Tehranipoor and Cliff Wang. 2011. *Introduction to hardware security and trust*. Springer Science & Business Media.
[35] Serge Vaudenay. 2002. Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS .... In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques: Advances in Cryptology (EUROCRYPT '02)*. Springer-Verlag, London, UK, UK, 534–546. http://dl.acm.org/citation.cfm?id=647087.715705
[36] Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. 2013. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 697–708.
[37] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. 2005. Padding Oracle Attacks on CBC-Mode Encryption with Secret and Random IVs. In *Proceedings of the 12th International Conference on Fast Software Encryption (FSE'05)*. Springer-Verlag, Berlin, Heidelberg, 299–319. https://doi.org/10.1007/11502760_20
[38] Lin Yuan, Gang Qu, Lahouari Ghout, and Ahmed Bouridane. 2006. VLSI design IP protection: solutions, new challenges, and opportunities. In *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*. IEEE, 469–476.
[39] YongBin Zhou and DengGuo Feng. 2005. Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. (2005). http://eprint.iacr.org/2005/388 zyb@is.iscas.ac.cn 13083 received 27 Oct 2005.

A CRITIQUE OF THE P1735 STANDARD

The P1735 standard recommends a lot of troubling cryptographic design choices that make the encrypted IP vulnerable to many attacks. Some of these attacks are described in this paper, but there could be more. The standard is also vague and erroneous in some of its security sensitive specifications and claims. In this section, we enlist these shortcomings of the standard.

- The standard states, "the protected IP has 100% fidelity to the original IP representation". This is not true as one could drop/add random ciphertext blocks owing to missing authentication checks on the Data Block.
- It makes no recommendations for AES-CBC padding, and leaves this important security decision to the tool vendors.
- In absence of a padding scheme, and any authentication whatsoever, AES decryption never fails. However, the resulting plaintext may get corrupt. The standard, on the contrary, has some mechanism due to which decryption could fail. This security sensitive design decision is again left at the discretion of the tool vendors.
- The standard mentions encrypting each IP with a one-time session key [13, Section 1], but it does not define a session explicitly.
- It recommends PKCS#1 V1.5 padding scheme for RSA. This scheme has been exploited as a side-channel to recover the underlying plaintext which is the session key in the digital envelope.
- In the licensing mechanism, the length of the encrypted messages is sent in clear text in both public-key and secret-key encryption. This makes the encrypted messages susceptible to ciphertext extension/truncation attacks in absence of any authentication.
- The standard is vague in its specification of the license response protocol. There are different security sensitive parameters like the length field, the command byte, the license id, etc, that are exchanged between the proxy server and the tool in each of their messages. Hence, it is crucial to clearly state how each of these parameters is checked/handled by the tool. However, the standard (again) leaves these security sensitive decisions at the discretion of the tool vendors.