Hardware-Entangled Software Execution using Dynamic PUFs

Wenjie Xiong
Yale University, wenjie.xiong@yale.edu

Hardware-Entangled Software Execution using Dynamic PUFs

Wenjie Xiong[1], André Schaller[2], Stefan Katzenbeisser[2], and Jakub Szefer[1]

[1]Computer Architecture and Security Laboratory, Yale University, USA
[2]Security Engineering Group, Technische Universität Darmstadt, Germany

caslab.csl.yale.edu

www.seceng.informatik.tu-darmstadt.de

1. Hardware-Entangled Software Execution (HESE)

The goal of the Hardware-Entangled Software Execution (HESE) protection scheme is to protect software from malicious modification and to bind the execution of the software to a specific device that it was compiled for.

Protection Mechanisms:

- *Tamper detection:*
  - ***Dynamic PUF*** binds the execution to an authorized device and detects the abnormal timing of software execution.
  - ***Self-checksum*** **function** checks the integrity of the code segment. And multiple checksum functions will check each other mutually.
- *Tamper response:*
  - ***Call graph scrambling code*** decides the callee function based on the checksum, the PUF response and a reference value.
  - ***Register value scrambling code*** changes the value in the registers if the PUF response and the checksum match a reference value.

Contributions:

- We introduce the notion of Dynamic PUFs which have time-dependent responses.
- We develop a practical Dynamic PUF with a controlled PUF interface in kernel module and the DRAM on Intel Galileo Gen 2 platform.
- We propose a linear checksum function on a prime finite field, which allows for mutual authenticating of self-checksumming functions with low overhead.
- We implement a fully automatic framework that deploys the HESE protection scheme to the existing unprotected source code through an AutoPatcher written in Python that works on LLVM assembly.

2. Dynamic PUFs

Physically Unclonable Functions (PUFs) extract the unique and stable physical features from physical objects. Given a challenge, a PUF instance can generate a stable response, which is a function of the challenge and the physical features.

Dynamic PUFs have time-dependent PUF responses, i.e., the response depends not only on the challenge but also on the time of the PUF query.

DRAM PUFs are an example of Dynamic PUFs. As shown in Fig. 1, after reset at T_0, the PUF will generate different responses for queries at T_1 and T_2.

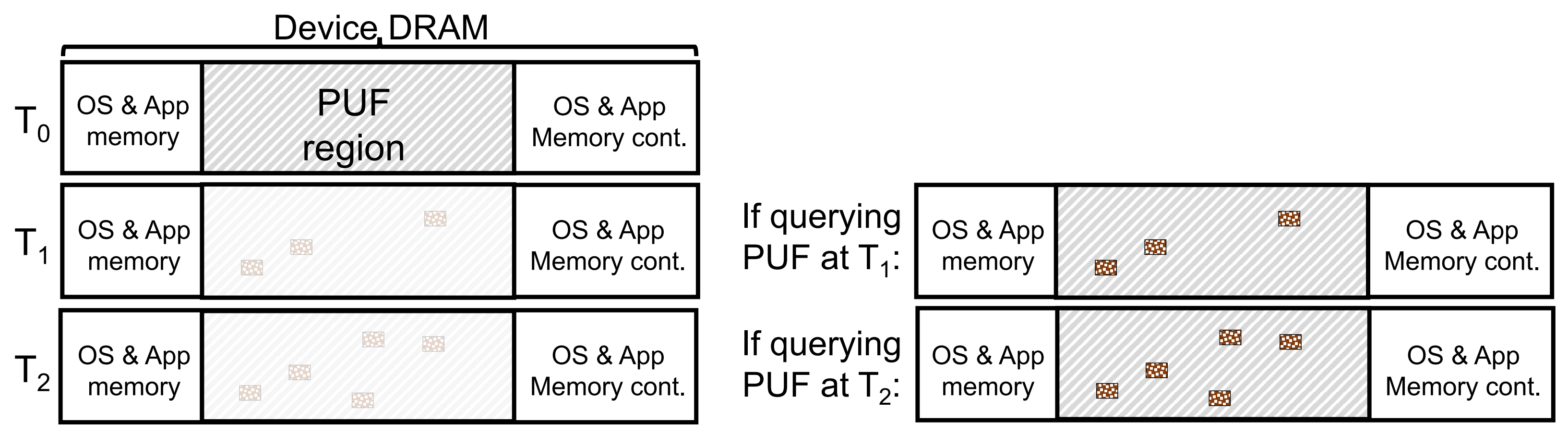

Fig 1. Time-dependent DRAM PUF responses.

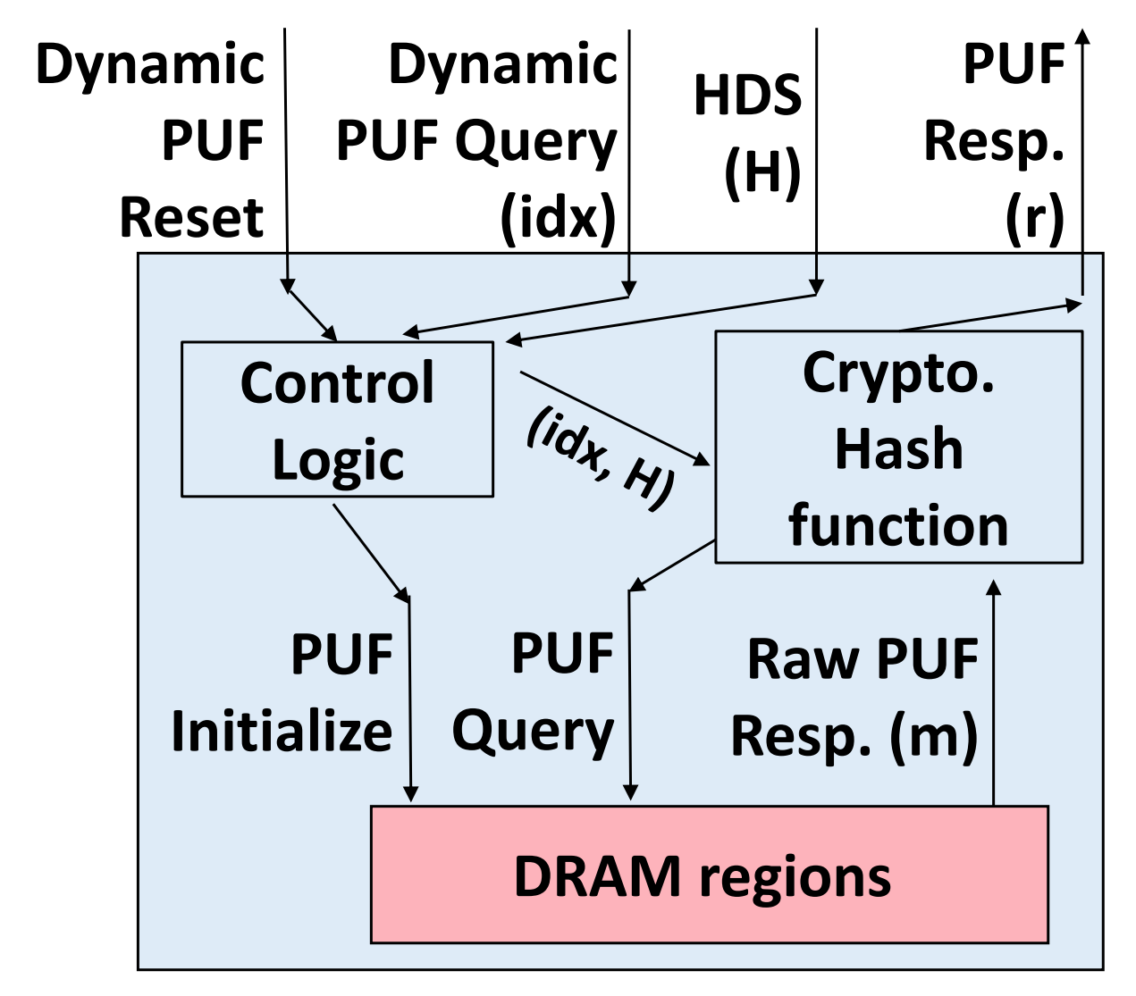

Fig 2. Controlled PUF interface to DRAM PUF.

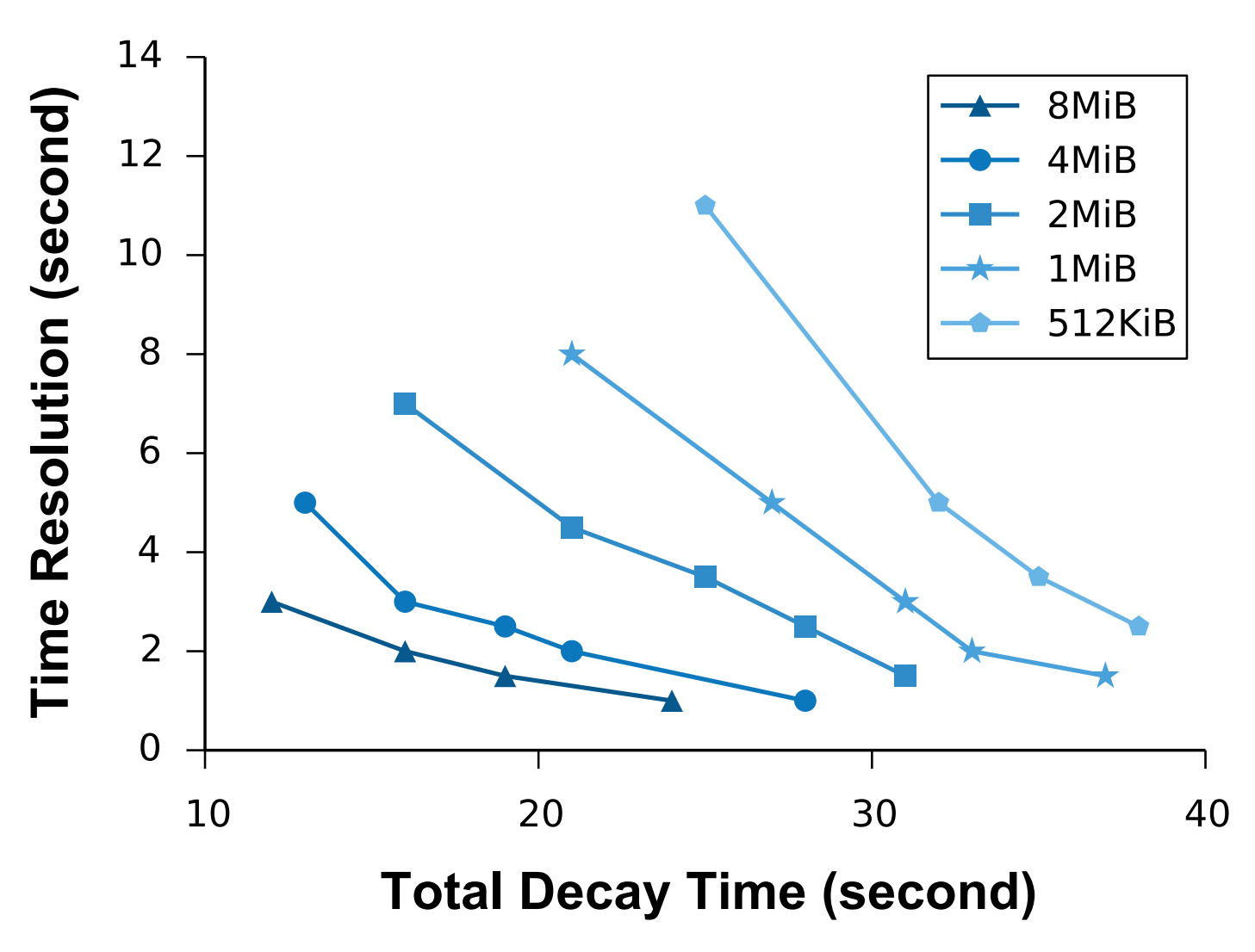

Fig 3. Time Resolution of DRAM PUFs.

HDS: As PUF measurements always have noise, a ***Helper Data System (HDS)*** is needed to help correct the error in the PUF measurements. And to make a PUF query, ***idx*** should be given to indicate which DRAM region and HDS to use.

Controlled PUF*:* DRAM PUFs is a weak PUF, meaning attackers can exhaustively measure all possible PUF responses. A Controlled PUF interface is needed to prevent the attacker from directly access the raw DRAM PUF responses (Fig. 2).

The time resolution of DRAM PUF is the smallest d_t such that the difference between responses at T_1 and T_1+d_t are bigger than the noise of PUF responses measured at T_1 repeatedly (Fig. 3).

3. Linear Checksum Function on a Finite Field

Given code segment $\mathbf{D=[d_1, d_2, \ldots, d_n]}$, where $\mathbf{d_i}$ is the $\mathbf{i^{th}}$ 32-bit word in $\mathbf{D}$, the linear checksum function is defined as $\mathbf{h_0(D)=0}$ and $\mathbf{h_i(D)= d_i+c*h_{i-1}(D)}$, for $\mathbf{0<i\le n}$, where $\mathbf{c}$ is a parameter. The addition and multiplication are over the $\mathbf{2^{32}}$**-5 prime field**, which has lightweight implementations with integer arithmetic on 32-bit processors.

The checksum is also equal to $\mathbf{h(D)=\sum_i l_i * d_i}$, where $\mathbf{l_i=c^{n-i+1}}$ is the multiplier for $\mathbf{d_i}$.

4. Protecting Software with Different Control Flows

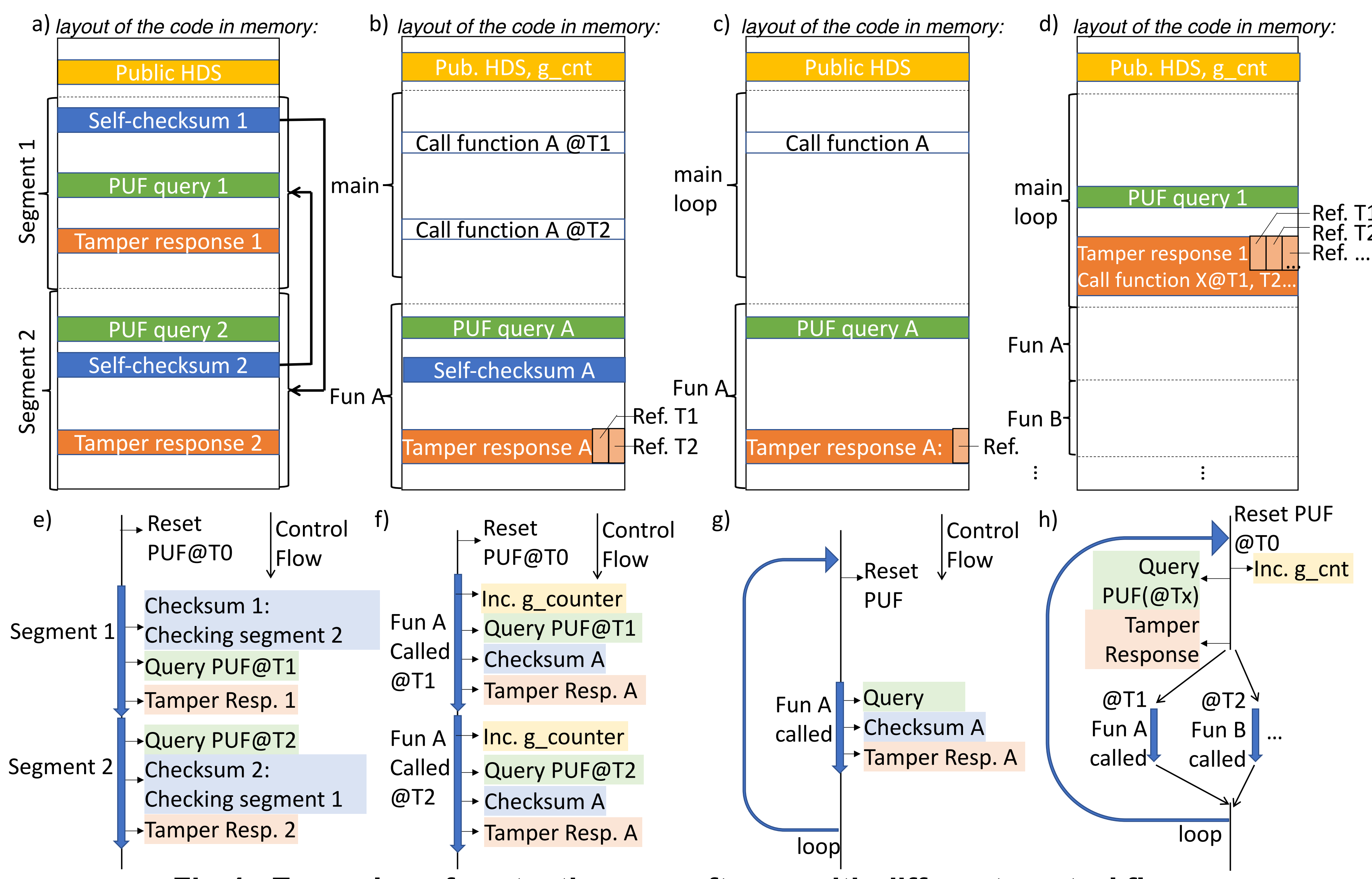

Fig 4. Examples of protection on software with different control flow.

Straight-line code (Fig. 4. a, e): The tamper response function will change the program behavior depending on the checksum and PUF response. As shown in the Fig. 4, The self-checksumming functions check each other mutually.

Repeated function calls (Fig. 4. b, f): If the PUF query is called several times, Dynamic PUF response will be different for each query. So a reference value is needed for each query. A global counter points to the reference value to use.

Function call in a loop (Fig. 4. c, g): Dynamic PUF can be reset at the beginning of a loop. And the PUF query will return the same response in every iteration.

Time-dependent control flows (Fig. 4. d, h): In a loop, a different function can be called by the tamper response function based on the time-dependent Dynamic PUF response.

5. Deployment of Protection

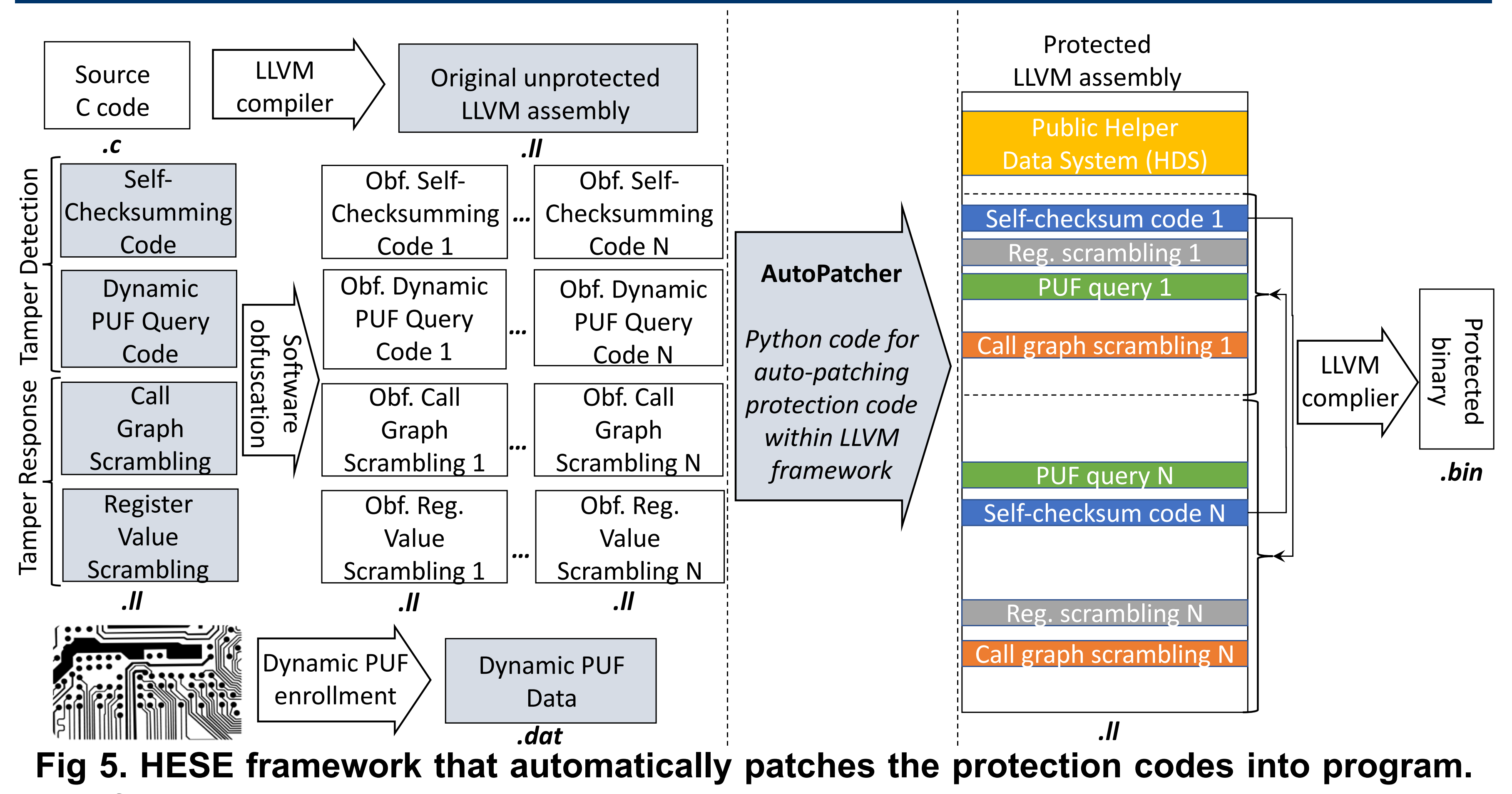

Fig 5. HESE framework that automatically patches the protection codes into program. The framework takes the source code, the protection codes and PUF enrollments as inputs and the AutoPatcher applies the protection to LLVM assembly.

Steps of AutoPatcher to apply the protection scheme:

1. Assign places to insert protection codes, assign code segments to checksum functions, and get expected PUF responses.
2. Pre-patch protection code and compute checksum value.
3. Solve reference values for all tamper response function and patch.

An AutoPatcher is implemented in Python, and the protection scheme is applied to AES and SHA applications of different control flow types. To patch the sample programs, it takes about 2.6 seconds. The runtime overhead is shown in Fig. 6. The **overlap factor** is the number of checksum functions that check the same code segment.

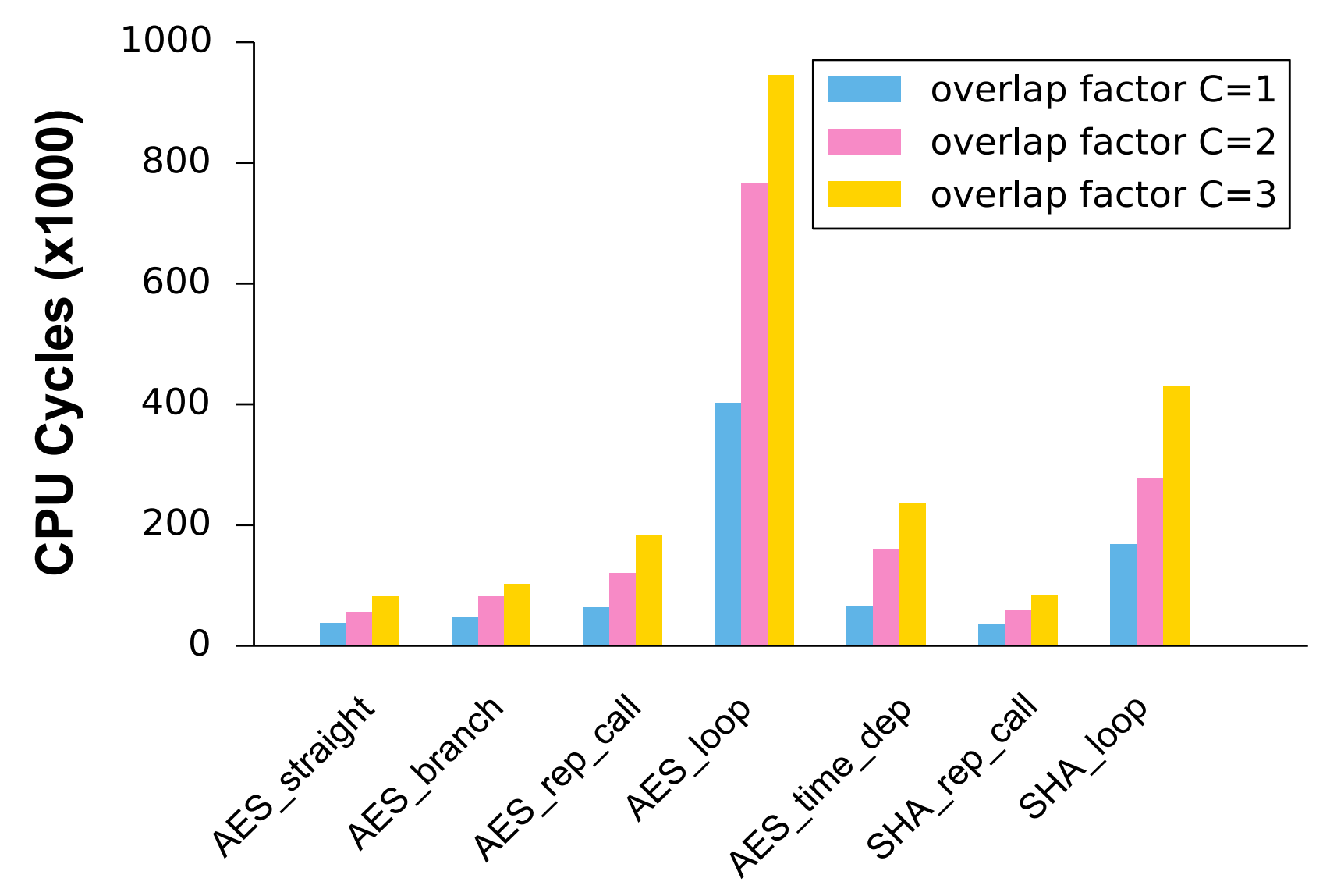

Fig 6. Overhead of the protection codes.