MITRE eCTF 2025: DRM for Satellite TV Design Documentation

Jack Roscoe, Jihun Hwang (Jimmy), et al.

Team `Purdue3` (`b01lers`), Purdue University, West Lafayette, IN, USA

February 27, 2025

1 Introduction

Team Overview. The team designated `Purdue3` represents the Purdue Capture the Flag (CTF) team `b01lers`, along with two faculty advisors: Christina Garman and Santiago Torres-Arias. `Purdue3` (`b01lers`) consists of undergraduate and graduate students (Master's and Ph.D.) from various departments such as Computer Science, Computer and Information Technology, and Electrical and Computer Engineering. Students had an option to participate for credit via independent study courses administered by the aforementioned faculty advisors.

Competition Overview The MITRE Embedded Capture The Flag™ (eCTF) is a semester-long cybersecurity competition that allows students to develop secure designs for and attacks against embedded systems. MITRE's eCTF is split into two phases. In the first phase, teams design and implement a secure embedded system based on given functionality and security requirements. In the second phase, teams analyze and attack the designs of other competing teams, receiving "flags" of the form `ectf{...}` and are awarded points for doing so. Additional points are awarded for being the first to successfully attack a team's design, having the best poster and documentation, etc.

The 2025 MITRE eCTF scenario involves designing and implementing key parts of a secure satellite television system. Teams must design an `encoder` which will encode given television frames to be sent to a third party satellite. The satellite will then send encoded frames to listening `decoders`, the second item teams must design. `Decoders` send decoded frames to a listening host computer.

System Overview. The `decoder` takes the form of firmware residing on a MAX78000FTHR [Analog] evaluation board by Analog Devices, Inc. The board serves as the architectural basis of the secure satellite TV system whose goal is to prohibit unauthorized/unregistered users from receiving decoded data streams from protected television channels. The `encoder` takes the form of a Python script.

2 Threat Model

2.1 Security Requirements

(SR1) Attackers must not be able to decode TV frames without a decoder that has a valid and active subscription to that channel.

(SR2) The decoder should only decode valid TV frames generated by the satellite system the decoder was provisioned for.

(SR3) The decoder should only decode frames with strictly monotonically increasing timestamps.

For more information, refer to *Technical Specifications - Security Requirements* section in the official documentation [eCTF25].

2.2 Attack Scenarios

There is 1 deployment for all attack scenarios.

(AS1) **Expired Subscription:** Decode frames on channel 2 for which we have only an expired subscription.

(AS2) **Pirated Subscription:** Decode frames on channel 3 for which we only have a subscription for a different decoder.

(AS3) **No Subscription:** Decode frames on channel 4, for which we have no subscription.

(AS4) **Recording Playback:** Decode frames in a recording of channel 1, from before our subscription was active.

(AS5) **Pesky Neighbor:** Force a remote decoder to decode a frame that was not sent by the satelite, or force them to decode frames out of order.

For more information, refer to the *Attack Phase Flags and Scenarios* section of the official documentation for the competition [eCTF25].

2.3 Adversaries

The goal is to secure the satellite TV system largely against two different types of attackers:

- Traditional attackers that aim to exploit various bugs in the firmware such as memory corruption or cryptographic weaknesses.
- Powerful hardware attackers. We assume the most powerful hardware attacker could, in theory, read the entire contents of flash on any board to which they have physical access. This means hardware attacker can read all keys stored on a decoder they have physical access to.

3 Cryptographic Primitives

In this section, we outline the cryptographic primitives/schemes that were used to provide data confidentiality, authenticity, and integrity to our protocols.

3.1 Source for Random Number Generation

We utilize the host OS as a source of randomness to generate global secrets. These generated secrets in turn ensure secure encryption of frames. The protocol does not require any randomness on the decoder itself, allowing us to only rely on eCTF server's randomness, which is highly trusted.

3.2 `XChaCha20-Poly1305` Authenticated Encryption

Authenticated Encryption (AE) is a cryptographic primitive that can be used to provide both confidentiality and authenticity of messages. Roughly speaking, AE can be thought of as encryption together with a MAC tag τ, which is *internally* included as part of the ciphertext c and carefully checked during decryption.

Given a shared secret key k, encrypting an authenticated message m into ciphertext c is described as follows, where n is a public random nonce (stands for 'number used once'):

$$c \coloneqq \mathsf{Enc}_k(m, n)$$
$$\mathsf{Dec}_k(c, n) \coloneqq m \text{ or } \mathsf{Error}$$

`XChaCha20-Poly1305` [Arc20] is a variant of `ChaCha20-Poly1305` [NL18, B+08] that uses longer nonces (192 bits as opposed to 96 bits for `ChaCha20-Poly1305`). It uses an initial round of `ChaCha20` with the key and a portion of the nonce to derive a key for each message. `XChaCha20-Poly1305` is proven secure assuming `ChaCha20-Poly1305` is secure. `XChaCha20-Poly1305` was chosen over `ChaCha20-Poly1305` to further reduce the probability of randomly generated nonce being reused accidentally [Ber11]. While one might argue that the probability of the said nonce collision occurring is sufficiently low already, the probability may not be non-negligible at some point after more than a certain number of nonce was sampled and used (e.g., birthday paradox). One way to prevent a nonce collision is to have the protocol be 'stateful' and keep track of the used nonces, but this creates an additional 'burden' to the system (note that it would also need to be kept hidden somewhere as well), and can be difficult to implement and maintain for multi-party settings. Readers interested in a deeper dive into this topic may consult [DGGP21, BCV25], though it is not required for understanding this work.

3.3 `Ed25519` Digital Signatures

Digital signatures, similar to Message Authentication Codes (MACs), are used to authenticate the integrity of data. Unlike MACs, however, they have two separate keys: a *private* signing key to sign some message m and produce a *signature*, and a *public* verification key to check that the message has not been tampered with since signing.

Given a public-private keypair $(\mathsf{pk}, \mathsf{sk})$, signing a message m is described as follows:

$$\sigma \coloneqq \mathsf{Sign}_{\mathsf{sk}}(m)$$
$$\mathsf{Ver}_{\mathsf{pk}}(m, \sigma) \to \{0, 1\}$$

We decided to use Ed25519 (i.e., EdDSA over the Curve25519 Twisted Edwards elliptic curve) due to its robustness against various side-channel attacks [FA17, OKBNA23], weak random nonces [JL17], and key substitution attacks [BCJZ21] compared to other popular digital signature schemes such as ECDSA and RSA, and is also easily compatible with embedded systems. Curious readers can reference the RFC 8032 standards document [JL17] for more details on its construction.

3.4 `Argon2id` Password Hashing

A hash function $H : \{0, 1\}^* \to \{0, 1\}^\ell$ takes in an arbitrary-length output and compresses it into a fixed-length (ℓ-bit) output. Cryptographic hash functions must also satisfy a number of other useful properties, such as ensuring that very few collisions occur where two messages m, m' can hash to the same digest $H(m) = H(m')$, similar inputs cascade into drastically different outputs and, most importantly, the hash is a one-way function that an attacker cannot efficiently reverse. However, typical cryptographic hash functions such as SHA-2 are often designed to be computed *incredibly* quickly, hashing millions of times per second on a typical PC. For some purposes, such as passphrase verification, this is highly non-ideal, as an attacker who can know or guess candidate inputs can generate a *rainbow table* or *dictionary* of possible input-hash mappings in a matter of minutes, especially with access to highly parallelizable computations.

Our system uses `Argon2id` since it is a very well-established password-hashing scheme that provides memory-hardness guarantees and allows us to tune the time/memory-tradeoffs via customizable parameters to maximize work factor.

4 Satellite Protocols

4.1 Secret Parameters

4.1.1 Global Secrets

Several cryptographic keys will be generated when the deployment is built:

- $\mathsf{sub}_{\mathsf{sk}}$: 256-bit secret used to derive subscription encryption keys.
- $\mathsf{sub}_{\mathsf{priv}}$: 256-bit secret used to derive `Ed25519` private keys for signing and verifying subscriptions.
- $\mathsf{c0}_{\mathsf{sk}}$: `XChaCha20-poly1305` key used to encrypt all communication on channel 0.
- $\mathsf{c}i_{\mathsf{sk}}$: root key used to derive all encryption keys for channel $i \in \{1, \ldots, 8\}$.
- $\mathsf{c}i_{\mathsf{pub}}$, $\mathsf{c}i_{\mathsf{priv}}$: `Ed25519` public and private keys for signing frames on channel $i \in \{1, \ldots, 8\}$.

We assume no attacker can directly access the global secrets.

4.1.2 Secrets Stored on Decoder

When a decoder image is built, only some of the secrets from the global secrets will be stored on the decoder, and others will be derived from global secrets.

Assume DID is the 4 byte id of the decoder being built. The following cryoptographic keys will be stored on the decoder:

- $\mathsf{DID\_sub}_{\mathsf{sk}} = \texttt{Argon2}(\mathsf{salt} = \mathsf{sub}_{\mathsf{sk}}, \mathsf{DID})$: Key used to encrypt subscription data of this particular decoder.
- $\mathsf{DID\_sub}_{\mathsf{pub}} = \texttt{PublicKeyFor}(\texttt{private\_key} = \texttt{Argon2}(\mathsf{salt} = \mathsf{sub}_{\mathsf{priv}}, \mathsf{DID}))$: `Ed25519` public key used to verify subscription data signatures for this particular decoder.
- $\mathsf{c0}_{\mathsf{sk}}$: `XChaCha20-Poly1305` secret key used to decrypt data sent on channel 0.
- $\mathsf{c0}_{\mathsf{pub}}$: `Ed25519` public key used to verify signatures for frames sent on channel 0.

As stated in our threat model, we assume the hardware attacker has access to all these secrets.

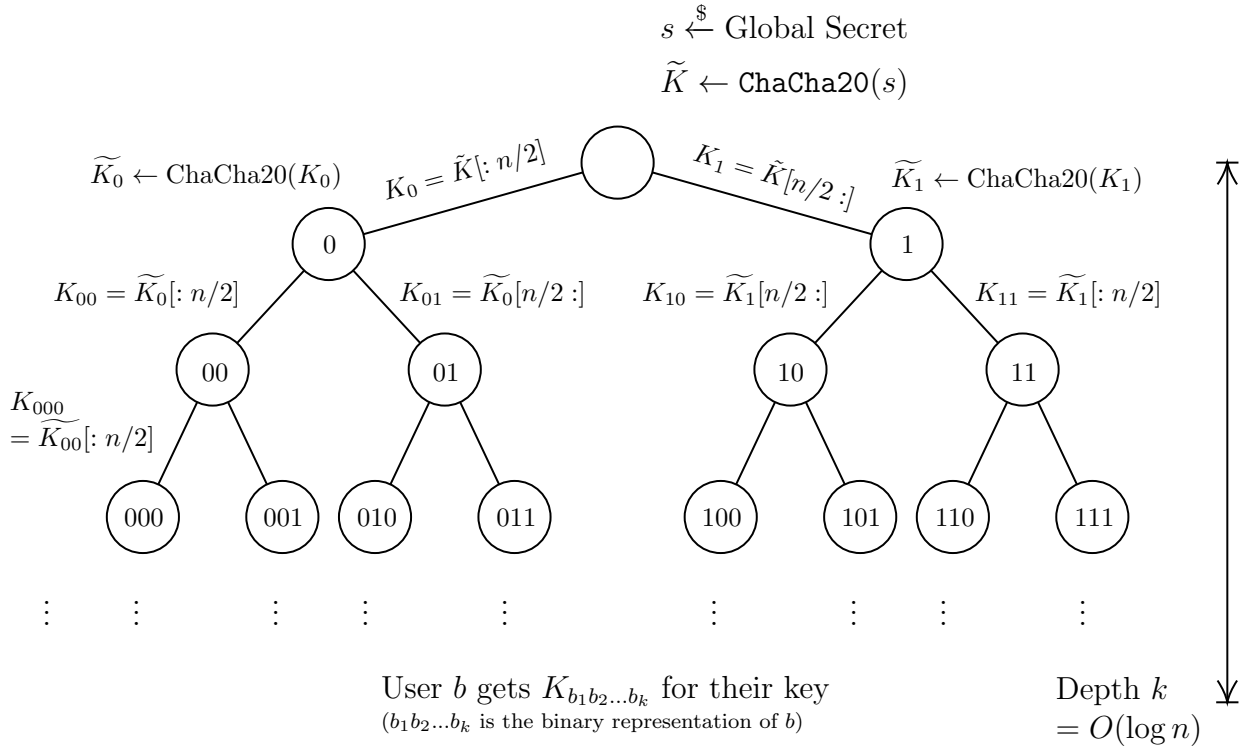
4.2 Key Derivation Tree

In order for a decoder to only be able to decrypt frames within its subscription, we wanted to derive a unique key for each timestamp on a channel, and have the data in a subscription only allow the decoder to generate keys within its subscription window.

To accomplish this, we use a construction similar to the Goldreich-Goldwasser-Micali (GGM) construction of a pseudorandom function (PRF) from a pseudorandom number generator (PRNG) [GGM86]. We have a tree structure with 64 levels, where every node has a 256-bit value. The children of each node can be obtained by feeding the 256-bit parent value as a

seed into the ChaCha20, which will output a 512-bit block; the role of ChaCha20 here is to serve as a cryptographically secure PRNG, as it is currently used across multiple domains, both in theory and in practice [Nau24, KW24, BSO+17]. We then use the first 256 bits as the left child node, and the last 256 bits as the right child node. The root of the tree is a random value generated in the global secrets, and the root value determines the value of every node in the tree.

To obtain a key for a given timestamp, we use the bits of the timestamp to trace a path from the root to a leaf node, and this 256 bit leaf value will be used as the key for a particular timestamp. To trace this path, we go bit by bit in the timestamp, and if the bit is 0, we go to the right child. If the bit is 1, we go to the left child. This means a key can be generated in $O(\log n)$, where n is the number of bits in a timestamp.



To restrict a decoder to only be able to generate keys for a certain region of timestamps, we can send several intermediate nodes in the subscription data, instead of the root node. The decoder could then generate keys only for timestamp values in one of the subtrees rooted at one of these internal nodes. To cover a given region of consecutive leaf nodes, it takes only $O(\log n)$ subtrees rooted at certain internal nodes, so $O(\log n)$ internal nodes can be sent in the subscription data to the decoder.

Due to the one-way nature of ChaCha20, it is practically impossible to recover the seed given an output block, otherwise the PRNG could be predicted. This means that given some internal nodes, it is not possible to recover their parent nodes, and therefore, it is not possible to recover any leaf nodes outside of the specified time range.

4.3 Decoder Payload Format

All sensitive data sent to the decoder is sent in a common format. The data is encrypted with `XChaCha20-Poly1305`, and the ciphertext, associated data, nonce and `Poly1305` tag is signed with `Ed25519`. It is important to sign the nonce as well otherwise an attacker with the symmetric key but not the private key could change the nonce, recaulculate `Poly1305` tag, but keep the same ciphertext, which would then result in a different decryption.

The keys used for encrypting and signing the payload vary depending on if it is a subscription or an encoded frame. In addition there is some associated data which is not encrypted, but is still verified by `Ed25519` and `Poly1305`. The content and size of this associated data varies depending on the type of payload.

A diagram of the decoder payload format is shown below.

Field	Size
`Ed25519` Signature	64 bytes
`XChaCha20-Poly1305` Nonce	24 bytes
`Poly1305` Tag	16 bytes
Ciphertext	*Varies*
Plaintext Associated Data	*Varies*

The following algorithm run on the encoder shows how to encrypt a decoder payload:

`PayloadEncrypt`(DATA, `associated_data`, `symmetric_key`, `signing_key`)

`nonce` $\leftarrow\!\!\$$ OSRNG()
(`ct`, `tag`) $\leftarrow$ $\text{Enc}_{\texttt{symmetric\_key}}$(DATA, `associated_data`, `nonce`)
`payload` $\leftarrow$ `nonce`$\|$`tag`$\|$`ct`$\|$`associated_data`
return $\text{Sign}_{\texttt{signing\_key}}$(`payload`)$\|$`payload`

The following algorithm run on the decoder shows how to decrypt and verify a decoder payload:

`PayloadDecrypt`(`payload`, `symmetric_key`, `verifying_key`)

`signature`$\|$`nonce`$\|$`tag`$\|$`ct`$\|$`associated_data` $\leftarrow$ `payload`
Verify $\text{Ver}_{\texttt{verifying\_key}}$(`nonce`$\|$`tag`$\|$`ct`$\|$`associated_data`, `signature`) = True
return $\text{Dec}_{\texttt{symmetric\_key}}$(`ct`, `associated_data`, `tag`, `nonce`)

4.4 Subscriptions

The subscription should contain the necessary data for only a particular decoder to decrypt frames in a given region of timestamps on a channel. This can be accomplished by sending several intermediate GGM nodes as described before. The public key of the channel is also sent.

We also send various bookkeeping information, such as the start timestamp, and the channel number of the subscription.

The subscription data is encrypted and signed in the common decoder payload format. We use DID_sub_{sk} as the symmetric key, and DID_sub_{pub} and DID_sub_{priv} as the public / private signing keypair. Since DID_sub_{sk} is symmetric key known only to the target decoder, it ensures no other decoders can decrypt the subscription.

In order to persist subscriptions across resets, we store the decrypted and processed subscription data in flash. We reserve 1 flash page for each subscription, which means we reserve 8 flash pages in total, for the maximum possible number of channels subscribed at once which is 8.

4.5 Frame Encoding & Decoding

Frames are encrypted and signed with the common decoder payload format. The raw data of the frame passed to the encoder is encrypted to ciphertext, while the channel number and timestamp remain unencrypted as associated data.

We use the per channel key derivation tree to derive a secret key based on the timestamp, and we use the per channel public-private key pair $\mathsf{c}i_{\mathsf{pub}}$ and $\mathsf{c}i_{\mathsf{priv}}$ to sign and verify the payload.

We also keep track of the last decoded frame in memory to ensure we decode frames in a strictly monotonically increasing order.

The encoder will use the following procedure to encode a frame, where $\texttt{Channel}_{\texttt{id}}$ is the channel number, t is the timestamp, and DATA is the data to encode:

$\texttt{Encode}(\texttt{Channel}_{\texttt{id}}, t, \mathsf{DATA})$

if $\texttt{Channel}_{\texttt{id}} = 0$ **then**
  $\texttt{key} \leftarrow \mathsf{c}0_{\mathsf{sk}}$
else
  $\texttt{key} \leftarrow \texttt{key\_derive}(\mathsf{c}i_{\mathsf{sk}}, t)$

return PayloadEncrypt(
  DATA,
  $\texttt{associated\_data} = \texttt{Channel}_{\texttt{id}} \| t$,
  $\texttt{symmetric\_key} = \texttt{key}$,
  $\texttt{signing\_key} = \mathsf{c}i_{\mathsf{priv}}$,
)

The decoder will then decode the frames as follows:

```
Decode(frame‖signature)
─────────────────────────────────────────────────────
Channel_id‖t ← frame.associated_data

Verify Last Decoded Timestamp < t
  and t is within subscribed range of time

if Channel_id = 0 then
  key ← c0_sk
else
  Verify we can derive key with timestamp t
  subscribedroots ← get_subscription_data(Channel_id)
  key ← key_derive(subscribedroots, t)

plaintext ← PayloadDecrypt(
  frame,
  symmetric_key = key,
  verifying_key = ci_pub,
)
Last Decoded Timetamp ← t

return plaintext
```

4.6 Security Analysis

Even if we assume that a powerful hardware attacker can read flash memory or effect the board in any way, we believe our design should keep all 5 flags safe. Flags should be safe in each scenario as outlined below:

(AS1) Expired Subscription: An attacker cannot derive keys for timestamps outside the subscription window because they do not have any of the intermediate GGM nodes corresponding to the new timestamps. So, they cannot predict output of the PRF, which means they can't generate the key for any timestamp outside the subscription window. This means they cannot decrypt any of the new frames with the flags.

(AS2) Pirated Subscription: An attacker cannot decrypt the subscription data because they do not have the key for the decoder used to encrypt the subscription data.

(AS3) No Subscription: An attacker cannot derive any keys if they do not have any subscription data with GGM tree nodes.

(AS4) Recording Playback: For same reason as the expired subscription scenario explained above, an attacker cannot derive a key for frames outside of the subscription

window.

(AS5) Pesky Neighbor: An attacker could derive the symmetric key and encrypt an invalid frame, but they do not have the private key to sign the frame. Since they do not have physical access to the remote decoder, they cannot perform a hardware attack to potentially bypass a signature check. The only issue would be if an attacker could get remote code execution, then they could print out an invalid decoded frame over UART.

5 Additional Security Features

In order to ensure secure operation even while attackers have physical access to the board, we have extensively explored hardware security protections and countermeasures to "harden" the embedded system against side-channel, injection, and other hardware attacks:

- We utilized the Rust programming language to help protect against various memory safety bugs which could give an attacker code execution. We tried to minimize our usage of unsafe code to further reduce the probability of such a bug being present in our design.
- Implemented ASLR-like memory layout randomization by randomizing offsets in the firmware binary to make it harder to perform attacks on our system.
- Utilized the MAX78000 boards's memory protection unit, in order to make sure there are no RWX memory sections an attacker could execute shellcode in.
- Employed audited, constant-time implementations of cryptography and other downstream-dependant libraries to ensure that it will be harder to limit the ability for attackers to leverage hardware-level timing and other side-channel attacks; and

References

[Analog] Analog Devices, *Max78000fthr: Evaluation kit for the max78000*, Official webpage of Analog Devices containing overview of the evaluation board MAX78000FTHR and official documentations.

[Arc20] Scott Arciszewski, *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*, Tech. Report `draft-irtf-cfrg-xchacha-03`, Internet Engineering Task Force (IETF), 01 2020, Last updated: 2023-05-02.

[B+08] Daniel J Bernstein et al., *Chacha, a variant of salsa20*, Workshop record of SASC, vol. 8, 2008, pp. 3–5.

[BCJZ21] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao, *The Provable Security of Ed25519: Theory and Practice*, 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 1659–1676.

[BCV25] Tim Beyne, Yu Long Chen, and Michiel Verbauwhede, *A robust variant of ChaCha20-poly1305*, Cryptology ePrint Archive, Paper 2025/222, 2025.

[Ber11] Daniel J Bernstein, *Extending the salsa20 nonce*, Workshop record of Symmetric Key Encryption Workshop, vol. 2011, 2011.

[BSO+17] Séamus Brannigan, Neil Smyth, Tobias Oder, Felipe Valencia, Elizabeth O'Sullivan, Tim Güneysu, and Francesco Regazzoni, *An investigation of sources of randomness within discrete gaussian sampling*, Cryptology ePrint Archive (2017).

[DGGP21] Jean Paul Degabriele, Jérôme Govinden, Felix Günther, and Kenneth G Paterson, *The security of chacha20-poly1305 in the multi-user setting*, Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 1981–2003.

[FA17] Hayato Fujii and Diego F. Aranha, *Curve25519 for the Cortex-M4 and Beyond*, Progress in Cryptology – LATINCRYPT 2017 (Tanja Lange and Orr Dunkelman, eds.), 2017, pp. 109–127.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali, *How to construct random functions*, Journal of the ACM (JACM) **33** (1986), no. 4, 792–807.

[JL17] Simon Josefsson and Ilari Liusvaara, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, RFC 8032, January 2017.

[KW24] Bong Gon Kim and Dennis Wong, *Smart contract-based secure verifiable random function using chacha20 sequence in blockchain*, Proceedings of the 2023 5th International Conference on Blockchain Technology (New York, NY, USA), ICBCT '23, Association for Computing Machinery, 2024, p. 41–51.

[eCTF25] MITRE Corporation, *2025 eCTF: The 2025 MITRE Engenuity Embedded Capture the Flag Competition*, https://rules.ectf.mitre.org/about/index.html.

[Nau24] Mykhailo Naumenko, *Cryptographically secure pseudorandom number generators*, Master's thesis, Univerzita Karlova, Matematicko-fyzikální fakulta, 2024.

[NL18] Yoav Nir and Adam Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, 06 2018.

[OKBNA23] Daniel Owens, Rabih El Khatib, Mojtaba Bisheh-Niasar, and Reza Azarderakhsh, *Efficient and Side-Channel Resistant Ed25519 on ARM Cortex-M4*, Safety and Security in Heterogeneous Open System-on-Chips Platform Workshop – SSH-SoC 2023, 2023.

A Terminology and Notation

Abbreviations

AE Authenticated Encryption.

AEAD Authenticated Encryption with Associated Data.

AES Advanced Encryption Standard.

API Application Programming Interface.

AS Attack Scenario (aka Attack Phase in eCTF rules).

ASLR Address Space Layout Randomization.

C&C Command and Control.

CR Challenge-Response.

eCTF Embedded Capture the Flag.

Ed25519 Edwards-curve Digital Signature Algorithm over Curve25519.

GGM Pseudorandom function construction of Goldreich, Goldwasser, and Micali.

HMAC (Hash-based) Message Authentication Code.

HW Hardware.

I^2C Inter-Integrated Circuit.

ID Identification Number.

N/A Not Applicable.

NIST National Institute of Standards and Technology.

PKI Public-Key Infrastructure.

RFC Request for Comments.

SHA Secure Hash Algorithms.

SR Security Requirement.

SSL Secure Socket Layer.

TLS Transpoprt Layer Security.

UART Universal Asynchronous Receiver-Transmitter.

List of Symbols

$\circ'$ Different or modified $\circ$.
$\overline{\circ}$ List of $\circ$.
$\leftarrow$ Assign equal to.
$\leftarrow\!\!\$$ Assign from random sample to.
$=$ Compare equal to.
$\neq$ Compare not equal to.
$:=$ Defined equal to.
$\{\ldots\}$ Set.
$\{\ldots\}^*$ Kleene star (≥ 0 repetitions in set).
$\|$ (Byte)string concatenation.
$\in$ (Is contained) in.
$\notin$ Not in.
$\oplus$ XOR.
c Ciphertext.
H Cryptographic hash function.
k Cryptographic key.
ℓ Length (in Bytes).
m, M Message or plaintext.
n Number used once ("nonce").
pk Public key.
r Random number.
σ Signature.
sk Secret/signing key.
τ HMAC Tag.