Friet: An Authenticated Encryption Scheme with Built-in Fault Detection

Thierry Simon[1,2] Lejla Batina[1] Joan Daemen[1] Vincent Grosso[1,3]
Pedro Maat Costa Massolino[1] Kostas Papagiannopoulos[1,4]
Francesco Regazzoni[5] Niels Samwel[1]

[1]Radboud University [2]STMicroelectronics [3]CNRS/Univ. Lyon
[4]NXP Semiconductors Hamburg [5]University of Lugano

Eurocrypt, May 2020

Motivation

Lightweight cryptographic primitive

Motivation

Lightweight cryptographic primitive with fault detection

- Using an Error Detecting Code (EDC)
  - ParTI [Schneider et al., ePrint 2016/648]
  - `CRAFT` [Beierle et al., ToSC 2019]

Motivation

Lightweight cryptographic primitive with built-in fault detection

- Using an Error Detecting Code (EDC)
  - ParTI [Schneider et al., ePrint 2016/648]
  - `CRAFT` [Beierle et al., ToSC 2019]
- Designed with building blocks efficiently adaptable to a specific EDC

Designing a Permutation for a Specific Error Detecting Code

The Parity Code $[4, 3, 2]$

- Set of encodable messages: $\mathbb{Z}_2^3$
- Set of codewords: C contains all $(a, b, c, d) \in \mathbb{Z}_2^4$ satisfying the parity equation

$$a \oplus b \oplus c \oplus d = 0$$

- Encoding function: $Enc : \mathbb{Z}_2^3 \to C, (a, b, c) \mapsto (a, b, c, a \oplus b \oplus c)$
- Decoding function:

$$Dec : \mathbb{Z}_2^4 \to \mathbb{Z}_2^3, (a, b, c, d) \mapsto \begin{cases} (a, b, c) \text{ if } a \oplus b \oplus c \oplus d = 0, \\ \perp \text{ else} \end{cases}$$

Building a $[4, 3, 2]$-abiding Permutation

Let π be a 384-bit permutation whose state (a, b, c) is divided in three 128-bit limbs. We define $\bar{\pi} : (a, b, c, d) \mapsto (a', b', c', d')$ a 512-bit permutation whose state is divided in four limbs such that

- $\bar{\pi}$ preserves the parity: $\bar{\pi}(C^{128}) = C^{128}$ or equivalently

$$a \oplus b \oplus c \oplus d = 0 \iff a' \oplus b' \oplus c' \oplus d' = 0.$$

  We say that $\bar{\pi}$ abides the parity code.

- $\bar{\pi}$ extends π: if $(a, b, c, d) \in C^{128}$ then

$$\bar{\pi}(a, b, c, d) = (a', b', c', d') \Rightarrow \pi(a, b, c) = (a', b', c').$$

  We say that π is the embedding of $\bar{\pi}$ by the parity code.

Building a $[4, 3, 2]$-abiding Permutation

Let π be a 384-bit permutation whose state (a, b, c) is divided in three 128-bit limbs. We define $\bar{\pi} : (a, b, c, d) \mapsto (a', b', c', d')$ a 512-bit permutation whose state is divided in four limbs such that

- $\bar{\pi}$ preserves the parity: $\bar{\pi}(C^{128}) = C^{128}$ or equivalently

$$a \oplus b \oplus c \oplus d = 0 \iff a' \oplus b' \oplus c' \oplus d' = 0.$$

  We say that $\bar{\pi}$ abides the parity code.

- $\bar{\pi}$ extends π: if $(a, b, c, d) \in C^{128}$ then

$$\bar{\pi}(a, b, c, d) = (a', b', c', d') \Rightarrow \pi(a, b, c) = (a', b', c').$$

  We say that π is the embedding of $\bar{\pi}$ by the parity code.

Building a $[4, 3, 2]$-abiding Permutation

Let π be a 384-bit permutation whose state (a, b, c) is divided in three 128-bit limbs. We define $\bar{\pi} : (a, b, c, d) \mapsto (a', b', c', d')$ a 512-bit permutation whose state is divided in four limbs such that

- $\bar{\pi}$ preserves the parity: $\bar{\pi}(C^{128}) = C^{128}$ or equivalently

$$a \oplus b \oplus c \oplus d = 0 \iff a' \oplus b' \oplus c' \oplus d' = 0.$$

  We say that $\bar{\pi}$ abides the parity code.

- $\bar{\pi}$ extends π: if $(a, b, c, d) \in C^{128}$ then

$$\overline{\pi}(a, b, c, d) = (a', b', c', d') \Rightarrow \pi(a, b, c) = (a', b', c').$$

  We say that π is the embedding of $\bar{\pi}$ by the parity code.

Detecting Faults

In order to compute $\pi(a, b, c)$

1. Initialize the parity limb d as $d = a \oplus b \oplus c$.
2. Compute $\bar{\pi}(a, b, c, d) = (a', b', c', d')$.
3. Verify the parity equation $a' \oplus b' \oplus c' \oplus d' = 0$.
4. If the parity holds return (a', b', c'), else a fault is detected.

And Concretely?

The permutation π is split as a sequence of step functions where either

- a single limb is modified by XORing a function ϕ of the state, e.g.

$$f(a, b, c) = (a \oplus \phi(a, b, c), b, c)$$

- two limbs are permuted, e.g. $g(a, b, c) = (b, a, c)$.

Each step function can then be extended by:

Limb adaptation Recomputing ϕ and XORing the result to d

$$\bar{f}(a, b, c, d) = (a \oplus \phi(a, b, c), b, c, d \oplus \phi(a, b, c)).$$

Limb transposition Reordering the limbs

- Non-native: if $\phi(a, b, c) = b \oplus c$ then $\bar{f}(a, b, c, d) = (d, b, c, a)$.
- Native: $\bar{g}(a, b, c, d) = (b, a, c, d)$.

And Concretely?

The permutation π is split as a sequence of step functions where either

- a single limb is modified by XORing a function ϕ of the state, e.g.

$$f(a, b, c) = (a \oplus \phi(a, b, c), b, c)$$

- two limbs are permuted, e.g. $g(a, b, c) = (b, a, c)$.

Each step function can then be extended by:

Limb adaptation Recomputing ϕ and XORing the result to d

$$\bar{f}(a, b, c, d) = (a \oplus \phi(a, b, c), b, c, d \oplus \phi(a, b, c)).$$

Limb transposition Reordering the limbs

- Non-native: if $\phi(a, b, c) = b \oplus c$ then $\bar{f}(a, b, c, d) = (d, b, c, a)$.
- Native: $\bar{g}(a, b, c, d) = (b, a, c, d)$.

And Concretely?

The permutation π is split as a sequence of step functions where either

- a single limb is modified by XORing a function ϕ of the state, e.g.

$$f(a, b, c) = (a \oplus \phi(a, b, c), b, c)$$

- two limbs are permuted, e.g. $g(a, b, c) = (b, a, c)$.

Each step function can then be extended by:

Limb adaptation Recomputing ϕ and XORing the result to d

$$\bar{f}(a, b, c, d) = (a \oplus \phi(a, b, c), b, c, d \oplus \phi(a, b, c)).$$

Limb transposition Reordering the limbs

- Non-native: if $\phi(a, b, c) = b \oplus c$ then $\bar{f}(a, b, c, d) = (d, b, c, a)$.
- Native: $\bar{g}(a, b, c, d) = (b, a, c, d)$.

Fault Model

What are the fault detection capabilities of the code-abiding permutation?

- ✓ Single limb fault
  - Any simple fault affecting only one limb is guaranteed to be caught.
- ✗ Other simple fault
  - e.g. flipping the i-th bit of two different limbs will not break the parity equation
- ✗ Multiple faults
  - e.g. injecting the same fault in the two computations of ϕ during a limb adaptation

⚠ Extra care must be taken when verifying the parity equation since the countermeasure does not cover the verification itself!

Checking the Parity Equation

Verifying the parity equation after each step function or after each round is not worth it. A single check at the end of the permutation is enough:

- An attacker that is capable of injecting multiple compensating faults over different step functions should also be able to inject them in the same step function.
- Frequent checks impact the performance.

Friet

Friet

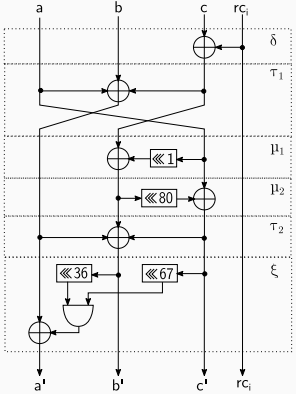
- Fault-Resistant Iterative Extended Transformation
- Authenticated encryption scheme
  - Duplex construction [Bertoni et al., Selected Areas in Cryptography, 2011]
  - SpongeWrap mode
- FRIET-PC: the 384-bit underlying cryptographic permutation
- FRIET-P: the 512-bit implemented permutation abiding the parity code $[4, 3, 2]$

Friet-PC pseudocode

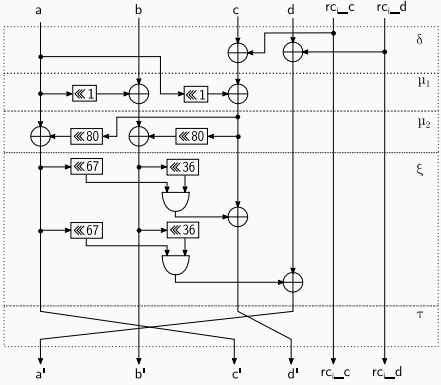
- State: three 128-bit limbs (a, b, c)
- 24 rounds from $i = 0$ to 23, with a 6-step round function:

$$
\begin{aligned}
\delta :&\quad c \leftarrow c \oplus rc_i && \triangleright \textit{ round constant addition}\\
\tau_1 :&\quad (a, b, c) \leftarrow (a \oplus b \oplus c, c, a) && \triangleright \textit{ transposition}\\
\mu_1 :&\quad b \leftarrow b \oplus (c \lll 1) && \triangleright \textit{ mixing step}\\
\mu_2 :&\quad c \leftarrow c \oplus (b \lll 80) && \triangleright \textit{ mixing step}\\
\tau_2 :&\quad b \leftarrow a \oplus b \oplus c && \triangleright \textit{ transposition}\\
\xi :&\quad a \leftarrow a \oplus ((b \lll 36) \wedge (c \lll 67)) && \triangleright \textit{ non-linear step}
\end{aligned}
$$

From Friet-PC to Friet-P



FRIET-PC round function



FRIET-P round function

Performances

Hardware (ASIC)

AE Scheme	Area (GE)	Freq. (MHz)	Throu. (Mb/s)	Power (μW) static	dynamic
Ketje-Sr [1]	9478	503	16096	161	2152
FRIET-C (1R/Cy)	6943	508	2322	110	1724
FRIET (1R/Cy)	9253	508	2322	148	2226
FRIET-C (2R/Cy)	8890	508	4064	141	1737
FRIET (2R/Cy)	11100	508	4064	174	2245

ASIC Nangate 45 nm standard cell

[1] [Bertoni et al., Caesar submission, 2016]

Software

Permutation	Rounds	Cycles/byte	Cycles/byte per round
Xoodoo [2]	12	13.20	1.10
Friet-PC	24	17.78	0.74
Gimli [3]	24	21.81	0.91
Friet-P	24	24.23	1.01

ARM® Cortex-M3/M4

- Competitive performances with Xoodoo and Gimli
- Friet-P is 36% slower then Friet-PC, mainly due to the additional load and store instructions

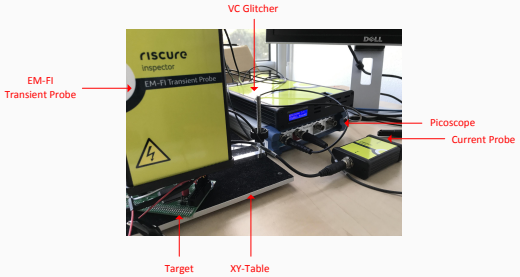
[2] [Daemen et al., ToSC 2018]
[3] [Bernstein et al., CHES 2017]

Fault Resistance Evaluation

Hardware Simulation

- Experiment: Injection of single-bit glitches on a simulated hardware implementation of FRIET-P
  - At RTL level
  - After synthesis
- Result: 100% fault detection rate

EM Fault Injection in Software



- Experiment: Electromagnetic fault injection on a single round implementation of FRIET-P on ARM® Cortex-M4
  - Chip divided as 100×100 grid
  - 10 grid scans with 10 faults injected per position

Result	Normal	Reset	Detected	Undetected
Number	860488	138916	596	0

Conclusion

Conclusion

In this presentation, we saw:

- New design approach for crypto primitives: choosing building blocks that can be efficiently adapted to abide a specific EDC
- Illustration with FRIET for the $[4, 3, 2]$ parity code
  - Detection of single limb faults
  - Competitive performance in hardware and software

More in the paper about:

- Generalization to larger codes
- The authenticated encryption scheme
- Cryptographic properties: diffusion, algebraic degree, trails, ...
- Addressing Statistical Ineffective Fault Attacks

Thanks for your attention!

- Friet Paper
  `https://eprint.iacr.org/2020/425`
- Implementations
  `https://github.com/thisimon/Friet.git`