IACR Transactions on Symmetric Cryptology
ISSN 2519-173X, Vol. 2021, No. 3, pp. 170–217. DOI:10.46586/tosc.v2021.i3.170-217

PLCrypto: A Symmetric Cryptographic Library for Programmable Logic Controllers

Zheng Yang[1], Zhiting Bao[2], Chenglu Jin[3], Zhe Liu[4*] and Jianying Zhou[5]

[1] Southwest University, Chongqing, China youngzheng@swu.edu.cn
[2] Chongqing Institute of Engineering, Chongqing, China bztbaozhiting@gmail.com
[3] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands chenglu.jin@cwi.nl
[4] Nanjing University of Aeronautics and Astronautics, Nanjing, China zhe.liu@nuaa.edu.cn
[5] Singapore University of Technology and Design, Singapore, Singapore
jianying_zhou@sutd.edu.sg

Abstract. Programmable Logic Controllers (PLCs) are control devices widely used in industrial automation. They can be found in critical infrastructures like power grids, water systems, nuclear plants, manufacturing systems, etc. This paper introduces *PLCrypto*, a software cryptographic library that implements lightweight symmetric cryptographic algorithms for PLCs using a standard PLC programming language called structured text (ST). To the best of our knowledge, *PLCrypto* is the *first* ST-based cryptographic library that is executable on commercial off-the-shelf PLCs. *PLCrypto* includes a wide range of commonly used algorithms, totaling ten algorithms, including one-way functions, message authentication codes, hash functions, block ciphers, and pseudo-random functions/generators. *PLCrypto* can be used to protect the confidentiality and integrity of data on PLCs *without additional hardware or firmware modification*. This paper also presents general optimization methodologies and techniques used in *PLCrypto* for implementing primitive operations like bit-shifting/rotation, substitution, and permutation. The optimization tricks we distilled from our practice can also guide future implementation of other computation-heavy programs on PLCs. To demonstrate a use case of *PLCrypto* in practice, we further realize a cryptographic protocol called *proof of aliveness* as a case study. We benchmarked the algorithms and protocols in *PLCrypto* on a commercial PLC, Allen Bradley ControlLogix 5571, which is widely used in the real world. Also, we make our source codes publicly available, so plant operators can freely deploy our library in practice.

Keywords: Programmable Logic Controllers · Industrial Automation · Symmetric Cryptography · Cryptographic Library

1 Introduction

It is indisputable that the Industrial Internet of Things (IIoT) adoption in industrial control systems or critical infrastructures has excellent potential in the future. The concept of IIoT allows all components in such systems to be connected and coordinated intelligently and efficiently. Research conducted by Morgan Stanley and Automation World Magazine in 2015 predicted that the global market of IIoT would grow to 123 billion USD in 2021 [Sta16]. However, according to the same report, the manufacturers expressed their concerns about cybersecurity and their legacy-installed base [Sta16]. This reality imposes a vital question for us: how to secure legacy industrial systems? In this paper, we will

*Corresponding Author

Received: 2021-06-01 Accepted: 2021-08-01 Published: 2021-09-17

show our solution as a significant step towards solving this problem. In particular, we will show **how to retrofit legacy programmable logic controllers (PLCs) to secure their communications against network attackers without additional hardware or new firmware**.

The cores of industrial control systems are PLCs that control the physical processes directly. Thus, PLCs are usually the primary targets for attackers to compromise. However, widely used commercial PLCs lack proper security protections, like encryption and authentication. For example, suppose attackers are somehow connected to the operational technology network. In that case, they can easily intercept and manipulate the communication (e.g., over Common Industrial Protocol (CIP)) between PLCs and supervisory control and data acquisition (SCADA) servers.

To formalize the security needs in the automation industry, ODVA drafted in 2015 the first version of a security specification, *CIP Security* [ODV19], for the communication of design automation devices, including PLCs. In the specification, ODVA highlighted the need for device authentication, data integrity, and data confidentiality. Following this specification, PLC vendors have started designing and producing new *CIP Security* capable devices. However, the reality is that there are still a huge number of legacy devices running in the field, and the average lifetime of devices in factories is around 20 years [Dec18]. This also means that it will take at least another 20 years for the manufacturers to fully adapt to today's technology. Even today, *CIP Security* capable PLCs are not mainstream products from leading PLC vendors. For example, in a document published in September 2019 by Rockwell Automation, one leading PLC vendor, they only have one PLC model, ControLogix 5580, that supports *CIP Security* [Bra19].

Our Solution. To support legacy PLCs in the real world with no extra cost, we propose to secure PLC communications by developing a comprehensive symmetric cryptographic library, *PLCrypto*, on the control logic layer. Note that the control logic program is running on top of the firmware of a PLC. It is the only layer that one PLC user (e.g., an operator of an industrial control system) can program to realize various control and computation functionalities.

We develop our cryptographic library on Allen Bradley PLCs from Rockwell Automation [Roc20] because it is one of the two leading PLC vendors (SIEMENS and Rockwell Automation) in the world, each of which has more than 20% market share in the global market of PLCs as of 2017 [Deu17]. Although the implementations we developed are specific for Allen Bradley PLCs, our library can be easily migrated to the PLCs from other vendors. It is because *PLCrypto* is developed in structured text (ST), which is one of the standard programming languages for PLCs defined in IEC-61131-3 [JT13].

Realizing cryptographic algorithms on PLCs is challenging. The main difficulties we encountered are:

1. To program PLCs, we have to use a particular set of programming languages defined in IEC-61131-3 [JT13], i.e., ladder diagram, function block diagram, structured text, sequential function chart, and instruction list (deprecated). We selected structured text for developing our library because other PLC programming languages are graphical programming languages and are not suitable for implementing complex arithmetic operations.

2. The programming model of structured text is very restricted, especially on Allen Bradley PLCs [Bra18a], e.g., it does not support pointers, bit-wise shifting/rotation operations. Therefore we have to use more expensive alternative methods to implement these operations in cryptographic algorithms.

3. PLCs are not ideal devices to store secrets because everyone connected to the PLCs over the network can read/write all the variables (or tags in PLC terminology) in

the PLCs. The attacker may exploit this feature of PLC to implement a family of attacks, which we jointly call *tag manipulation attacks.*

4. PLCs are primarily used to control physical processes, so they are not optimized for complex logical or arithmetic operations in cryptographic algorithms. During the process of implementing *PLCrypto*, we discovered a variety of optimization tricks. For example, we notice that we can easily access individual bits in variables on PLCs. Based on this observation, we introduced one trick called hard-coding to significantly improve the performance of certain cryptographic algorithms. For example, our hard-coding implementation of a one-way function (OWF) is 2× faster than the straightforward implementation.

PLCrypto includes cryptographic algorithms providing the security properties that are highly demanded in real-world applications, such as confidentiality, integrity, collision-resistance, and pseudo-randomness. Most of the included algorithms are either standardized by ISO/IEC or the state-of-the-art symmetric cryptographic algorithms that fit the PLC environment. *PLCrypto* encompasses Message Authentication Code (MAC) algorithm Chaskey [MMH+14], block ciphers PRESENT [BKL+07, PRE19], SPECK, and SIMON [BSS+13], and collision-resistant hash functions PHOTON [GPP11, JG16] and SPONGENT [BKL+13]. *PLCrypto* also contains some subset-sum based cryptographic primitives as efficient alternatives, like one-way functions (OWF) and universal one-way hash functions (UOWHF). In the meantime, we realize some basic operations (such as big integer addition and subtraction) to support our implementation of block ciphers. Note that the resilience to potential sophisticated side-channel attacks is not considered in this paper, and it is one of the future works.

To demonstrate how one can extend the application of *PLCrypto* beyond communication security, we use *PLCrypto* to implement a proof of aliveness (PoA) protocol [JYvDZ19] on an Allen Bradley PLC. The protocol was designed to prove the aliveness of devices in critical infrastructures to a remote server. To the best of our knowledge, this is the first implementation of proof of aliveness protocol on a commercial off-the-shelf PLC, and the authors of [JYvDZ19] only implemented their protocol on a much more powerful device, Raspberry Pi, in C as a prototype. Our new evaluation results of PoA demonstrate its practicality on commercial PLCs.

By making this library publicly available, we believe the research community can benefit from it for future research. Open-source codes also greatly facilitate the deployment of cryptographic algorithms on legacy PLCs by the plant operators, who typically have no background in cryptography or cryptographic engineering.

Contributions. In this paper, we made the following significant contributions:

1. To the best of our knowledge, *PLCrypto* is the first cryptographic library implemented for PLCs using the languages defined in IEC-61131-3. This allows cryptography to be easily integrated into industrial systems to protect communications without the need for additional hardware or firmware modification.

2. *PLCrypto* is a comprehensive symmetric cryptographic library, covering a variety of symmetric cryptographic algorithms as shown in Figure 1. Our library is the first one that can resist tag manipulation attacks.

3. We present a case study of potential applications of *PLCrypto* by realizing a proof of aliveness protocol.

4. We evaluate the performance of *PLCrypto* on Allen Bradley PLCs made by Rockwell Automation, which is the second-largest PLC vendor in the world. We also introduce many implementation tricks to optimize the performance of the algorithms in *PLCrypto* significantly.

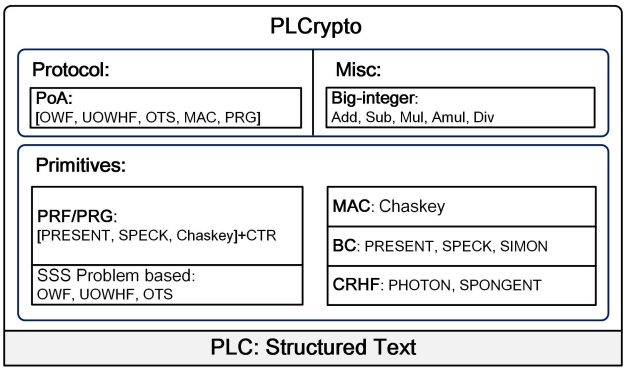


Figure 1: Overview of *PLCrypto* Library.

5. We make our library publicly available online to facilitate both future research and real-world deployment.

Organization. Related works and necessary background are introduced in Section 2 and 3. Section 4 describes the threat model, and Section 5 presents an overview of our techniques for realizing the secure and optimized implementation of *PLCrypto* on PLCs. The implementations, use case study, and benchmark results are presented in Section 6, 7, and 8, respectively. Finally, we conclude the paper and point out possible future works in Section 9. The source codes and user manual of *PLCrypto* can be freely downloaded at https://github.com/PLCrypto/PLCrypto.

2 Related Work

Industry and academia have made various proposals to use cryptography to secure PLC communications. However, in contrast to *PLCrypto*, none can be realized on commercial PLCs without any additional cost. We generally categorize the related works into two categories.[1] One requires additional hardware to be attached to PLCs to retrofit legacy PLCs. The other kind of work relies on a hypothetical scenario, where the firmware of PLCs can be programmed by PLC users or plant operators. However, this is usually not the case in reality. The firmware of mainstream commercial PLCs is not open-sourced. Also, to change the firmware of a commercial PLC, one will need to sign new firmware using a private key only known by the PLC vendor. Therefore, we argue that it is either too costly (additional hardware) or infeasible (firmware modification) for plants to secure their legacy PLCs. This naturally leads to the solution presented in this paper: a cryptographic library running on the control logic layer, which is one layer above the firmware.

Additional Devices. Allen Bradley introduced an additional CIP Security capable communication module, which can be attached to an Allen Bradley PLC [Bra19]. However, this will be a costly solution as every PLC in the plants will need to install a new communication module. American gas association (AGA) has made one proposal in AGA report 12 to recommend using cryptographic algorithms to protect the communication between field devices and the centralized control server called supervisory control and data acquisition system (SCADA) [Ass06]. In the report, the authors recommended

[1] We are aware of one work which implemented block ciphers, SIMON and SPECK, on commercial PLCs [DG17], but the PLC they used is extremely powerful and runs Windows CE. Also, their PLCs are from a vendor that is not among the top eight PLC vendors who take more than 90% market share in total globally according to [Deu17]. Moreover, the authors did not open-source their codes, and the language used in their implementation is not specified, so we cannot compare *PLCrypto* with theirs.

attaching another device called SCADA Cryptographic Module (SCM) to both parties of communication. Hence, the SCM becomes a proxy that can encrypt the message sent from the source and decrypt the message at the destination. A similar idea is also realized in [CATO17], where the authors proposed to reduce the computational overhead of the cryptographic methods by selectively encrypting security-critical messages, e.g., reading/writing requests. The proposed framework is implemented using a Raspberry Pi to tap the communication. Although this solution can be realized in a system with commercial PLCs, it is still not scalable as it requires new devices as a proxy for every PLC. Besides encryption and signature, authenticated key exchange protocols were also proposed for the applications on PLCs in critical infrastructures [JYAZ19]. It exploits the historical data stored on the SCADA server as additional authentication factors. To make this authenticated key exchange protocol compatible with legacy PLCs, one additional proxy has to be added.

Firmware Modification. Some researchers took a different approach to introduce cryptography into PLC systems. They tried to propose methods that are integrated into the firmware of PLCs. However, typically only the PLC vendors have the source codes of PLC firmware, and only the vendors can modify it. Alves *et al.* proposed to add a cryptographic layer in the network layer of PLCs, so they used AES-256 in cipher block chaining mode to protect both the confidentiality and integrity of messages [AMY17]. As a follow-up research, Alves *et al.* augmented the cryptographic layer in [AMY17] with machine learning-based intrusion detection in the framework of OpenPLC [ADM18]. Cryptographic encryption and integrity check are also embedded in Snapshotter design in [JVvD18]. Snapshotter system is a secure logging system of PLCs, which logs all security-related events on PLCs and then encrypts the messages using AES. All the above proposals require modifying the firmware of PLCs for tighter integration, so the performance evaluations in [AMY17, ADM18, JVvD18] are conducted in an open-source PLC framework called OpenPLC [Alv20] running on Raspberry Pi 3, instead of commercial PLCs.

Optimizations in Cryptographic Engineering. Another area of related works to this paper is the cryptographic engineering optimization tricks introduced by researchers working on different embedded device platforms. Hard coding and pre-computation are widely adopted techniques in accelerating computation. For example, they can be seen in cryptographic libraries like OpenSSL [YH20] and MIRACL [Mir18]. The OpenSSL implementation (as well as in the optimized code of Rijndael cipher [RBB00]) involves several hard-coded lookup tables of AES, which can significantly save time for AES round computation. Another optimization technique related to our work is the Bitslicing which breaks down the implementation of a symmetric cipher into logical bit operations that then can be computed in parallel. Such kind of optimization fits well on platforms with good parallel processing capabilities (e.g., GPU) [NAI17]. However, these traditional optimization techniques might not be effective on all platforms. In [SHB09], Stark *et al.* showed that the Bitslicing technique [RSD06] is quite inefficient in Javascript. This fact motivates Stark *et al.* to specifically develop a symmetric cryptographic library in Javascript. In this Javascript crypto library, they introduced several optimization techniques (including hard-coding strategies) tailored to couple the specific characteristics of Javascript interpreters. We stress that the traditional hard-coding and Bitslicing optimization techniques cannot be straightforwardly applied to implement cryptographic algorithms on PLC as well. That is, we face many unique challenges and difficulties in realizing *PLCrypto* as mentioned in Section 1, e.g., the restriction imposed by the programming language and the physical constraints of the resource-constrained devices. Beyond those difficulties, we also need to prevent tag manipulation attacks (detailed in Section 4), which are unique on PLCs.

3 Preliminaries

In this section, we introduce various notations and cryptographic primitives used in this paper.

General Notations. We denote the security parameter by κ, an empty string by $\emptyset$, and the set of integers between 0 and $n-1$ by $[n] = \{0, \ldots, n-1\}$. If X is a set, then $x \xleftarrow{\$} X$ denotes the action of sampling a uniformly random element from X. If X is a probabilistic algorithm, then $x \xleftarrow{\$} X$ denotes that X runs with fresh random coins and returns x. We denote the binary representation of a value X with bit size l_n as a vector $x = \langle x[0], x[1], \ldots, x[l_n - 1] \rangle \in \{0,1\}^{l_n}$, i.e., x can be represented as a bit array. We let $|\cdot|$ be an operation to calculate the bit-length of a value. We let $||$ be an operation of concatenating two strings. We let 0^n denote a bit string consisting of n zeros.

In Table 1, we summarize some important notations used in this paper.

Table 1: Some important notations

Notation	Description
κ	Security parameter
$[n]$	The set of integers between 0 and $n-1 \subset \mathbb{N}$
$\xleftarrow{\$}$	Action of sampling a uniformly random element
$\lvert\cdot\rvert$	Bit-length of a value
$\lll, \ggg$	Rotate left and right
$\ll, \gg$	Left and right shift
$\lvert\rvert$	Concatenation of two bit strings
0^n	A string consisting of n zeros

3.1 Background of PLC Programming

PLC Basics. Programmable logic controllers (PLCs) are a class of embedded devices designed specifically for controlling industrial devices (such as sensors and actuators) in industrial control systems. On a PLC, the control program is running periodically. In each period, the PLC takes inputs, executes its control program based on the inputs (from sensors), and generates outputs (to steer actuators). The period is typically called a scan cycle. All variables are called tags in PLC programming, so we use "tags" and "variables" interchangeably hereinafter. In addition, PLCs are usually connected with the supervisory control and data acquisition (SCADA) system in an industrial control system. SCADA system is responsible for collecting the operational data provided by all PLCs in the system and coordinate their control behavior to achieve the best performance.

Structured Text (ST). ST is a programming language defined by PLCOpen in IEC 61131-3 [JT13] that is tailored for PLC programming. Here, we review the ST used by Allen-Bradley (AB) PLCs [Bra18b] based on which our *PLCrypto* is realized. Note that Allen-Bradley only supports a simplified version of ST that abandons some useful data types and functionalities for cryptographic engineering. For example, unsigned integer and user-defined functions are not supported. Hence, our implementation becomes more challenging and valuable. Implementing *PLCrypto* on such a restricted platform also implies its portability on other platforms with full-fledged ST.

The syntax of ST is developed to look like that of a high-level programming language (such as Pascal or C) with loops, variables, conditions, and operators. We list the main data types and operators used by *PLCrypto* in Table 2. Moreover, PLC adopts twoâĂŹs complement to represent an integer. We use $A[\cdot]$ to denote an 1-dimensional array and $A[\cdot,\cdot]$ to denote an 2-dimensional array. We summarize the pros and cons of using ST to program as follows.

Table 2: Main data types and operators in ST

Data type	Description
SINT	Short integer, $-128, \ldots, +127$
DINT	Double integer, $-2^{31}, \ldots, +2^{31}-1$
REAL	Real number, $\pm 10^{\pm 38}$
BOOL	Bit in $\{0,1\}$

Operators	Description
MOD	Modulo-divide
AND, OR, XOR	Logical AND, OR, and XOR
+,-,*,/	Add, Substract, Multiply, Divide
**	Exponent (x to the power of y)

Benefits. An advantage of ST compared to other high-level programming languages is that it allows direct access to every bit of a SINT or DINT type variable. This feature can be a great boost of performance for cryptographic algorithms on PLCs, i.e., one can append ".[·]" to a SINT or DINT variable to read a bit of the variable. For example, given a DINT variable t, we can get the 6-th bit of t by using $t.[5]$ or $t.5$.

Drawbacks. ST does not have *pointer* like data type (as in C language), and it does not support dynamic memory management. Therefore, sending parameters between two routines (functions) might be costly, particularly concerning large-sized data (e.g., hash messages). Also, ST does not have *bit-wise shift/rotate* instructions, which are primitive functions used in many cryptographic algorithms (e.g., block ciphers). These drawbacks of ST impede the performance of the cryptographic algorithms running on PLC.

3.2 Subset-sum Problem

Let $A = \{a_0, a_1, \ldots, a_{l_n-1}\}$ be a set of l_n numbers, where $l_n \in \mathbb{N}$ and each a_i (for $i \in [l_n]$) is an l_n-bit integer. The subset-sum problem is one of Karp's NP-complete problems [IN96, DRX17, CG20] which can be viewed as inverting the following function:

$$\mathsf{F}_{\mathrm{sss}}(x, A) = \sum_{i=0}^{i=l_n-1} x[i]a_i \pmod{2^{l_n}}. \tag{1}$$

where $x \in \{0,1\}^{l_n}$, and A is a fixed parameter of $\mathsf{F}_{\mathrm{sss}}$. In *PLCrypto*, we choose the modular addition under the field $\mathbb{F}_{2^{l_n}}$ for efficiency. Given a target value $t \in \{0,1\}^{l_n}$, inverting $\mathsf{F}_{\mathrm{sss}}$ is to find an appropriate x such that $\mathsf{F}_{\mathrm{sss}}(x, A) = t$.

As shown in [IN96], many cryptographic primitives, such as one-way function (OWF), and universal one-way hash function (UOWHF), can be built from the subset-sum problem. Namely, the function $\mathsf{F}_{\mathrm{sss}}(x, A)$ directly implies the construction of OWF. UOWHFs [NY89] is also known as target collision-resistant hash function such that it is hard to find a collision where one pre-image is chosen independently of the hash function parameters. UOWHF can be realized with $\mathsf{F}_{\mathrm{sss}}(x, A)$ by appropriately truncating some bits from its output for compression purposes. We denote such a UOWHF by H_{sss}, which takes as input a message $m \in \{0,1\}^{l_m}$ and outputs a hash value $t \in \{0,1\}^{l_h}$. Moreover, Steinfeld *et al.* [SPW06] pointed out that higher-order UOWHF can also be constructed from subset-sum assumption, so it is feasible to build a UOWHF function with variable input-length $l_m \geq l_n$ using $\mathsf{F}_{\mathrm{sss}}(x, A)$ as a compression function. UOWHF has many cryptographic applications, e.g., it is widely used for hashing long messages before signing with a digital signature scheme.

3.3 Algorithms in PLCrypto

In *PLCrypto*, we include cryptographic algorithms providing the security properties that are mostly desired in real-world applications, such as confidentiality, integrity, collision-resistance, and pseudo-randomness. Most of the included algorithms are either standardized by ISO/IEC or the state-of-the-art symmetric cryptographic algorithms that fit the PLC environment. Besides subset-sum problem based OWF and

UOWHF, *PLCrypto* encompasses the MAC algorithm Chaskey [MMH+14], block ciphers PRESENT [BKL+07, PRE19], SPECK and SIMON [BSS+13], and collision-resistant hash functions PHOTON [GPP11, JG16] and SPONGENT [BKL+13]. We also include a cryptographic protocol called Proof of Aliveness (PoA) [JYvDZ19], which serves as an application example to show how to use the cryptographic algorithms in *PLCrypto*. As our implementation may require big-integer operations, we also realize some basic operations (such as addition, subtraction, multiplication, and division) to show their performance on PLCs. More details of these algorithms are reviewed in Appendix A.

4 Threat Model

In this section, we describe the threats against PLCs. Generally speaking, most commodity PLC platforms offer little security protection for remote access, which enables attackers to exploit systematic vulnerabilities. In the following, we discuss the attacker capability and the settings of PLCs.

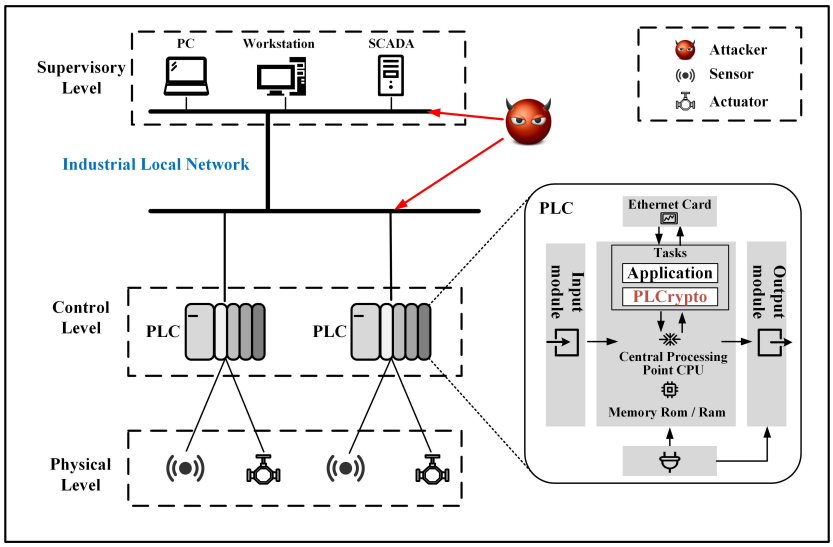


Figure 2: System Model of Industrial Control Systems. PLCs sit between the physical processes and supervisory level. Remote attackers are assumed to be able to get access to the operational technology network.

Attackers. To better illustrate the threats against PLCs, we present the high-level system model and threat model in Figure 2. PLCs are connected via wired/wireless industrial local area network to computers (e.g., SCADA in the monitor center, or even another PLC in the network). The PLCs can exchange system operation data and control messages with the SCADA and other connected PLCs. However, since the network module of many commercial PLCs do not support any security features, the network communication to PLCs is usually unprotected and open to attackers once they get access to one of the devices or access points in the network.

In this work, we mainly consider network attackers against PLCs, who can take control of the communication of a plant where the target PLCs are installed. The goal of the network attackers is to manipulate the executable code or data stored in the PLC and try to steal the secrets stored in the PLC (e.g., the system status and parameters). We assume such attackers could be powerful enough to connect to PLCs and to receive/inject new messages via communication ports (e.g., launched by remote PLC management software

such as Studio 5000 [Bra09b] or third-party customized program like Pycomm [AM18]). In addition, a network attacker may leverage its communication power to download/upload the control program from/to PLC,[2] if it is allowed, and manipulate the tags of PLC (i.e., read and modify tags). However, we do not consider insider attackers who can physically access the PLC and locally manipulate it. We also assume that no additional security appliance is attached to the PLCs to provide secure communication, such as Stratix 5950 [Bra20], which may cost thousands of dollars each.

Tag Manipulation Attack. To facilitate the control and supervision of PLCs, modern PLCs usually support online tag reading/writing capabilities without interrupting the operations of PLCs. Any tags (variables) in the control program can be written or read anytime, including when the program is being executed (within scan cycles). This feature is very useful when operators need to update certain parameters of the system without shutting down the whole service. However, this online tag manipulation capability would trivially lead to a family of attacks on cryptographic algorithms, which we collectively call *tag manipulation attacks* (TMA). A simple read or write operation to a sensitive tag can leak sensitive information (e.g., secret keys) and compromise operational parameters (e.g., the rotational speed of connected centrifuges, in the case of Stuxnet [FMC11]). For the security of *PLCrypto*, we introduce *tag manipulation attacks*, as a kind of adversaries that are unique on PLC platforms. Tag manipulation attacks would jeopardize not only the confidentiality but also the integrity of critical tag values. In the implementation section of *PLCrypto*, we will show the effects of tag manipulation attacks on various algorithms and how to prevent them. We assume that no extra trusted hardware is used to store the cryptographic secrets for PLC. Note that we are the *first* to introduce this unique threat model and propose countermeasures to the attacks on PLC platforms.

Sketch of the Abstract Security Model for *PLCrypto*. Including the aforementioned tag manipulation attacks, we can model the attack capabilities by a setup with interactive Turing machines and a traditional black-box security notion $\mathcal{N}$ of the corresponding algorithm (e.g., $\mathcal{N}$ = Authenticated encryption with associated data), where

- The "game" machine maintains all variables in two mutually-exclusive collections $\mathsf{HL}.x$ and $\mathsf{TL}.x$, for hard-coded tags and the other tags, respectively.
- All variables $\mathsf{HL}.x$ in HL are assigned with a value in game initialization.
- Any variable $\mathsf{TL}.x$ can be created, written, read, or deleted by (the code of) the oracles of $\mathcal{N}$.
- Apart from the oracles of the traditional security notion $\mathcal{N}$ would expose, the attacker would get two additional oracles readVar(varName) and writeVar(varName, varValue).
  - readVar(varName) returns the latest value of the variable TL.varName for any currently existing variable in TL.
  - writeVar(varName, varValue) sets the value of TL.varName to varValue for any currently existing variable in TL.
- The oracle calls are atomic, so any variables created or deleted in a single oracle call are not visible to the attacker, as in the real world.

Note that there is still a discrepancy with real life, which is ensuring that a function can execute within a single scan cycle on a device. As this is platform-dependent, this assumption must be checked per-device model.

[2] In Section 5, we will show how we can stop attackers from downloading/uploading control programs by specific system settings.

5 *PLCrypto* Overview

5.1 System Level Settings for Security

PLC Modes and Task Types. Allen Bradley's PLC [Bra09a, Bra09b] provides different kinds of operating modes and tasks that may affect the behaviors of attackers. Namely, attackers may have to tailor their attack strategies under specific operating mode or task type. In particular, the appropriate choice of operating mode and task type may provide the necessary security guarantee required by our implementation.

PLC has four modes of operation for running a project, i.e., program mode, run mode, test mode, and remote mode. The modes can only be selected by a hardware switch on the PLC device, so no remote adversary can change the PLC mode once it is set. The details of these operating modes can be found in [Bra09a]. Here the key difference (which is also the key issue related to security) among these operating modes is that a computer cannot remotely manipulate (e.g., download or upload) the program of PLC in the **run mode**. However, it is feasible for an adversary to manipulate or read PLC programs in all the other modes. In the run mode, PLC can carry on all its features, such as reading the inputs, scanning the logic, and outputting the results, except for being re-programmed. Also, the computer connected to the PLC can still remotely manage the running status (including tag values) in the run mode.

Another factor that affects security is the type of PLC tasks. An Allen-Bradley PLC can support and schedule multiple tasks of various types, but a PLC can run only one task at a time. A task may be interrupted by another task which has a higher priority. There are three kinds of tasks that we can configure, i.e., continuous task, event task, and periodic task. The continuous task runs all the time in the background. It is used to repeatably run the operations which do not belong to motion, communication, and periodic or event tasks. An event task performs a function only when a specific event (trigger) occurs. A periodic task performs a function at a specific period, and it can interrupt any lower priority tasks. Besides, a PLC may have other integrated system-level tasks. The most relevant one is the communication task which is responsible for exchanging data between PLC and remote monitoring computers, e.g., online tag reading/writing and control logic editing.

A crucial security issue regarding a program task is whether it can be interrupted by the communication task (which might send/receive sensitive tag values to/from a connected attacker), when it is executing. According to the PLC manual [Bra09b], the periodic and event tasks cannot be interrupted by a communication task.

PLC Settings. To guarantee the secure implementation of our cryptographic library in a PLC against the network attackers (i.e., preventing the attackers from trivially stealing the secrets in a PLC), we first assume that the PLC is switched to "run" model using a physical switch on the PLC, so that the network attackers is not allowed to download/upload a project to/from it. If the PLC needs to be re-programmed, it should be done by an administrator with strict supervision, e.g., locally switching the mode to "program" by a staff. In practice, it is very rare for critical infrastructures to update the control program of PLCs because nobody wants to interrupt the operations. In this way, the network attackers are unable to get the secrets hard encoded within a control program. In a PLC, the control program is executed repeatedly/periodically, and the time between the repeated executions is called a scan cycle. To prevent the network attackers from remotely reading and writing tags within a scan cycle, we assume that all tasks implementing cryptographic algorithms are assigned to be event or periodic tasks with priority higher than the communication task, so that they cannot be interrupted by the communication task. In this setting, within scan cycles, the network attackers cannot read the *intermediate* tag values which may contain secrets, but all variables are still readable/writable by the attacker after *PLCrypto* thread completes execution or between *PLCrypto* executions.

Security Validation. We implemented the tag manipulation attacks (TMA) against our Allen-Bradley PLC via Studio 5000 (the official tool for managing every detail of an Allen-Bradley PLC, including read-only tags). In our experimental attack, we develop a toy task which does the following steps:

- Initialize DINT tags TARGET := 0;
- Repeat the code TARGET := TARGET + 1 for 100 times;
- After the above loop execution, if TARGET $\leq$ 100, TARGET := 0, else set the attack result ATTACK_RESULT := TARGET.

Note that we are testing the modification capability by assigning TARGET with a value that is larger than 100 via Studio 5000 during the execution of the above toy task. If such a modification of TARGET is successful within the execution of the toy task, then it will not be cleared in the last step, and we will observe the modified value in the other tag ATTACK_RESULT.

We first set the type of the toy task to be *continuous*. Then, through the run-time tag management interface of Studio 5000, we can see the change of the TARGET during the execution, and we can manually modify its value using Studio 5000. However, when we change the PLC setting as mentioned above (i.e., we change the type of the toy task to be *periodical* and set its priority to be higher than the communication task), then we can only observe zero through Studio 5000. This means that the TMA within one scan cycle are prevented if we properly configure task types and PLC modes. More details on PLC modes and tasks are in Section 3.1 (also in the manual [Bra09b]).

However, a PLC task is usually repeatedly executed and two executions of the same task share tags, so we still need a solution to protect the critical tags (e.g., cryptographic keys) between two consecutive executions from TMA. In other words, we still need to prevent TMA that manipulates tag values between scan cycles, which we will discuss in Section 5.3.

5.2 Overview of Implementation Tricks

Our primary security concern on *PLCrypto* implementation is to resist the *tag manipulation attacks*, in particular for protecting the cryptographic keys. Besides all system settings above, the most important implementation trick for this problem is to hard-code the concrete values of critical tags (either keys or parameters) into the program code to protect the **confidentiality** of them, since it is possible to prevent the attackers from accessing the program code after the system is set to the RUN model (as discussed in Section 4). In our implementation, we will leverage two hard-coding strategies:

- **Hard-coding with Runtime Loading** (**HC-RL**): Unlike the program on PC (which is immune to TMA), it is insecure to load the tag values once and use them across multiple scan cycles on PLC, since adversaries can trivially read the tag values via communication tasks in the interval between two executions of the algorithm. Note that the attacker does not have to issue read/write requests exactly in the interval to manipulate the tag values successfully. The received read/write requests will be scheduled when the communication task is running. In this **HC-RL** strategy, an algorithm must load the concrete values of critical tags on the fly at the beginning of every execution (in a scan-cycle), e.g., *key_tag* := 12345. At the end of the execution, the algorithm should erase the tag, e.g., *key_tag* := 0, if it is a secret, such as a cryptographic key. This strategy is suitable for the situation when only a small amount of tags are sensitive and need to be hard-coded. This approach is simple and easy to implement.

- **Hard-coding at Where-used** (**HC-WU**): When there are lots of hard-coding tags, the **HC-RL** approach may become a performance bottleneck of the implementation. For example, as each number a_i in the parameter $A = \{a_0, a_1, \ldots, a_{l_n-1}\}$ of subset-sum based OWF would be represented by $\lceil l_n/31 \rceil$ DINT numbers in the implementation, it requires $\lceil l_n^2/31 \rceil = 2304$ assignments (when $l_n = 256$) to load the entire A matrix, which would take a lot of time on a PLC. Hence, in this solution, we hard-code a tag value at where it is used. E.g., for an expression $t[i] := m[i] + a[i]$ where $a[i]$ stores a secret tag value 12345, we can transform it into $t[i] := m[i] + 12345$. However, this approach has a side-effect: if such a hard-coded expression is wrapped in a loop statement, e.g., for $i := 1$ to 100 do $t[i] := m[i] + a[i]$, then we have to unroll the loop into a list of expressions realizing the equivalent function with concrete index i, such as $t[1] := m[1] + 12345$, $t[2] := m[2] + 12345$, …, $t[100] := m[100] + 12345$. In *PLCrypto*, we utilize a Python script to generate these expressions automatically.

Another goal of this paper is to seek efficient implementations of the selected algorithms on PLC. The core objective of our optimization technique is to reduce the number of computation steps. To do so, we shall take full advantage of the bit-accessibility of ST to improve the efficiency of implemented algorithms and realize the necessary functionalities like shifting and rotation. Besides, we extensively rely on pre-computation in conjunction with hard-coding strategies to improve the performance. In other words, we can pre-process many computation steps of an algorithm and hard-code them in exchange for efficiency. We summarize the optimization ideas in the following:

- **Bit-wise Read and Write** (**B-RW**): With the bit-wise accessibility of ST, we can read and write a bit of an integer just like accessing the integer (e.g., obtain the carry bit in big-integer addition), unlike the implementation (e.g., [BKL+12]) based on C language that needs shifting and AND operations.[3] With the bit-wise write capability, we can erase bits with few cheap assignments. This could be useful in realizing the modular operations modulo 2^n, i.e., we only need to clear the bits beyond the $(n-1)$-th bit.

- **Bit-wise Move** (**B-MV**): Thanks to **B-RW**, we could also directly move a bit to the target position with only one assignment statement, e.g., $s.[j/32] := t.[i]$ moves the i-th bit of t to the $j/32$-th bit of s. Relying on this approach, we could efficiently implement the basic functionalities, including shift/rotate and permutation box. In addition, when the indices can be pre-determined (or pre-computed), e.g., in bit-wise rotations, we can pre-compute all target positions (e.g., $j/32$ in the above example) and hard-code all movement steps following the **HC-WU** strategy.

- **Merge Bit-wise Operations** (**B-MO**): The objective of this approach is to merge bit-wise operations of a procedure (e.g., permutation box) into other procedures (e.g., substitution box) instead of executing these procedures independently so that it could reduce some intermediate computation steps. For example, one can apply the permutation box to each intermediate substitution result on the fly rather than at the end of the substitution procedure (to avoid the steps for assembling the small intermediate substitution results to a large value). An example of this optimization approach could be found in our implementation of PRESENT.

Nevertheless, to realize the above general hard-coding and optimization ideas, we still need to study and test the concrete optimization steps for specific algorithms.

[3] Using AND operation to get a nibble can outperform **B-RW** only when we are accessing the least significant bits (LSB). Otherwise, we have to shift the extracted bits to LSB using division, which costs more time. Things become more complicated if the sign bit is involved.

5.3 Security Principles against Tag Manipulation Attacks

Overview of Security Principles. We summarize our comprehensive principles for preventing tag manipulation attacks concerning different attack targets as below:

- **Confidentiality and Integrity of Any Tags Within One Scan Cycle:** As we mentioned in Section 5.1, once the mode of the PLC is set to the RUN mode and the cryptographic task has a higher priority than that of the communication thread, remote attackers can no longer interrupt the cryptographic task within one scan cycle and steal/compromise any intermediate values. This principle can be seen in all the algorithms implemented.

- **Confidentiality of Any Secret Constants between Scan Cycles:** Even if we have the necessary PLC system settings mentioned above, the communication task will still be scheduled between every two consecutive scan cycles. Thus, an attacker can request to read/write to any tags when the cryptographic task is not running. As a rule of thumb, at the end of a cryptographic task, all intermediate values that can potentially leak the secret must be cleared. Also, all secret values (e.g., secret keys) are hardcoded in the control program using **HC-RL** or **HC-WU**. Since the PLC is at the RUN mode, the attacker cannot access the program itself; he/she has no way to *directly* read the secret values from the program or the tags. This design principle is used in all algorithms involving a secret.

- **Integrity of Constants between Scan Cycles:** In cryptographic algorithms, pre-defined constants play a critical role. Sometimes, if the constants are tampered with by an attacker, a fault injection attack can be launched. To prevent such an attack, we apply our hard-coding implementation tricks **HC-RL** and **HC-WU**. Essentially, all constants need to be loaded again from the program at the beginning of a cryptographic task to prevent any malicious modification of the constants between scan cycles. We present an example of this threat scenario in the implementation of subset-sum based OWF in Section 6.1, where we need to protect the integrity of the public constant matrix.

- **Integrity of Public Variables between Scan Cycles:** The primitive algorithms (like OWF, BC, MAC, and HASH) we implemented are all stateless, so we can safely clean all variables used inside the stateless algorithms. However, when we use the algorithms in a larger context (e.g., in a protocol or in a certain mode), we may need to keep a public state variable over multiple scan cycles. This public variable is subject to TMA as well. We have to compute a MAC to protect the integrity of the states at the end of a cryptographic task, and check its integrity before it is used again in the next scan cycle. Fortunately, the MAC algorithm implemented is very efficient. An example of this practice can be found in Section 7 where we integrate multiple algorithms and implement a protocol called PoA, in which a public monotonic counter needs to be maintained.

- **Confidentiality and Integrity of Secret Variables between Scan Cycles:** Though we have not encountered any secret variables that need to be kept in multiple scan cycles in our implementations, for the sake of completeness, we will recommend using encryption and MAC algorithms (or an Authenticated Encryption scheme) to protect any secret variables in such a case.

Minimal Soft/Hardware Requirements. Given the above security analysis against tag manipulation attacks, we summarize the minimal software and hardware features required for securely running *PLCrypto* as follows:

1. The PLC supports standard ST defined by IEC-61131-3;
2. The PLC supports priority based task scheduling and supports at least one kind of task whose priority level is higher than that of the communication thread;
3. The PLC has a hardware switch to set the PLC to run mode, preventing remote users from access the control program;
4. The PLC has enough memory space to run *PLCrypto* code as specified in Table 6 for each algorithm.

5.4 Selection Criteria of Algorithms in *PLCrypto*

Generally speaking, we mainly select cryptographic algorithms which are standardized and efficient. We also consider algorithms (e.g., subset-sum based OWF) that are provably secure and efficient and can serve as an easy-to-understand example for demonstrating the TMA threats. In consideration of efficiency, it is possible to apply the above hard-coding and optimization strategies as *metrics* for selecting the cryptographic algorithms in *PLCrypto*. In the following, we list our selection criteria in detail:

1. **Standardized** (**STD**). Standardized algorithms are usually widely recognized and accepted in practice, so our top priority of choosing algorithms is to pick algorithms standardized by either ISO (International Organization for Standardization) or NIST (National Institute of Standards and Technology).
2. **Promising Performance on PLCs** (**PPP**). We also try to seek algorithms that are more suitable for PLC, such as subset-sum based OWF and SPECK. Our benchmark results for such algorithms may provide a baseline for future research.
3. **Pre-computation Friendly** (**PCF**). Algorithms that are easy to pre-compute their expensive operations could reduce the computation cost, such as PHOTON with a precomputed tables-based implementation (applying both the SBOX and the MixColumns coefficients at the same time).
4. **Bit-wise Operable** (**BWO**). The above bit-wise optimization strategies can optimize algorithms that comprise of many bit-wise operations (such as permutation box).
5. **Short Keys and Parameters** (**SKP**). Since keys and parameters should be hard-coded due to TMA, this criterion would affect the performance significantly.

Table 3 summarizes the techniques and selection criteria applied to the algorithms in *PLCrypto*. The PoA is used as a "*use case*" to show: i) usage of algorithms in *PLCrypto*; ii) example of protecting the **integrity** of tags across scan-cycles; iii) new performance results of PoA on commercial PLCs.

6 PLCrypto Implementation

In this section, we elaborate on the implementations of selected cryptographic algorithms in *PLCrypto*. Our implementations are done on PC and PLC, respectively. On a PC, we mainly use Python to automate the initialization and code generation (e.g., key sampling, parameter generation, pre-computation, and key-dependent hard-coding) of algorithms. To load the parameters (e.g., A of $\mathsf{F}_{\mathsf{sss}}$), PLC can run an independent task for initializing those parameters (represented as tags) used by the cryptographic algorithms. For readability, we skip the details of the initialization task, which just consists of a few assignment steps

Table 3: Overview of algorithms in *PLCrypto*, and their corresponding selection criteria and optimization tricks.

Algorithms	Selection Criteria	Hard-coding	Optimization
Subset-sum based OWF, UOWHF, OTS	**PPP**, **PCF** **BWO**	**HC-WU**	**B-RW**
Bit-wise Shift/Rotate	**PPP**, **PCF** **BWO**	**HC-WU**	**B-MV**
Big-integer Operations	**STD**, **BWO**	N/A	**B-RW**
Chaskey	**STD**, **PPP**	**HC-RL**	Bit-wise Rotate Big-integer Add/Sub
PRESENT	**STD**, **PCF** **BWO**	**HC-RL** **HC-WU**	**B-MO**
SPECK, SIMON	**STD**, **PPP**	**HC-RL**	Bit-wise Rotate Big-integer Add/Sub
PRF/PRG	**STD**, **PCF**	**HC-RL** **HC-WU**	PRESENT-CTR SPECK-CTR
PHOTON	**STD**, **PCF**	**HC-RL**	**B-RW**
SPONGENT	**STD**, **PCF**	**HC-RL**	**B-MV**
PoA	**Use Case**	**HC-RL** **HC-WU**	SPECK-CTR Chaskey OWF, UOWHF, OTS

in the ST program. Here we focus on describing the detailed optimizations of algorithms implemented on PLC using ST. As the secret keys of algorithms should be hard-coded, the keyed functions will no longer explicitly take as input the keys in the following description. Here we focus on describing the detailed optimizations of algorithms implemented on PLC using ST. Also, we present some pseudo-codes in Appendix C.

Notations. Let $\hat{\lll}$ and $\hat{\ggg}$ be hard-coded left rotation and right rotation operations implemented on a PLC. $\hat{\ll}$ and $\hat{\gg}$ represent hard-coded left/right shifting operations. And DN denotes the number of DINT variables that are required to represent a big number.

6.1 Implementation of OWF and UOWHF

In this subsection, we show the implementations of subset-sum (sss) based OWF and UOWHF. We include the subset-sum based OWF because it is much more efficient than using other lightweight cryptographic hash functions as OWF. And we will use it as an example to show a special form of TMA when the attacked tags are parameters (which would be usually treated less carefully) and the feasibility of our optimization approaches.

In the following, we first introduce a concrete TMA on subset-sum based OWF. Then we present two approaches to implement OWF: the first one follows the original steps of the algorithm, and the second one exploits a time-space trade-off to improve the efficiency of OWF by leveraging hard-coding strategy **HC-WU** and optimization approach **B-RW**.

A Tag Manipulation Attack on OWF. We first study the importance of hard-coding the parameters. Consider the situation that one initializes the parameter $A = \langle a_0, a_1, \ldots, a_{l_n-1} \rangle$ once with a separate initialization task but uses it repeatedly across executions/scan cycles. However, when such an initialization task is done, the network attackers are able to modify A to launch a *tag manipulation attack* to recover the pre-image x. For example, to obtain j-th bit x, the attacker only needs to set $\{a_i\} = 0$ for $i \neq j$ and $i \in [l_n]$. It is obvious that the evaluation result of OWF would be either a_j or 0 that could be used to infer $x[j]$ trivially.

Algorithm 1: Evaluation of Subset-sum based OWF

Input: DINT ARRAY $x[\mathrm{DN}_t]$ where $\mathrm{DN}_x = \lceil \frac{l_n}{32} \rceil$.
Output: DINT ARRAY $t[\mathrm{DN}_x]$, $\mathrm{DN}_t := \lceil \frac{l_n}{31} \rceil$.
1: $v := 0$; //v is used for indexing the bits of x
2: $u := 0$; // u is a DINT index of x
3: Initialize DINT ARRAY $\{A[i,j]\}_{i\in[l_n],j\in[\mathrm{DN}_t]}$, which stores the values of the parameter $\langle a_0, a_1, \ldots, a_{l_n-1}\rangle$.
4: **for** $i := 0$ to $l_n - 1$ by 1 **do**
5: **if** $v = 32$ **then**
6: $v := 0$; $u := u + 1$; //switch to the next 32-bit block
7: **end if**
8: **if** $x[u].[v]$ **then**
9: $\{t[j]\}_{j\in[\mathrm{DN}_t]} := \{t[j]\}_{j\in[\mathrm{DN}_t]} \hat{+} \{A[i,j]\}_{j\in[\mathrm{DN}_t]}$;
10: **end if**
11: $v := v + 1$;
12: **end for**
13: **return** $\{t[i]\}_{i\in[\mathrm{DN}_t]}$;

Baseline Implementation of OWF. To avoid the TMA, we can load A on the fly in each OWF evaluation in this implementation scenario. As a baseline, we first realize the OWF following **HC-RL** hard-coding strategy. To realize an addition modulo 2^{l_n}, we appeal to the standard *multiple-precision addition* [MvOV96, Algorithm 14.7]. We let $\hat{+}$ denote the big-integer addition implemented on PLC modulo 2^{l_b} where $l_b \geq 32$. To efficiently get the carry bit, we choose to use a digit base 2^{31} to realize big-integer addition so that we can obtain the carry bit (i.e., the sign bit of a DINT variable) with only one assignment statement. In this way, we can avoid dealing with the overflow caused by the sign bit, which may need many additional judgments or logical operations.

The input $x \in \{0,1\}^{l_n}$ of $\mathsf{F}_{\mathrm{sss}}(x, A)$ is represented by $\mathrm{DN}_x = \lceil \frac{l_n}{32} \rceil$ DINT variables while we realize the evaluation sub-algorithm on PLC. We initialize the parameter A hard-coded in the PLC program by sampling a random $l_n \times \mathrm{DN}_t$ two-dimensional 31-bit integer array $A[l_n, \mathrm{DN}_t]$ on PC, where the set of integers $\{A[i,j]\}_{j\in[\mathrm{DN}_t]}$ represents the i-th l_n-bit number a_i of A, and $\mathrm{DN}_t = \lceil \frac{l_n}{31} \rceil$.

The evaluation sub-algorithm of $\mathsf{F}_{\mathrm{sss}}(x, A)$ is shown in Algorithm 1. We can leverage the **B-RW** optimization idea to easily get a bit of x by $x[u].[v]$, where the variable $u \leq \mathrm{DN}_x$ is an ARRAY index and the variable $v \leq 30$ is bit index of x. So the evaluation of $\mathsf{F}_{\mathrm{sss}}$ can be realized with two-layer nested loops, in which the outer layer is to decide whether $A[i,\cdot]$ should be involved based on $x[u].[v]$, and the inner loop is to calculate the l_n-bit big-integer addition.

Faster Implementation of OWF. Note that the initialization of the parameter A in Algorithm 1 requires $l_n \times \mathrm{DN}_t$-times assignments, which take almost half of the computational cost of the entire algorithm. To improve the performance, we could apply **HC-WU** hard-coding and **B-RW** optimization strategies. Specifically, we can leverage "If" statement in ST to hard-code A and all computation steps involved in Equation 1. To do so, we make use of Python to automatically generate all of the l_n "If" statements between Line 8 and Line 10, with concrete values of u, v, i, j, DN_t and DN_x. In the meantime, the big-integer addition involved in each "If" statement is hard-coded following strategy **HC-WU**.

Implementation of UOWHF. The UOWHF H_{sss} with fixed message input length can be straightforwardly obtained by an OWF $\mathsf{F}_{\mathrm{sss}}{}'$ with customized input and output spaces,

so we have $\mathsf{H}_{sss} = \mathsf{F}_{\mathrm{sss}}{}'$. Instead of using length-preserving OWF, we specifically set the output length of $\mathsf{F}_{\mathrm{sss}}{}'$ to be half of the input, i.e., $l_m = l_n$ and $l_h = l_n/2$. Then each element a_i in the public parameter $A = \{a_0, a_1, \ldots, a_{l_n-1}\}$ has l_h-bit. The implementation of $\mathsf{F}_{\mathrm{sss}}{}'$ is similar; thus, we omit repetition here. To extend the message space, we can divide an arbitrarily long message into multiple $l_n/2$-bit message blocks and hash them one by one iteratively.

Security Analysis. Here, we focus on analyzing the resistance of TMA against our implementation of OWF. The security of UOWHF implementation is implied by that of OWF. Note that the subset-sum based OWF only leverages constant tags between scan cycles, i.e., the parameter A. The network attacker cannot manipulate the **HC-RL** hard-coded tags of A between two scan cycles since the OWF task would load and refresh the tag values in every scan cycle from the code. By our PLC settings, the network attacker cannot read/write the tags (including parameter A and all other intermediate results) during the execution of an OWF task. In a nutshell, our OWF implementation blocks all attack surfaces of TMA attackers.

Furthermore, it is also important to understand the side-channel leakage of the implementation via some obvious side channels like timing.[4] Note that the performance of OWF depends on the number of one bits of the input value, i.e., no operations are done for zero bits. Hence the network attackers may exploit timing-based side-channel information to infer the input of the OWF. We roughly analyze this kind of threat based on the different usage of OWF. If the OWF is used as a hash function for message compression, then runtime would have no impact on the collision resistance of it. When the input of OWF is a secret, the runtime of OWF will be close to the average case since the input should be chosen at random with sufficient large entropy (so the hamming distance between two secrets should be close). Nevertheless, to resist timing-based side-channel attacks, the runtime of OWF is better to be constant. To achieve this, one could add dummy operations to handle the zero bits in the input to ensure that the runtime of OWF to be always the worst-case performance. We leave a concrete solution for preventing timing-based side-channel attacks as one of the future works.

6.2 Shifting and Rotation Operations

Shifting and rotation operations are extensively used by symmetric cryptographic algorithms. Unfortunately, some PLCs [Bra18a] do not provide any bit-wise shifting/rotation instruction in structured text (ST). Therefore, we have to develop bit-wise shifting/rotation operations first using ST as a building block for implementing other algorithms. Of course, a shifting operation can be realized by multiplication or division operations; however, such an approach is inconvenient and inefficient for signed DINT variables and big integers. Note that some PLCs do not support 32-bit unsigned integers. So a few more operations (including arithmetic and logic operations) should be carried out to deal with the sign bit and overflow, in particular when a big integer is involved. Another benefit of our hardcoded shifting/rotation operations is its constant runtime which is independent of the positions being shifted/rotated. Hence, they leak nothing through timing.

Our solution is to leverage the bit-wise accessibility of a DINT variable to directly move a bit into the corresponding target position, i.e., by utilizing the optimization approach **B-MV**. And it can be easily applied for big integers, which are represented by a few DINT variables. We will call this approach as bit-assign shift/rotate. Specifically, we develop a function $\mathsf{Shift}(Dir, isRot, m, pos)$ to realize all shift/rotate operations that are needed by

[4] Since the exact execution time of atomic operations may vary from one platform (e.g., the underlying processor and the compiler) to another, in this paper, we analyze timing attack resilience of *PLCrypto* based on the number of atomic operations. The exact timing attack resilience needs to be carefully tested before *PLCrypto* is deployed.

PLCrypto, where $Dir \in \{L, R\} = \{1, -1\}$ denotes the direction in either left (L) or right (R), $isRot \in \{0, 1\}$ is the operation type where 0 denotes shifting and 1 indicates rotation, m is the operand being shifted or rotated, and pos is the number of bits to shift or rotate. The concrete steps of Shift are shown in Algorithm 2.

Algorithm 2: Shift/Rotate Operations

Input: DINT $Dir \in \{L, R\} = \{1, -1\}$ where $L = 1$ is left and $R = -1$ is right, BOOL $isRot \in \{0, 1\}$; DINT ARRAY $m[\mathrm{DN}_m]$; and SINT pos, where $1 \le pos \le (32 \cdot \mathrm{DN}_m) - 1$.

Output: DINT ARRAY $m[\mathrm{DN}_m]$, $m := m \lessgtr pos$ where $\lessgtr \in \{\hat{\ll}, \hat{\gg}, \hat{\lll}, \hat{\ggg}\}$.

1: $posp := pos \cdot Dir$; //$posp$ would be negative if $Dir = -1$
2: **if** $isRot = 1$ **then**
3: **for** $i := (32 \cdot \mathrm{DN}_m) - 1$ to 0 by 1 **do**
4: NP := $(i + posp)$ MOD $(32 \cdot \mathrm{DN}_m)$; //new position
5: $r[\mathrm{NP}/32].[\mathrm{NP\ MOD\ 32}] := m[i/32].[i \text{ MOD } 32]$;
6: **end for**
7: **end if**
8: **if** $isRot = 0$ **then**
9: $B := (Dir + 1)/2$; Start := $(32 \cdot \mathrm{DN}_m) - 1 - pos \cdot B$;
10: End := $pos \cdot B - posp$;
11: **for** $i :=$ Start to End by -1 **do**
12: $r[(i + pos)/32].[(i + pos) \text{ MOD } 32] := m[i/32].[i \text{ MOD } 32]$; //move i-th bit to new position
13: **end for**
14: **end if**
15: **return** $\{m[i]\}_{i \in [\mathrm{DN}_m]} := \{r[i]\}_{i \in [\mathrm{DN}_m]}$;

Remark 1. We stress that all arithmetic operations in Shift can be pre-computed and hard-coded if pos is known in a specific algorithm (e.g., Chaskey and SPECK). That is, we could apply the hard-coding strategy **HC-WU** and use Python to generate all assignments involved in Line 5 or Line 12 with concrete array index values. So the hard-coded version of Shift is very efficient since it requires only a constant number of assignments determined by the bit-length of m.

In Figure 3, we show a code snippet of a concrete hard-coded rotate function with 64-bit operator, i.e., Speck_SR_m$\hat{\lll}$3, used in the implementation of SPECK. Here the rotated message Speck_SR_m is represented by two 32-bit DINT variables, where Speck_SR_m[1] stores left-most 32-bit and Speck_SR_m[0] otherwise. The rotation result Speck_Rotate_Result is also stored in two DINT variables. For example, to rotate the 52-th bit to 3 positions is to move Speck_SR_m[1].[19] to Speck_Rotate_Result[1].[22].

We will use the set of operators $\{\hat{\ll}, \hat{\gg}, \hat{\lll}, \hat{\ggg}\}$ to denote all kinds of operations realized by $\mathsf{Shift}(Dir, isRot, m, pos)$, e.g., $m \hat{\ll} pos$ is short for $\mathsf{Shift}(1, 0, m, pos)$.

6.3 Implementation of MAC Algorithm Chaskey

Chaskey [MMH+14] is a very efficient MAC family specifically designed for 32-bit microcontrollers by Mouha *et al.* It is standardized by ISO/IEC 29192-6:2019 [CHA19]. To initialize the program, we first generate a MAC key on PC so that it can be hard-coded following strategy **HC-RL**.

The MAC evaluation algorithm of Chaskey is a permutation based scheme. We implement the most efficient permutation π_c [MMH+14, §3.2] which is realized by our

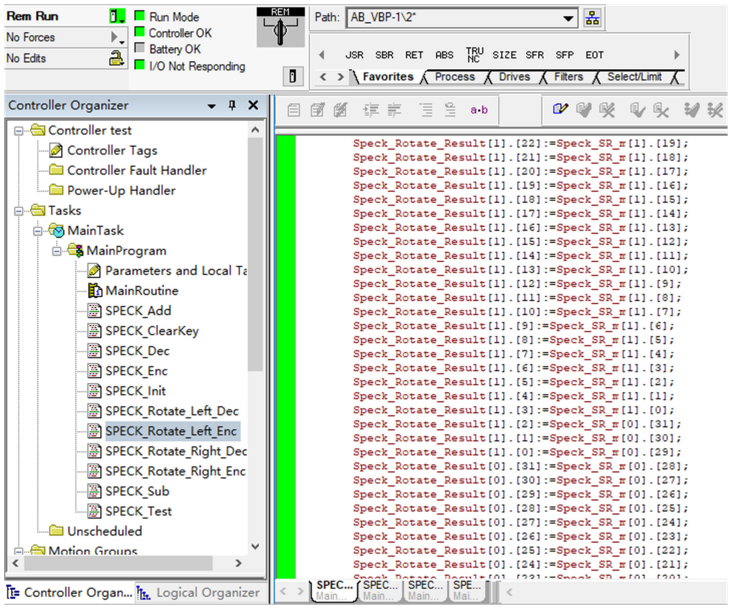


Figure 3: Code Snippet of Hard-coded Rotation

new shift/rotate Algorithm 2. Other steps of Chaskey are implemented following its specification.

Security Analysis. Since we only need to protect the keys of Chaskey, which are protected based on our **HC-RL** hard coding strategy, no network attackers can read/write those fixed tags in multiple scan cycles. Similarly, the network attacker cannot manipulate the tags within one scan cycle because of our PLC settings. Hence, our implementation of Chaskey can prevent TMA.

Moreover, The major operations, i.e., rotation, in the permutation sub-algorithm π_c is hardcoded as presented in Section 6.2 have constant-time execution time. And all other codes of Chaskey mainly involve standard arithmetic operations whose performance is independent of the input message or internal states. Hence, our Chaskey implementation has constant runtime. Namely, it does not leak any timing-based side-channel information to network attackers.

6.4 Implementations of Block Ciphers: PRESENT, SPECK, and SIMON

In this subsection, we introduce the implementations of block ciphers included in *PLCrypto*. The first one is PRESENT invented by Bogdanov *et al.* [BKL+07], and recently standardized by ISO/IEC 29192-2:2019 [PRE19]. The other two cipher families included in *PLCrypto* are SPECK and SIMON, respectively, which are designed by Ray *et al.* [BSS+13], and standardized by ISO/IEC 29167-21:2018 [SPE18]. Both schemes support various parameter sets and key sizes, so they offer more flexibility than PRESENT for users in choosing proper parameters for their applications.

The key generation of all these block ciphers can be pre-computed, so we implemented the key generation procedure on PC in Python and hard-coded the round keys in the generated ST code of the encryption and decryption schemes following **HC-RL**. Since the decryption is an inversion of the encryption for all implemented block ciphers, we will just describe the implementations of the encryption algorithms. Since we pre-compute key scheduling of the block ciphers in our implementation, it will limit the usable modes of op-

Algorithm 3: Encryption of PRESENT

Input: DINT ARRAY $m[2]$ storing 64-bit message.
Output: DINT ARRAY $c[2]$, storing 64-bit ciphertext.
1: Assign hard-coded keys $k_1||\ldots||k_T$ to KEY array $\{K[i]\}_{i\in[T+1]}$;
2: **for** $i := 0$ to $T-1$ by 1 **do**
3: AddRoundKey: St := St XOR m;
i.e., St[0] := St[0] XOR m[0] and St[1] := St[1] XOR m[1]
4: Run the merged SBOXLayer and PBOXLayer as:
5: **for** $j := 0$ to 15 by 1 **do**
6: i) Assign j-th nibble of St to tag stmp.$[z]$:= St$[u]$.$[4*v+z]$,
where $z \in [4]$, $u = j/8$, and $v = j$ MOD 8;
7: ii) Obtain the substitution result stmp := SBOX(stmp);
8: iii) ptmp[blk].[pos] := stmp.$[\delta]$, where $\delta \in [4]$,
blk := $\frac{\mathsf{PBOX}(j*4+\delta)}{32}$, and pos := PBOX$(j*4+\delta)$ MOD 32;
9: iv) Update the state St[0] := ptmp[0] and St[1] := ptmp[1];
10: **end for**
11: **end for**
12: Run AddRoundKey to generate final ciphertext
13: Clear $\{K[i]\}_{i\in[T+1]}$, St, stmp, and ptmp;
14: **return** $c = c[0]||c[1]$;

erations; e.g., turning a block cipher in *PLCrypto* into a hash function by DaviesâĂŞMeyer mode may not be feasible [MVOV18].

Implementation of PRESENT. It is built based on a variant of SP-Network [MvOV96] that consists of $T = 31$ rounds. Let SBOX be a 4-bit substitution box, and PBOX be a 64-bit permutation box. PRESENT can encrypt a $l_m = 64$-bit message block, and support two kinds of key lengths $l_k \in \{80, 128\}$.

Since the round key generation can be pre-computed on a PC, we just describe the realization of the rest of three main procedures: AddRoundKey, SBOXLayer, and PBOXLayer. The AddRoundKey is used to XOR the round keys with the current round state St initialized with the message in the first round. The function of SBOXLayer is to get each 4-bit input nibble from the St for the SBOX table lookup, which can be done via just four assignment operations because we can directly access each bit of a DINT variable with ST. Specifically, it can get the j-th 4-bit word of St as stmp.$[z]$:= St$[u]$.$[4*v+z]$, and obtain the substitution stmp := SBOX(stmp), where $z \in [4]$, $u = j/8$, and $v = j$ MOD 8. However, we observe that some parts of the steps of SBOXLayer and PBOXLayer can be merged following the optimization idea **B-MO**. To execute the complete SBOXLayer, each SBOX look-up result stmp should be assembled back to St (nibble-by-nibble), so that the final resulting state St would be taken as input to the PBOXLayer. Such assembling steps may require 64 assignments, which dominate 1/3 steps of the entire SBOXLayer and PBOXLayer (where an assignment statement roughly costs 1.17 μs). However, we figure out that it is possible to directly input the SBOX result stmp into PBOXLayer for permutation since the corresponding position of each bit of stmp in St is pre-determined.

We implement the encryption of PRESENT as Algorithm 3. For efficiency, we can also pre-compute the arithmetic operations in SBOXLayer(St) and PBOXLayer(St), i.e., $4*v+z$, $j/8$, j MOD 8, blk := PBOX$(j*4+\delta)/32$, and pos := PBOX$(j*4+\delta)$ MOD 32 in Step 4. Namely, we can implement SBOXLayer and PBOXLayer with only a few assignments after applying a **HC-WU**-like hard-coding strategy. We stress that the arithmetic operations involved in these steps shown in Algorithm 3 are just used here for demonstrating our intuition.

In Figure 4, we show a code snippet of the hard-coded SBOXLayer and PBOXLayer in our implementation of PRESENT. The involved SBOX is PT_Sbox = [12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2], and PBOX is PT_Pbox = [0, 16, 32, 48, 1, 17, 33, 49, 2, 18, 34, 50, 3, 19, 35, 51, 4, 20, 36, 52, 5, 21, 37, 53, 6, 22, 38, 54, 7, 23, 39, 55, 8, 24, 40, 56, 9, 25, 41, 57, 10, 26, 42, 58, 11, 27, 43, 59, 12, 28, 44, 60, 13, 29, 45, 61, 14, 30, 46, 62, 15, 31, 47, 63], where PT_Pbox is implied by the indices of the temporary tag PT_PState (in Figure 4) since it is hard-coded.

Taking the third times substitution result PT_State_tmp in Figure 4 as an example, the least significant bit PT_State_tmp.[0] of it corresponds to the 8-th bit of the result of SBOXLayer, so we could move it to the new position PBOX(8) = 2. Namely, this results in the assignment statement PT_PState[0].[2]:=PT_State_tmp.[0].

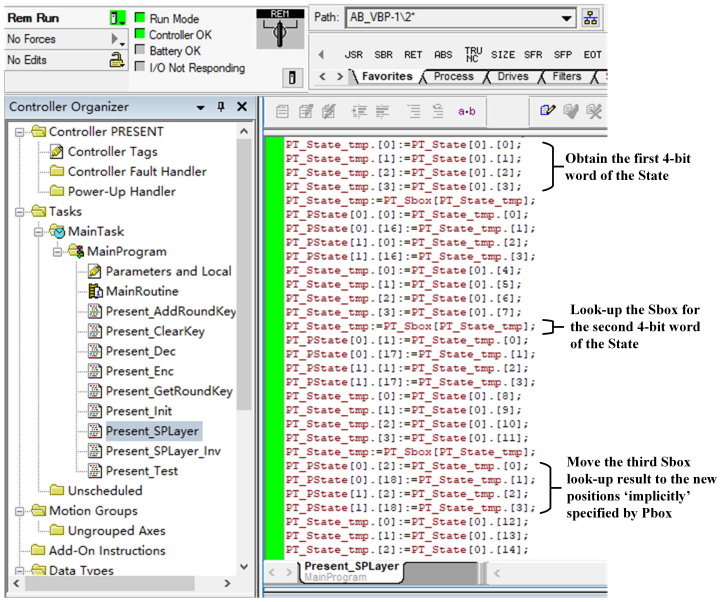


Figure 4: Code Snippet of the Hard-coded SBOXLayer and PBOXLayer of PRESENT

Implementation of SPECK and SIMON. The key techniques in our implementations are our tailored shifting/rotation operation and big-integer Add/Sub functions. The encryption algorithms of SPECK implemented on PLC only take a message as input since we hard-code the concrete values of pre-computed round encryption keys following the **HC-RL** strategy. To implement SPECK with a 128-bit message (each block having 64-bit), we leverage the big-integer addition (in encryption) and subtraction (in decryption) [MvOV96, Algorithm 14.9], respectively. The implementation of SIMON is similar to that of SPECK, which mainly relies on our hard-coded shifting/rotation function.

Implementation of PRF and PRG. Here we leverage block ciphers to realize both pseudo-random function (PRF) and pseudo-random generator (PRG), following the approach standardized in [SPLI06, EB07]. Note that Chaskey can be viewed as a PRF, and running in counter mode turns it into a PRG [MMH+14]. Hence, counter mode Chaskey is included in the comparison as well. We consider a PRG to have the same input as PRF, i.e., the PRG evaluation function has an additional input message $x \in \{0,1\}^{l_x}$ that could be the counter indexing the generated randomness.

Specifically, we utilize PRESENT, SPECK, and Chaskey to implement both PRF and PRG. To generate longer random bits for PRG, we leverage counter (CTR) mode in their operations.

Security Analysis. Due to the **HC-RL** hard coding of round keys, our block cipher implementations can resist TMA as well. From the algorithmic point of view, PRESENT, SPECK, and SIMON have no branches in the program. In addition, the hard-coded SBOXLayer/PBOXLayer, and the shifting/rotation operations in SPECK and SIMON run in constant time, so our implementations of block ciphers have constant runtime. Note that the security of our PRF/PRG is implied by that of the underlying block ciphers.

6.5 Implementations of Collision Resistant Hash Functions: PHOTON and SPONGENT

As described in the reference implementation of PHOTON [JG19], the most expensive three procedures *SubCell*, *ShiftRows*, and *MixColumnsSerial* in its core permutation function π_p can be pre-computed (by running the BuildTableSCShRMCS() function [JG19]), it can yield a two-dimensional $l_d \times 2^{l_s}$ table (SCShRMCS table) that can be used for online fast internal state calculation, where l_s is the bit-length of internal state. We adopt identical optimization in our implementation as well. Meanwhile, the SCShRMCS table is hard-coded with the **HC-RL** approach since it is not very large.

Since SPONGENT has a PRESENT-like permutation, we mainly hard-coded the entire PBOX layer with a similar idea in the PRESENT implementation. This roughly saves 4x computational cost comparing to straightforwardly translate reference implementation of SPONGENT [BKL+12] into ST code.

Security Analysis. Similar to the implementations of block ciphers, we protect the parameters (i.e., SBOX and PBOX) of both schemes using the hard coding strategies. Moreover, our implementations have constant runtime by their algorithm designs and our hard coding tricks.

7 Case Study: Proof of Aliveness

Background. Proof of aliveness (PoA) is a cryptographic notion that was recently proposed by Jin *et al.* [JYvDZ19]. Although PoA was proposed to attest the aliveness (working status) of CPS devices like PLC, to the best of our knowledge, it has never been implemented on a commercial off-the-shelf PLC. Here we briefly introduce the advanced PoA protocol $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ in [JYvDZ19]. $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ is composed of two procedures: i) proof generation; and ii) proof replenishment, where proof generation algorithm is used to generate a publicly verifiable proof every Δ_s seconds to attest its aliveness, and proof replenishment algorithm is used to re-initialize a new protocol instance when the proofs of the current instance are used up.

The protocol $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ has a multiple-chain structure to generate proofs, where the number of the sub-chain is denoted by a parameter η. The i-th sub-chain of $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ is an OWF-chain that starts from a head p^i_0 and ends at the tail p^i_N where N is the number of nodes in a sub-chain. The tail of the sub-chain is known by the verifier, and the nodes in sub-chains are periodically sent from the prover to the verifier in reverse order (from the tail to the head) as aliveness proofs. The heads of all OWF sub-chains are linked like a chain of random numbers generated by a PRG. The replenishment of a protocol instance means that the prover will select a new seed for PRG, and create a new multiple-chain structure, and commit all the tails (for verifying the new protocol instance) of sub-chains to the verifier using a one-time signature (OTS) scheme, whose signing keys are the sub-chain heads of the current instance.

A Tag Manipulation Attack on PoA. We stress that it is not secure to implement the PoA on PLCs directly following the original specification. According to the original design [JYvDZ19], we have to store some critical tags during the full lifetime of the protocol,

e.g., protocol instance counter P and sub-chain counter S. However, since these tags need to be updated frequently, they cannot be hard-coded in the program. Attackers can tamper with the tags, i.e., attackers can get any unused proofs by manipulating either P or S to trick PLCs to generate the future proofs that should not be generated at the current time.

Modifications of the Original PoA Protocol. To protect critical tags, we can generate a MAC value of tags and check their integrity at the beginning of a new scan cycle. Because of the reading/writing capability of network attackers, we cannot cache the proofs (in the memory) as in [JYvDZ19]; otherwise, all future proofs cached in the memory will be read by the attackers.

Implementation. To implement $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ in [JYvDZ19], we used $\mathsf{F}_{\mathrm{sss}}$, PRG realized by Chaskey-CTR, Chaskey, and H_{sss} in *PLCrypto*. The head nodes of sub-chains are generated by PRG, as $p_0^i := \mathsf{PRG}(\mathrm{P}||i||l_r)$. The OWF is instantiated with $\mathsf{F}_{\mathrm{sss}}$ with $l_r = 256$ (which is much more efficient than any other cryptographic hash functions like PHOTON on PLC). To replenish proofs, we run the initialization algorithm on PLC first and then sign the new verification state (tails) using Lamport's one-time signature (OTS) [Lam79] using the sub-chain heads of signing keys. So the minimum number of sub-chains is 256, i.e., the number of signing keys of OTS. Before signing the tail nodes, we compress them using UOWHF H_{sss} as a message domain extender [SPW06]. Also, we need to run the whole initialization in one scan cycle to avoid calculating MAC for too many tags. Fortunately, we can choose to use a relatively short sub-chain due to the replenishment feature, so the computation cost of the initialization is adjustable as demanded.

Security Analysis. The security of our PoA is guaranteed by the secure implementation of the concrete instances of its building blocks. The integrity of the counter between scan cycles relies on the security of the MAC scheme.[5]

Deployment Cost. Being a software-only security solution, *PLCrypto* does not require additional hardware components or firmware updates. Also, we do not need to modify the communication protocols. We just need to add a line in the original control program to call a function in *PLCrypto* with inputs.

8 Benchmark

Benchmark Settings. To show the performance of the cryptographic algorithms in *PLCrypto*, we implemented them on a mainstream commercial PLC [Pro18] from Allen Bradley, which has a controllogix 5571 processor and 2 megabytes (MB) memory. We benchmarked our algorithms with various parameters for comparison. We measured the time by averaging the results of 1000 repeated experiments if not specified separately. The execution time is measured using a built-in GSV (Get System Values) instruction to get the execution time of a scan cycle, which only contains the function under the test. The time in the average case and the worst case are reported separately as average/worst.

Correctness Verification. Our implementations are parameterized, so users can select the parameter settings which fit their applications most. To demonstrate the performance of our implementation, we chose a few widely-used security parameters (e.g., 64, 128, 256) to instantiate the implementations for benchmarking. We used the test vectors in the original papers or the reference implementations provided by algorithm inventors to test the correctness of our implementations (except for the subset sum problem based primitives, we wrote our own reference program to verify its correctness). When we are

[5] Note that, as a rule of thumb, before implementing any new protocols or algorithms that comprise algorithms in *PLCrypto*, one has to analyze the security of the new implementation based on the principles in Section 5.3.

Table 4: Runtimes of Important Atomic Operations on 32-bit DINT Variables

Instructions	**Runtime** (μs)
:=	1.17
+,-	1.51
*	2.65
/	2.99
MOD	3.10
XOR, AND, OR	2.3
**	31.5

testing the performance of *PLCrypto*, we used the same reference test vectors as mentioned above.

Performance of Atomic Operations. In Table 4, we show the benchmark results of some important atomic operations on 32-bit DINT variables on the experimented PLC. We measure the performance of each operation by averaging over 10,000 executions.

Runtimes of OWF. We implemented $\mathsf{F}_{\mathsf{sss}}$ with three input lengths $l_n \in \{256, 384, 512\}$. Since the performance of $\mathsf{F}_{\mathsf{sss}}$ is determined by the number of "1" bit in the input, we experimented with inputs having $\frac{l_n}{2}$ ones (as the average case) and l_n ones (as the worst case). We benchmarked our two implementation solutions: the "baseline" OWF implementation as Algorithm 1, and the improved solution ("Imp-IF") based on hard-coded If statements. The performance of subset-sum based OWF is shown in Table 5a in milliseconds (ms).

The subset-sum based OWF is efficient on PLC, which only needs a few milliseconds for different input lengths. The baseline implementation of the OWF with $l_n = 256$ costs 14.7 ms in the average case and 24.0 ms in the worst case. Clearly, our performance improvement is around two times. Nevertheless, the results show that $\mathsf{F}_{\mathsf{sss}}$ has much better performance on PLC than on other platforms, e.g., ARM [JYvDZ19].

Regarding H_{sss}, we first benchmarked the fixed message length version that is implemented based on "Imp-IF"-style $\mathsf{F}_{\mathsf{sss}}{}'$, whose performance of H_{sss} is also shown in Table 5a. In Figure 5, we show the performance of H_{sss} with message lengths varying from 0.5 kilobits (Kb) to 8 Kb, representing the commonly used parameters in most applications of PLC.

Table 5: Runtimes of $\mathsf{F}_{\mathsf{sss}}$, H_{sss} and Block Biphers

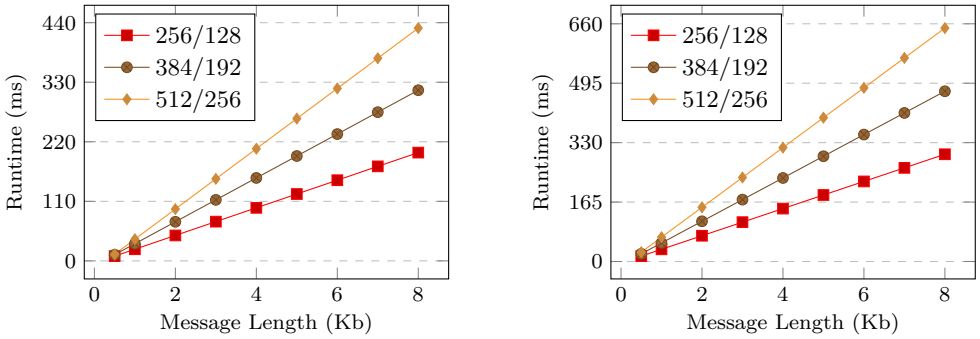
(a) $\mathsf{F}_{\mathsf{sss}}$ and H_{sss}

l_n	**Evaluation time (ms)**		
	OWF		UOWHF
	Baseline	Imp-IF	
256	14.7 / 24.0	7.9 / 15.6	3.9 / 7.8
384	30.7 / 51.3	17.6 / 35.0	8.8 / 17.5
512	52.5 / 88.7	31.2 / 62.0	15.6 / 31.0

(b) Block Ciphers

l_m/l_k	**Enc/Dec time (ms)**		
	PRESENT	SPECK	SIMON
32/64	-	1.6	3.4
64/128	7.1	4.6	7.8
128/128	-	9.5	23.7
128/256	-	10.4	25.8

Runtimes of Chaskey. The performance of Chaskey with different permutation rounds and message lengths is shown in Figure 6 (a). For $l_m = 128$ and $T = 8$, the evaluation time is about 2.7 ms, that is efficient enough to authenticate messages collected from sensors in almost real-time. Since the computation cost is dominated by the permutation algorithm π_c, it is not surprising that the performance Chaskey with 8-round permutation is 2x faster than that with 16-round (but the latter provides stronger security). Besides, we show that the performance of Chaskey with $T = 12$ rounds (recommended by ISO/IEC 29192-6:2019)



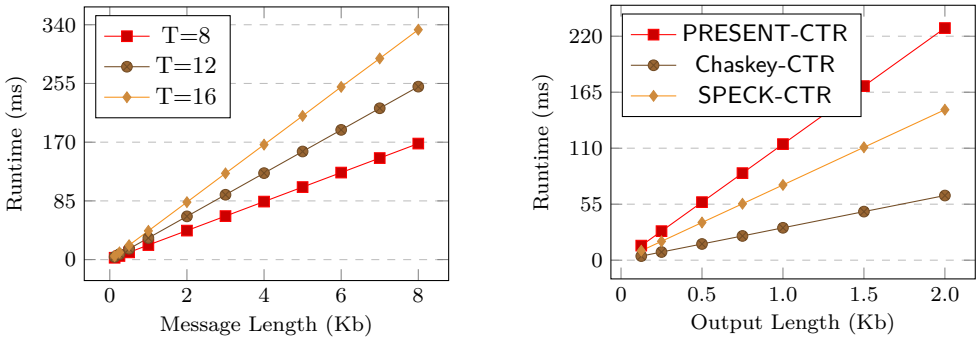
(a) Average **(b)** Worst

Figure 5: Runtimes of H_{sss} with Variable Message Lengths.

is between that of $T = 8$ and $T = 16$.

Runtimes of PRESENT. PRESENT supports either 80-bit or 128-bit key, but the difference between the two ciphers is only the key schedule procedure that can be done on PC. Hence, the encryption/decryption time on PLC for two different keys is identical. Through our hard-coding optimization towards PRESENT, it is no longer the slowest one among the three implemented block ciphers, as opposed to the results in [BSS+13], regarding software implementations. For example, PRESENT is 8x slower than SPECK and 5.6x slower than SIMON in [BSS+13], but we reduce the performance gap between PRESENT and SPECK to be 1.5x on PLC, and it is even faster than SIMON in our benchmark. Hence, PRESENT seems to be more suitable in the PLC environment. The encryption/decryption performance of PRESENT (around 7.1 ms) is fast enough for most CPS applications.

Runtimes of SPECK and SIMON. We benchmarked SPECK and SIMON with some typical parameters that are multiples of 32 (i.e., the bit-length of DINT) so that no bits in a DINT variable are wasted. The benchmark results of SPECK and SIMON on PLC are listed in Table 5b. SPECK and SIMON are also efficient on PLC since their encryption and decryption algorithms only consist of a few hard-coded rotations and XOR operations, which are very fast. This is also why we include them in *PLCrypto*. However, SPECK is more efficient than SIMON since it requires less shift/rotate and logical operations in each round, and it has fewer rounds than that of SIMON.

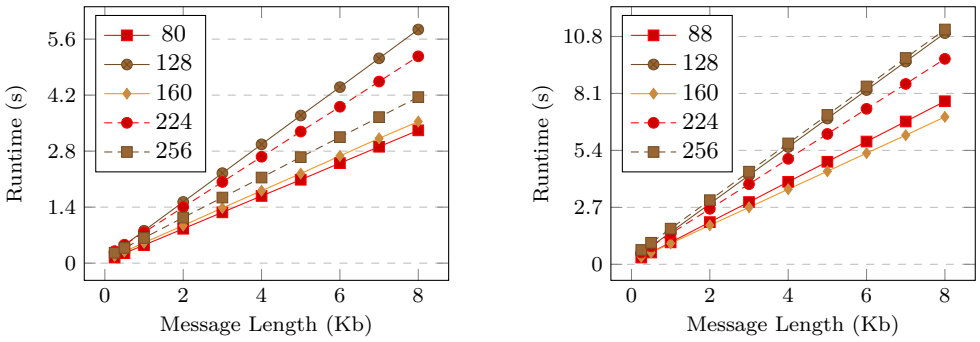


(a) Chaskey **(b)** PRF/PRG

Figure 6: Runtimes of Chaskey and PRF/PRG.

Runtimes of PRF and PRG. We benchmarked PRF/PRG with three concrete instantiations, PRESENT-CTR, SPECK-CTR, and Chaskey-CTR. For efficiency and security consideration, we chose the message length to be $l_m = 64$ for PRF and PRG. The message length of Chaskey is 128 bits. We set the key size of PRF/PRG/Chaskey to be $l_k = 128$ to meet practical security requirements. The performance of PRESENT-CTR, SPECK-CTR, and Chaskey-CTR is shown in Figure 6 (b). The performance is efficient and linear in the size of the generated randomness. To generate 256 bits randomness (e.g., as required in PoA), the most efficient instantiation Chaskey-CTR roughly needs 8.0 ms.

Runtimes of PHOTON and SPONGENT. The performance of the five most efficient ciphersuits of PHOTON and SPONGENT is shown in Figure 7 (a) and (b), respectively, where we use the output length to represent each implemented ciphersuit. In general, PHOTON is almost 2x faster than SPONGENT on PLC. PHOTON-80 is the fastest version, but it still needs about 170 ms to hash a 256-bit message and 1s to hash 2 Kb message. Though PHOTON-256 can provide stronger security (i.e., 128-bit collision-resistance) and is the third fastest version, it requires a much bigger SCSHRMCS table which roughly consumes 100 KB of memory (in contrast to the 4 KB required by PHOTON-160). For storage-constrained PLCs, PHOTON-160 provides a good trade-off between security and performance. Nevertheless, PHOTON might be used when collision-resistance is necessary and the task is not very time-critical.

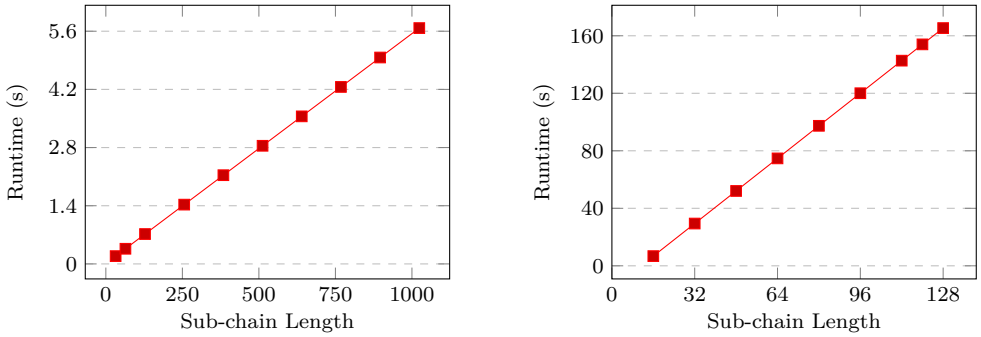


(a) PHOTON **(b)** SPONGENT

Figure 7: Runtimes of PHOTON and SPONGENT.

Runtimes of PoA. Firstly, we need to instantiate those parameters (i.e., the number η of sub-chains and the number N of the proofs in a sub-chain) to make all algorithms executable on PLC. It is not hard to see that the proof replenishment algorithm is more costly than the proof generation algorithm since it needs to run the whole initialization procedure at once. Therefore, we need to choose the parameters based on the cost of the replenishment. We set $\eta = 256$, which is the lower bound of the number of sub-chains for running the OTS. But we adjust the total number of proofs, i.e., N, from 64 to 128 when we evaluated the proof replenishment algorithm. For proof generation, we experimented with a longer sub-chain to test its performance bound. The performance of PoA on a PLC is shown in Figure 8. The proof generation time is below 6 seconds, even if $N = 1000$, while, in practice, the interval between two consecutive proof generations can be greater than 30 seconds [JYvDZ19]. When $N = 128$, the replenishment time is about 3 minutes. For these concrete parameters $N = 128$ and $\eta = 256$, a protocol instance has 32768 proofs that can be used for roughly 11 days. A PLC only needs to run the proof replenishment algorithm in the idle time every 11 days for 3 minutes before the proofs run out.

Storage Costs of Algorithms. Table 6 summarizes the storage costs of the implemented algorithms. For simplicity, we present the version of algorithms or the storage costs in



(a) Proof generation **(b)** Proof replenishment

Figure 8: Runtimes of Proof of Aliveness (PoA).

brackets, e.g., [256 , 384 , 512] stands for three input lengths of OWF. We only list the storage costs of some typical ciphersuits of algorithms. The storage costs are broken down into the size of the ST code of an algorithm and the tags that are created on PLC for executing the algorithms. The storage cost of most of the algorithms is less than 100 KB, which well fits the memory size of a commercial PLC (e.g., 2MB on controllogix 5571 [Pro18]).

Practicality Discussion. Our library is the first-of-the-kind, and we open-source our codes to encourage others to improve our implementation with respect to its performance and security. The practically acceptable range of runtime and memory consumption certainly depends on applications and devices. On the commercial PLC we tested, our memory consumption (shown in Table 6) is well below the total memory size (2 MB) of the PLC. The runtime of most of the algorithms in *PLCrypto* is on the orders of milliseconds. Usually, PLCs and their monitoring servers do not communicate very often, so *PLCrypto* has enough time to compute before sending data to the servers. For example, in a water treatment system, PLC data is collected once per second [GAJM16]. As another example, the PLCs controlling train systems report their status every 300 ms [JAYZ21]. Using Speck 64/128 for encryption and Chaskey with T = 16, one only needs 30 ms to encrypt and generate a MAC tag for 128-bit data.

Table 6: Storage Cost of Algorithms in *PLCrypto*

Algorithms		Storage (KB)		
		Code	Tag	Total
OWF [256 , 384 , 512]	Original	[67.5 , 145.0 , 254.0]	[9.1 , 19.6 , 34.2]	[76.6 , 164.6 , 288.2]
	Imp-IF	[267.0 , 575.0 , 993.2]	[0.1 , 0.1 , 0.2]	[267.1 , 575.1 , 993.4]
UOWHF [256 , 384 , 512]	Fix	[163.0 , 335.0 , 568.0]	[0.1 , 0.1 , 0.1]	[163.1 , 335.1 , 568.1]
	Vary	[164.0 , 336.0 , 569.0]	[1.1 , 1.1 , 1.2]	[165.1 , 337.1 , 570.2]
Chaskey		[12.0]	[1.2]	[13.2]
Block Ciphers [32 , 64 , 128]	PRESENT	[- , 14.3 , -]	[- , 0.5 , -]	[- , 14.8 , -]
	SPECK	[4.9 , 11.1 , 16.2]	[0.3 , 0.3 , 0.4]	[5.2 , 11.4 , 16.6]
	SIMON	[6.4 , 9.7 , 22.7]	[0.3 , 0.4 , 0.6]	[6.7 , 10.1 , 23.3]
PRF/PRG	PRESENT-CTR	[14.6]	[0.8]	[15.4]
	SPECK-CTR	[12.0]	[0.7]	[12.7]
	Chaskey-CTR	[17.5]	[2.2]	[19.7]
PHOTON [80 , 160 , 256]		[19.6 , 21.9 , 116]	[13.1 , 13.5 , 24.9]	[32.7 , 35.4 , 140.9]
SPONGENT [88 , 160 , 256]		[23.9 , 23.5 , 27.1]	[11.6 , 11.6 , 13.2]	[35.5 , 35.1 , 40.3]
PoA	Proof Generation	[285.0]	[83.6]	[368.6]
	Replenishment	[457.0]	[93.3]	[550.3]

9 Conclusion and Future Work

We implemented an efficient and secure cryptographic library *PLCrypto* for PLC based on ST. *PLCrypto* includes a wide range of symmetric cryptographic algorithms for realizing essential cryptographic functions (such as OWF, BC, PRF/PRG, MAC, Hash, and PoA). To use *PLCrypto* in practice, users can either build the application directly on the main-task of an algorithm in *PLCrypto* or copy the routines of cryptographic algorithms from *PLCrypto* to a target application.

To further extend this line of research, one can investigate the possibility of side-channel attacks on *PLCrypto* and enhance the library with side-channel resistance. Also, one can extend *PLCrypto* to include more algorithms, e.g., asymmetric cryptographic algorithms. We also encourage researchers to propose novel implementation tricks to further improve the performance of our open-source *PLCrypto*. Cryptographers are encouraged to develop new lightweight algorithms that better fit the programming constraints on the PLC platform. Due to both security and performance considerations, we used hard-coding techniques in our implementations, but hard-coding techniques also prevent us from frequently updating secret keys of algorithms in *PLCrypto*. One possible way to address this would be to have an authenticated encryption with associated data with hard-coded key, which would be used for pre-computed round key wrapping; the encrypted round keys can be stored in tags or memory between cryptographic function calls.

Acknowledgments

We would like to thank our shepherd Nicky Mouha and anonymous reviewers for their invaluable comments and suggestions. Zhe Liu is supported by the National Key R&D Program of China (Grant No.2020AAA0107700) and the National Natural Science Foundation of China (Grant No.61802180). Zheng Yang is supported by the Natural Science Foundation of China (Grant No. 61872051). Partial work of Zheng Yang and Zhiting Bao was done when they worked at SUTD.

References

[ADM18] Thiago Alves, Rishabh Das, and Thomas Morris. Embedding encryption and machine learning intrusion prevention systems on programmable logic controllers. *Embedded Systems Letters*, 10(3):99–102, 2018.

[Alv20] Thiago Alves. The openplc project, 2020. [Accessed May., 2020].

[AM18] Sridhar Adepu and Aditya Mathur. Assessing the effectiveness of attack detection at a hackfest on industrial control systems. *CoRR*, abs/1809.04786, 2018.

[AMY17] Thiago Alves, Thomas Morris, and Seong-Moo Yoo. Securing scada applications using openplc with end-to-end encryption. In *Proceedings of the 3rd Annual Industrial Control System Security Workshop*, pages 1–6, 2017.

[Ass06] American Gas Association. Cryptographic protection of scada communication, 2006. [Accessed May., 2020].

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: an ultra-lightweight block cipher. In *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

[BKL+12] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. Source codes of spongent. https://sites.google.com/site/spongenthash/downloads, 2012.

[BKL+13] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: the design space of lightweight cryptographic hashing. *IEEE Trans. Computers*, 62(10):2041–2053, 2013.

[Bra09a] Allen Bradley. Logix5000 controllers. 2009.

[Bra09b] Allen Bradley. Logix5000 controllers tasks, programs, and routines. 2009.

[Bra16] Allen Bradley. Memory card usability on logix5000 controllers. 2016.

[Bra18a] Allen Bradley. Logix 5000 controllers general instructions reference manual. 2018.

[Bra18b] Allen Bradley. Logix 5000 controllers structured text. 2018.

[Bra19] Allen Bradley. Cip security with rockwell automation products, 2019. [Accessed Jun., 2020].

[Bra20] Allen Bradley. Stratix ethernet device specifications. 2020.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. *IACR Cryptology ePrint Archive*, 2013:404, 2013.

[CATO17] John Henry Castellanos, Daniele Antonioli, Nils Ole Tippenhauer, and Martín Ochoa. Legacy-compliant data authentication for industrial control system traffic. In *International Conference on Applied Cryptography and Network Security*, pages 665–685. Springer, 2017.

[CG20] Jean-Sébastien Coron and Agnese Gini. A polynomial-time algorithm for solving the hidden subset sum problem. *IACR Cryptol. ePrint Arch.*, 2020:461, 2020.

[CHA19] *ISO/IEC 29192-6:2019 Information technology âĂŤ Lightweight cryptography âĂŤ Part 6: Message authentication codes (MACs)*. ISO/IEC, 2019.

[Dec18] Wolfgang Decker. Top tips for retrofitting legacy equipment, 2018. [Accessed Jun., 2020].

[Deu17] Deutsche Bank Research. Global PLC market share as of 2017, by manufacturer. https://www.statista.com/statistics/897201/global-plc-market-share-by-manufacturer/, 2017. Online; Last accessed the website in May 2020.

[DG17] Adrian-Vasile Duka and Béla Genge. Implementation of simon and speck lightweight block ciphers on programmable logic controllers. In *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*, pages 1–6. IEEE, 2017.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[DRX17] Srinivas Devadas, Ling Ren, and Hanshen Xiao. On iterative collision search for LPN and subset sum. In *TCC (2)*, volume 10678 of *Lecture Notes in Computer Science*, pages 729–746. Springer, 2017.

[EB07] John Kelsey Elaine Barker. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*. NIST Special Publication, 2007.

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[GAJM16] Jonathan Goh, Sridhar Adepu, Khurum Nazir Junejo, and Aditya Mathur. A dataset to support research in the design of secure water treatment systems. In *International Conference on Critical Information Infrastructures Security*, pages 88–99. Springer, 2016.

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In *CRYPTO*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[IN96] Russell Impagliazzo and Moni Naor. Efficient cryptographic schemes provably as secure as subset sum. *J. Cryptology*, 9(4):199–216, 1996.

[JAYZ21] Edwin Franco Myloth Josephlal, Sridhar Adepu, Zheng Yang, and Jianying Zhou. Enabling isolation and recovery in plc redundancy framework of metro train systems. *International Journal of Information Security*, Jan 2021.

[JG16] Axel Poschmann Jian Guo, Thomas Peyrin. Photon is a very lightweight family of hash functions (part of iso/iec 29192-5:2016). 2016.

[JG19] Axel Poschmann Jian Guo, Thomas Peyrin. Source codes of photon. https://sites.google.com/site/photonhashfunction, 2019.

[JT13] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer Berlin Heidelberg, 2013.

[JVvD18] Chenglu Jin, Saeed Valizadeh, and Marten van Dijk. Snapshotter: Lightweight intrusion detection and prevention system for industrial control systems. In *2018 IEEE Industrial Cyber-Physical Systems*, pages 824–829. IEEE, 2018.

[JYAZ19] Chenglu Jin, Zheng Yang, Sridhar Adepu, and Jianying Zhou. Hmake: Legacy-compliant multi-factor authenticated key exchange from historical data. *IACR Cryptology ePrint Archive*, 2019:450, 2019.

[JYvDZ19] Chenglu Jin, Zheng Yang, Marten van Dijk, and Jianying Zhou. Proof of aliveness. In *ACSAC*, pages 1–16. ACM, 2019.

[Lam79] Leslie Lamport. Constructing digital signatures from a one-way function. Technical report, Technical Report CSL-98, SRI International Palo Alto, 1979.

[Mir18] Miracl cryptographic library, 2018.

[MMH+14] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An efficient MAC algorithm for 32-bit microcontrollers. In *Selected Areas in Cryptography*, volume 8781 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2014.

[MvOV96] Alfred Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[MVOV18] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[NAI17] Naoki Nishikawa, Hideharu Amano, and Keisuke Iwai. Implementation of bitsliced AES encryption on cuda-enabled GPU. In *NSS*, volume 10394 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2017.

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *STOC*, pages 33–43. ACM, 1989.

[ODV19] ODVA. Cip security, 2019. [Accessed Jun., 2020].

[PRE19] *ISO/IEC 29192-2:2019 Information security - Lightweight cryptography - Part 2: Block ciphers*. ISO/IEC, 2019.

[Pro18] NHP Electrical Engineering Products. Catalogue no: 1756-l71. Technical report, 1756L71 Datasheet, 2018.

[RBB00] Vincent Rijmen, Antoon Bosselaers, and Paulo Barreto. Optimised Rijndael implementation. https://archive.org/details/rijndael-fst-3.0, 2000.

[Roc20] Rockwell Automation. Programmable Controllers. https://ab.rockwellautomation.com/Programmable-Controllers, 2020. Online; Last accessed the website in May 2020.

[RSD06] Chester Rebeiro, A. David Selvakumar, and A. S. L. Devi. Bitslice implementation of AES. In *CANS*, volume 4301 of *Lecture Notes in Computer Science*, pages 203–212. Springer, 2006.

[SHB09] Emily Stark, Michael Hamburg, and Dan Boneh. Symmetric cryptography in javascript. In *ACSAC*, pages 373–381. IEEE Computer Society, 2009.

[SPE18] *ISO/IEC 29167-21:2018 Information technology âĂŤ Automatic identification and data capture techniques âĂŤ Part 21: Crypto suite SIMON security services for air interface communications*. ISO/IEC, 2018.

[SPLI06] JunHyuk Song, Radha Poovendran, Jicheol Lee, and Tetsu Iwata. The advanced encryption standard-cipher-based message authentication code-pseudo-random function-128 (AES-CMAC-PRF-128) algorithm for the internet key exchange protocol (IKE). *RFC*, 4615:1–7, 2006.

[SPW06] Ron Steinfeld, Josef Pieprzyk, and Huaxiong Wang. Higher order universal one-way hash functions from the subset sum assumption. In *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 157–173. Springer, 2006.

[Sta16] Morgan Stanley. The internet of things and the new industrial revolution, 2016. [Accessed Jun., 2020].

[YH20] Eric A. Young and Tim J. Hudson. The openssl project, 2020. [Accessed May., 2020].

A Other Preliminaries

A.1 Chaskey

Here we review the first algorithm of Chaskey [MMH+14, Algorithm 3] that is the most efficient one for PLC. Chaskey consists the following two algorithms:

- *Key Generation*: This algorithm first randomly samples l_k-bit key $k \xleftarrow{\$} \{0,1\}^{l_k}$. Then it would derive two new sub-keys K_1 and K_2 from k for MAC evaluation. If $k[l_k - 1] = 0$ then $K_1 := (K \ll 1) \oplus 0^{l_k}$, else $K_1 := (k \ll 1) \oplus 0^{l_k-8}10000111$. And if $K_1[l_k - 1] = 0$ then $K_2 := (K_1 \ll 1) \oplus 0^{l_k}$, else it sets $K_2 := (K_1 \ll 1) \oplus 0^{l_k-8}10000111$. Finally, this algorithm returns key $K = (k||K_1||K_2)$. Chaskey [MMH+14, Algorithm 3] is designed with 128-bit concrete security, so we will use $l_k = 128$ accordingly in the sequel.
- *Evaluation*: We denote this algorithm by $\mathsf{Chaskey}(K, m)$ which takes as input a key $K = (k||K_1||K_2)$ and a message $m \in \{0,1\}^{l_m}$, and outputs a MAC value $\tau \in \{0,1\}^{l_k}$. The evaluation procedure will leverage on a permutation π_c which is defined later. Based on π_c, the evaluation algorithm first splits the input message m into ℓ blocks $m_0, m_1, \ldots, m_{\ell-1}$ of l_k-bit each. It then initializes $\mathrm{ST}_1 := k$ and computes $\mathrm{ST}_{i+1} := \pi_c(\mathrm{ST}_i \oplus m_i)$ for $i \in [\ell - 1]$. As for the last block of m, if $|m_{\ell-1}| = l_k$, then it sets $L := K_1$, otherwise its sets $L := K_2$ and $m_{\ell-1} := m_{\ell-1}||10^{l_k-|m_{\ell-1}|-1}$ (for padding). Eventually, it computes $\mathrm{ST}_\ell := \pi_c(\mathrm{ST}_{\ell-1} \oplus m_{\ell-1} \oplus L) \oplus L$. The MAC value τ is the t least significant bits of ST_ℓ.
  The permutation function π_c used above π_c consists of $T \in \{8, 16\}$ rounds, where 16-round is recommended for long-term security. Each round runs the following steps based on four 32-bit input variables v_0, v_1, v_2, and v_2:
  - $v_0' := ((v_1 + v_2) \lll 16) + ((v_3 \lll 8) \oplus (v_3 + v_2))$;
  - $v_1' := (((v_1 \lll 5) \oplus (v_1 + v_0)) \lll 7) \oplus ((v_3 + v_2) + ((v_1 \lll 5) \oplus (v_1 + v_0)))$;
  - $v_2' := (((v_3 + v_2) + ((v_1 \lll 5) \oplus (v_1 + v_0))) \lll 16)$;
  - $v_3' := (((v_1 + v_0) \lll 16) + (v_3 \lll 8)) \oplus (v_3 + v_2) \oplus ((v_3 \lll 8) \oplus (v_3 + v_2) \lll 13)$.

A.2 Block Ciphers

In this section, we review three lightweight block cipher families that are implemented in *PLCrypto*.

A.2.1 PRESENT

We review the algorithms of PRESENT as follows:

- *Key Generation*: This algorithm first randomly samples a key $k \xleftarrow{\$} \{0,1\}^{l_k}$ that shall be loaded into the key register for generating the subsequent encryption keys used in each round. Each round encryption key is the left-most 64-bit of the current key register which will be updated by a key scheduling procedure to generate the next round encryption key. The key schedule depends on the key length l_k. The main idea of the key scheduling is to rotating left the key register with 61 bits, partial bits the of the key register are passed through the SBOX (e.g., the left-most four bits for 80-bit key), and the round counter value i is exclusive-ored with the corresponding bits of the key register. The detailed key scheduling procedure is detailed in [BKL+07]. This algorithm will generate 32 round encryption keys $k_0, \ldots, K_{T-1}$ where $T = 32$.
- *Encryption*: We denote this algorithm by $\mathsf{PRESENT}_{\mathrm{enc}}(K, m)$ which takes as input an encryption key $K = (k_0|| \ldots ||k_{T-1})$ and a 64-bit message m, and outputs a 64-bit block ciphertext C. The encryption algorithm has 31-SP rounds, and each round runs three procedures: *AddRoundKey*, *SBOXLayer*, and *PBOXLayer*, which are executed one after the other. AddRoundKey takes as input the 64-bit round state St and the round encryption key k_i, and outputs the updated state St that is computed as $\mathrm{St}[i] := \mathrm{St}[i] \oplus k_i$ for $0 \le i \le 63$, where the initial round state is the message. For SBOXLayer, the input state St is represented as sixteen 4-bit words $w_{15}, \ldots, w_0$ where $w_i = \mathrm{St}[4i + 3]||\mathrm{St}[4i + 3]||\mathrm{St}[4i + 3]||\mathrm{St}[4i + 3]$ for $0 \le i \le 15$ and the output nibble $\mathsf{SBOX}[w_i]$ yields the updated words of St. PBOXLayer is a bit permutation procedure that moves the i-th bit of the input state St into bit position specified

by $\mathsf{PBOX}(\mathrm{St}[i])$. After the SP rounds, the encryption algorithm runs an additional AddRoundKey procedure with key k_T to obtain the final ciphertext C.

- *Decryption*: We denote this algorithm by $\mathsf{PRESENT}_{\mathrm{dec}}(K', C)$ which takes as input the encryption key $K = (k_{T-1}||\ldots||k_0)$ and a 64-bit ciphertext C, and outputs message m. The decryption algorithm is a reverse execution of encryption algorithm. That is, it runs 31 times AddRoundKey, PBOXLayer^{-1} and SBOXLayer^{-1}, and one additional AddRoundKey procedure, where PBOXLayer^{-1} and SBOXLayer^{-1} are inverse of the permutation box and sub-situation box, respectively.

A.2.2 SPECK

We let l_e be the bit-length of a word used by them, where l_e could be 16, 24, 32, 48, and 64. So each block of message has a bit-length $l_m = 2l_e$. Both SPECK and SIMON have encryption key of l_w-word, i.e., $l_k = l_e l_w$ bits, $l_w \in \{2, 3, 4\}$. We shall denote a concrete ciphersuit $\Xi \in \{\mathrm{SIMON}, \mathrm{SPECK}\}$ as Ξ-l_m/l_k. In the following, we review them respectively.

The parameters of SPECK include two right operands of rotation operations, denoted by α and β, which are constants determined by the parameter l_e. Specifically, if $l_e = 16$ (32-bit block) then we set $\alpha := 7$ and $\beta := 2$; otherwise we have $\alpha := 8$ and $\beta := 3$. In the following, we review algorithms of SPECK.

- *Key Generation*: This algorithm first randomly samples a key $k \xleftarrow{\$} \{0,1\}^{l_k}$, which is divided into words as $k = (k_0, k'_0, \ldots, k'_{l_w-2})$. Next, this algorithm expand the key k to generate additional keys $(k_0, \ldots, k_{T-1})$ that shall be used later in the T rounds encryption procedure, where $T \in \{22, 23, 26, 27, 28, 32, 33, 34\}$ is determined by l_e and l_w (see more details in [BSS+13, Figure 4.4]). Specifically, for $i \in [T-1]$, it generates keys as $k'_{i+l_w-1} := (k_i + (k'_i \ggg \alpha)) \oplus i$ and $k_{i+1} := (k_i \lll \beta) \oplus k'_{i+l_w-1}$. Finally, this algorithm returns key $K = (k_0||\ldots||k_{T-1})$.
- *Encryption*: We denote this algorithm by $\mathsf{SPECK}_{\mathrm{enc}}(K, m)$ which takes as input an encryption key $K = (k_0||\ldots||k_{T-1})$ and a $2l_e$-bit message $m = (m_0||m_1)$ (that is divided into two equal length sub-messages), and outputs a $2l_e$-bit block ciphertext $C = (c_0||c_1)$. Each round of encryption carries out a round function $\mathsf{RF}_k(x, y) = ((x \ggg \alpha) + y) \oplus k, (y \lll \beta) \oplus (x \ggg \alpha) + y) \oplus k)$. The entire encryption procedure is the composition $\mathsf{RF}_{k_{T-1}} \circ \mathsf{RF}_{k_{T-2}} \circ \ldots \circ \mathsf{RF}_{k_0}$, where in the initial round $x = m_1$ and $y = m_0$, and in final round $c_1 = x$ and $c_0 = y$.
- *Decryption*: We denote this algorithm by $\mathsf{SPECK}_{\mathrm{dec}}(K', C)$ which takes as input the encryption key $K' = (k_{T-1}||\ldots||k_0)$ and a $2l_e$-bit ciphertext $C = (c_0||c_1)$, and outputs message $m = (m_0||m_1)$. The decryption algorithm mainly leverages on an inverse of round function (used in the encryption), i.e., $\mathsf{RF}_k^{-1}(x, y) = ((x \oplus k) - ((x \oplus y) \ggg \beta)) \lll \alpha, ((x \oplus y) \ggg \beta)$, the rounds keys are used in a reverse order for decryption.

A.2.3 SIMON

It has the same message, key, and ciphertext spaces as SPECK. It consists the following three algorithms.

- *Key Generation*: This algorithm first randomly samples a encryption key $k \xleftarrow{\$} \{0,1\}^{l_e l_w}$, which is divided into words as $k = (k_0, \ldots, k_{l_w-1})$. Next, this algorithm expands the encryption key k to generate additional round keys $(k_{l_w}, \ldots, k_{T-1})$ that shall be used later in the $T \in \{32, 36, 42, 44, 52, 54, 68, 69, 72\}$ rounds encryption procedure. Specifically, for $i \in [l_w, \ldots, T-1]$, it runs the following steps to obtain the expansion: i) If $l_w = 5$ then $tmp = k_{i-1} \ggg 3) \oplus k_{i-3}$; ii) $tmp = tmp \oplus (tmp \ggg 1)$; iv) $k_i =\sim k_{i-l_w} \oplus tmp \oplus z_j[(i - l_w) \bmod 62] \oplus 3$, where z is a 62-bit constant (determined by T) and $z[j]$ is the j-th bit of z. Finally, this algorithm returns key $K = (k_0||\ldots||k_{T-1})$.

- *Encryption*: We denote this algorithm by $\mathsf{SIMON}_{\mathrm{enc}}(K, m)$ which takes as input a key $K = (k_0||\ldots||k_{T-1})$ and a $2l_e$-bit message $m = (m_0||m_1)$ (that is divided into two equal length sub-messages), and outputs a $2l_e$-bit block ciphertext $C = (c_0||c_1)$. Each round of encryption encompasses a two-stage Feistel map $\mathsf{RF}_k(x, y) = (y \oplus f(x) \oplus k, x)$ where $f(x) = (x \lll 1)\ \&\ (x \lll 8)) \oplus (x \lll 2)$.
- *Decryption*: We denote this algorithm by $\mathsf{SIMON}_{\mathrm{dec}}(K', C)$ which takes as input the encryption key $K' = (k_{T-1}||\ldots||k_0)$ and a $2l_e$-bit ciphertext $C = (c_0||c_1)$, and outputs message $m = (m_0||m_1)$. This algorithm leverage a round function $\mathsf{RF}_k^{-1}(x, y) = (y, x \oplus f(y) \oplus k)$ for decryption.

A.3 Collision-resistant Hash Functions PHOTON and SPONGENT

In this section, we briefly describe two collision-resistant hash function families, PHOTON and SPONGENT, that standardized by ISO/IEC [JG16].

A.3.1 PHOTON

Here we review the two algorithms of PHOTON:

- *Initialization*: This algorithm initializes the constants and parameters that shall be used by during the hash evaluation (including the ones used by π_p). If first sets the initial vector IV and the first internal state S_0 as $S_0 = \mathrm{IV} := 0^{l_t-24}||\frac{l_h}{4}||l_r||l_{r'}$. It generates T round constants $\{\mathrm{RC}[i]\}_{i\in[T]}$ by a 4-bit linear feedback shift register with maximum cycle length, where each $\mathrm{RC}[i]$ has l_s-bit. And it initializes l_d internal constants $\{\mathrm{IC}[i]\}_{i\in[l_s]}$ by shift registers with a cycle length of l_d, where each $\mathrm{IC}[i]$ has l_s-bit. Besides, it also initializes a $l_d \times l_d$ matrix $\bar{\mathsf{B}}$ satisfying that $\bar{\mathsf{B}}^{l_d}$ is a Maximum Distance Separable (MDS) matrix [DR02]. More details about those constants can be found in [GPP11, Appendix D] and [GPP11, Table 1].
- *Evaluation*: We denote this algorithm by $\mathsf{PHOTON}(m)$ which takes as input a message $m \in \{0,1\}^{l_m}$ and outputs a hash value $z \in \{0,1\}^{l_h}$. m is first divided into ℓ blocks $m_0 \ldots m_{\ell-1}$ of l_r bits each, where ℓ is determined by $|m|$. The last block may be padded with a "1" bit along with many zeros (if necessary). The evaluation executes *absorbing* and *squeezing* procedures. In the absorbing phase, for $i \in [\ell]$, it computes the $i+1$-th internal state as $S_{i+1} := \pi_p(S_i \oplus (m_i||0^{l_c}))$. Once all ℓ message blocks have been absorbed, it computes $S_{\ell+i+1} := \pi_p(S_{\ell+i})$ for $i \in [\ell']$, in the squeezing phase. Eventually, hash output is built by concatenating the successive $l_{r'}$-bit output blocks $z_0, \ldots, z_{\ell'-1}$ until it gets appropriate output size l_h, where z_i is the $l_{r'}$ left-most bits of internal state $S_{\ell+i}$.

Now we briefly review the permutation π_p. It would first divide the l_t-bit input into a $l_d \times l_d$-matrix $\bar{\mathsf{C}}$, where each cell of such a matrix has l_s-bit. That is, we have $l_t = l_d \cdot l_d \cdot l_s$. In the following, we uses $\bar{\mathsf{C}}[i, j]$ (for $i, j \in [l_d]$) to access the cell at the i-th row and j-th column.

The π_p has $T = 12$ rounds, and each round would run four procedures: *AddConstant*, *SubCell*, *ShiftRows*, and *MixColumnsSerial*. AddConstant computes $\bar{\mathsf{C}}[i, 0] := \bar{\mathsf{C}}[i, 0] \oplus \mathrm{RC}[v] \oplus \mathrm{IC}[i]$. SubCell applies the l_s-bit SBOX to substitute every cell of $\bar{\mathsf{C}}$. For $i, j \in [l_d]$, it computes $\bar{\mathsf{C}}[i, j] := \mathsf{SBOX}(\bar{\mathsf{C}}[i, j])$. If $l_s = 4$, then it uses PRESENT's SBOX [BKL+07], otherwise it uses the AES's SBOX [DR02]. For $j \in [l_d]$, ShiftRows computes for $i \in [1, \ldots, l_d - 1]$. In MixColumnsSerial, it computes $(\bar{\mathsf{C}}[0, j], \ldots, \bar{\mathsf{C}}[l_d - 1, j])^T = \bar{\mathsf{B}}^{l_d} \times (\bar{\mathsf{C}}'[0, j], \ldots, \bar{\mathsf{C}}'[l_d - 1, j])^T$ for $j \in [l_d]$, where the superscript T denote the transformation of a matrix, and $(\bar{\mathsf{C}}[0, j], \ldots, \bar{\mathsf{C}}[l_d - 1, j])^T$ is all cells in the j-th column of matrix $\bar{\mathsf{C}}$. The final result of π_p is updated matrix $\bar{\mathsf{C}}$.

A.3.2 SPONGENT

We review the SPONGENT construction via the following algorithms:

- *Initialization*: This algorithm randomly generates a 8-bit SBOX as in the state-of-the-art (C++) reference implementation of SPONGENT [BKL+12]. It also initializes the parameters including the rate l_r of bits of input or output handled in the permutation procedure, the capacity l_c of bits of internal state, and the hash output length l_h in bits. These parameters can uniquely determined a specific version of SPONGENT.
- *Evaluation*: We denote this algorithm by $\mathsf{SPONGENT}(m)$ which takes as input a message $m \in \{0,1\}^{l_m}$ and outputs a hash value $z \in \{0,1\}^{l_h}$. This algorithm first pads input message m with the same approach of PHOTON, and divides m into l_r-bit blocks $(m_1, m_2, \ldots, m_\ell)$, where $\ell = \frac{|m|}{l_r}$. $\mathsf{SPONGENT}(m)$ encompasses two phases: *Absorbing* and *Squeezing* . In Absorbing phase, each l_r-bit message block is xored into the first l_r bits of the state and each resultant state is updated by the permutation function π_s. Squeezing phase is used to generate the hash value, which would iteratively get the first l_r bits of the state and apply the permutation function π_s to update the state, until l_h bits are obtained.

The core permutation function π_s operates over the state St with size $l_g = l_r + l_c$ in T-rounds, where $T \in \{45, 70, 90, 120, 140\}$ that corresponds to the parameters $l_h/l_c/l_r$. For $i \in [T]$, π_s runs three procedures: $\mathrm{St} := \mathsf{VI}(i) \oplus \mathrm{St} \oplus \mathsf{IV}(i)$, SLayer, and PLayer, to update the state. $\mathsf{IV}(i)$ is the state of an linear feedback shift register (LFSR) in round i, which outputs the round dependent constant and is xored to the right-most bits of state. $\mathsf{VI}(i)$ is the bits of $\mathsf{IV}(i)$ represented in reversed order, which is xored to the left-most bits of state. The initial values of $\mathsf{IV}(i)$ can be found in [BKL+13, Table 2]. The SLayer procedure is identical to that of PRESENT. PLayer moves the j-th bit of St to new position $\mathsf{sPBOX}(j)$, where $\mathsf{PBOX}(j)$ returns $j \cdot l_g/4 \bmod l_g$ if $j \in [l_g - 1]$ and $l_g - 1$ otherwise.

A.4 Proof of Aliveness

A proof of aliveness (PoA) scheme is a two-party protocol in which a client idC proves its aliveness at a certain time to a server idS. We represent the time elapse via a discrete-time slots $\{T_i\}$, and any two of them has a time interval Δ_s, i.e., $T_{i+1} - T_i = \Delta_s$. Let Δ_{rc} denote the life-span of a PoA protocol instance, and T_{att} be the aliveness tolerance time. Basically, if the server idS fails to receive any valid proof from the client idC within T_{att}, then idC is considered to be dead.

Here we review the second protocol $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ proposed in [JYvDZ19] and implemented on PLC. We briefly review the protocol execution phases of $\Pi^{\mathsf{PRG}}_{\mathsf{OWF}}$ as below:

- *Initialization*: The life-span Δ_{rc} of the protocol is first divided into η time periods, and each period has a length $\Delta'_{rc} = \lfloor \frac{\Delta_{rc}}{\eta} \rfloor$. In the i-th ($i \in [\eta]$) time period, idC runs $ss^i || p^i_0 := \mathsf{PRG}(ss^{i-1})$ to get the corresponding initial secret p^i_0 of the i-th sub-chain and the PRG seed ss^i, It generates the i-th verify-point p^i_N (for verifying the aliveness proofs in the corresponding chain) by computing N-times of the OWF F. Namely, each node p^i_j in this chain is computed from its predecessor p^i_{j-1} and the OWF F, i.e., $p^i_j := \mathsf{F}(p^i_{j-1})$. At the end, the client would keep the state $(ss^u, p^1_0, \{T^i_{end}\}_{0 \le i \le \eta}, \eta, u)$, where u is a variable initialized to be 1 to track the index of the stored PRG seed and the OWF-chain head. And the server has state $(\{p^i_N, T^i_{end}\}_{i \in [\eta]}, T_{ack}, \eta)$, where T_{ack} is the latest time that the verifier received a valid aliveness proof.
- *Proof Generation*: To get a proof for time T, the client idC should first figure out the index i of the OWF-chain that should be used, which is determined by the current time. Then, idC gets the i-th OWF-chain head based on the current stored PRG seed ss^i by literally running $u := u + 1$ and $ss^u || p^u_0 := \mathsf{G.Gen}(ss^{u-1})$ until $u = i$. Then, idC gets the number M of OWF that needs to be computed as $M := \frac{T^u_{end} - T}{\Delta_s}$. For $i \in [M]$, idC runs

$p_i := \mathsf{F}(p_{i-1})$. After this, idC sets $p_T := p_M$ which is returned as the aliveness proof for time T.

- *Proof Verification:* Upon receiving a proof $p = p_T$, the server idS verifies it by checking whether $T - T_{ack} < T_{att}$, and the current verify-point p_j^u can be computed from p_T. That is, idS first obtains $Z := \frac{T-T_{ver}}{\Delta_s}$ (i.e., the number of OWF evaluations for verification), and sets $p'_0 := p_T$. For $i \in [Z]$, idS computes $p'_i := \mathsf{F}(p'_{i-1})$. If $p_Z = p_j^u$ then p_T is valid. If $T - T_{ack} > T_{att}$ and the p_T is valid, idS sets the verify-point by setting $p_j^u := p_T$ and $T_{ack} := T$.

Proof Replenishment. Another necessity of PoA is the proof replenishment feature, which is used to reinitialize the protocol instance when the proof is about to use out. In [JYvDZ19], Jin et al. proposed to use a one-time signature scheme (OTS) to achieve this feature, in which the signing key of OTS is the OWF-chain heads of the sub-chains. To replenish the proofs, the prover can initialize a new protocol instance to get the state $(\{p_N^{i,new}, T_{end}^{i,new}\}_{i\in[\eta]}, T_{ack}^{new}, \eta)$, and then sign it based on the OWF-heads of the last protocol instance. The OTS instance used in [JYvDZ19] is the first OTS introduced by Lamport [Lam79], which is based on OWF. The signing key of Lamport OTS consists of 256 OWF keys, and signing procedure is to select 128 OWF keys from the signing key according to the bits of the signed message.

A.5 Big-integer Operations

Most of the modern cryptographic algorithms rely on big-integer operations (over 128 bits). But they are not directly supported by PLCs. In this work, we are going to benchmark the standard big-integer operations, such as multiple-precision addition, subtraction, multiplication, and division (modulo), and binary multiplication in an additive group, that are runnable on PLC (i.e., it can be finished within a scan cycle of PLC)[6]. Here we leverage on the most common radix representation approach to represent a big-integer. That is, the representation of a positive integer a is represented as a sum of multiples of powers of a base b, i.e., $a = a_{l_a-1}b^{l_a-1} + a_{l_a-2}b^{l_a-2} + \ldots + a_1 b + a_0$, where the integers a_i (for $i \in [l_a]$) are called digits. And $b = 2^{l_b}$. In the following, we review the classic big-integer algorithms (as reference) introduced in [MvOV96].

- *Multiple-precision Addition* [MvOV96, Algorithm 14.7]: We denote this algorithm by $\mathsf{Add}(x, y)$ which takes as input positive integers x and y which are represented as l_a digits, and outputs the sum $w = x + y = (w_{l_a} \ldots w_0)_b$. This algorithm first initializes carry $c := 0$. For $i \in [l_a]$, it computes the result of each digit as $w_i := (x_i + y_i + c) \bmod b$; if $w_i > b$, it sets $c := 1$, and $c := 0$ otherwise. The digit w_{l_a} is assigned to be the last carry bit.
- *Multiple-precision Subtraction* [MvOV96, Algorithm 14.9]: We denote this algorithm by $\mathsf{Sub}(x, y)$ which takes as input positive integers x and y represented in l_a-digit, and outputs the difference $w = x - y = (w_{l_a-1} \ldots w_0)_b$. It first initializes carry $c := 0$. Next, for $i \in [l_a]$, it computes $w_i := (x_i - y_i + c) \mod b$; If $(x_i - y_i + c) \geq 0$, then it sets $c := 0$, and $c := -1$ otherwise.
- *Multiple-precision Multiplication* [MvOV96, Algorithm 14.12]: We denote this algorithm by $\mathsf{Mul}(x, y)$ which takes as input l_a-digit multiplier x and l_y-digit multiplicand y, and outputs product $w = x \cdot y = (w_{l_a+l_y-1} \ldots w_0)_b$. First, it initializes $w := 0$. For $i \in [l_y]$, it runs $c := 0$, then for $j \in [l_a]$, it computes $(uv)_b := w_{i+j} + x_j y_i + c$, sets $w_{i+j} := v$ and $c := u$, where u and v are base b digits and u may be 0. Then, it sets $w_{i+l_a} := u$.
- *Multiple-precision Division* [MvOV96, Algorithm 14.20]: We denote this algorithm by $\mathsf{Div}(x, y)$ which takes as input positive integer l_a-digit dividend x and divisor l_y-digit y, and outputs $(l_a - l_y + 1)$-digit quotient $q = x/y = (q_{l_a-l_y} \ldots q_0)_b$ and l_y-digit remainder

[6] Our benchmark results imply that some other expensive operations (like modular exponentiation) cannot be done within a scan cycle of PLC.

$r = (r_{l_y-1} \dots r_0)_b$, where $l_a \geq l_y \geq 1$, and $y \neq 0$. The algorithm does the following steps:

- While $x \geq yb^{l_a-l_y}$, it computes $q_{l_a-l_y} := q_{l_a-l_y} + 1$ and $x := x - yb^{l_a-l_y}$;
- For $i \in [l_a - 1, \dots, l_y]$, it runs the following steps: If $x_i = y_{l_y-1}$, then it sets $q_{i-l_y} := b - 1$, else it sets $q_{i-l_y} := \lfloor (x_i b + x_{i-1})/y_{l_y-1} \rfloor$; While $q_{i-l_y}(y_{l_y-1}b + y_{l_y-2}) > x_i b^2 + x_{i-1}b + x_{i-2}$, it computes $q_{i-l_y} := q_{i-l_y} - 1$; $x := x - q_{i-l_y} y b^{i-l_y}$; If $x < 0$ then it sets $x := x + yb^{i-l_y}$ and $q_{i-l_y} := q_{i-l_y} - 1$;
- Eventually, the quotient q has been calculated, and the reminder is $r := x$.

Note that the division operation can also be used as a modular operation, in which case only the reminder r is returned.

- *Multiplication in an Additive Group* [MvOV96, Algorithm 14.92]: We denote this algorithm by $\mathsf{AMul}(e, g)$ which takes as input two l_a-digit integers e and g in additive group $\mathbb{Z}_m$, and outputs the product in $\mathbb{Z}_m$, i.e., $w = e \cdot g \pmod m = (w_{l_a+l_y-1} \dots w_0)_b$. AMul first initializes the product $w := 0$. For $i \in [l_a - 1, \dots, 0]$: it computes $w = \mathsf{Add}(w, w)$, and $w = \mathsf{Add}(w, g)$ if the i-th bit of e is 1.

B Remarks on Extending the Life-span of PoA Instances

To improve password replenishment, it is possible to leverage external storage, such as an SD card that is supported by PLC [Bra16], so that we could compute each new tail node at the idle time of PLC and store it at the SD card. The time to compute a tail node is identical to the worst-case time to generate a password in the proof generation procedure. As we assume that the PLC is running in RUN mode, these tail nodes cannot be tampered by adversaries. The compression of tail nodes based on UOWHF could also be done at the idle time as well, which costs about 1s. For the online OTS signing procedure, it only needs to read tails and the corresponding hash value from SD, and run the PRG to generate the OWF-chain heads accordingly, that roughly costs 6s. In this way, the PoA instance could use a much longer sub-chain, so that the interval between two replenishment procedures can be longer as well. For example, an POA instance can be used for 91 days when $N = 1024$. To facilitate the proof generation algorithm, one could also store checkpoints in sub-chains in the SD card. Nevertheless, it is an open question to figure out the optimal implementation strategy for PoA on PLC with external storage devices.

C Other Pseudo-codes

C.1 Pseudo-codes of Chaskey

The MAC evaluation function $\mathsf{Chaskey}(m)$ is shown by Algorithm 5. The input message $m \in \{0,1\}^{l_m}$ is represented as a DINT ARRAY $m[\ell * \mathrm{DN}_k]$, where $\ell * \mathrm{DN}_k = \frac{128}{32}$ for 128-bit security. We define a DINT ARRAY $K[3\mathrm{DN}_k]$ to store the key $K = k||K_1||K_2$. To hard-code the key, the first step of $\mathsf{Chaskey}$ is to assign $\{K[i]\}_{i \in [3\mathrm{DN}_k]}$ with corresponding concrete value of $k||K_1||K_2$ generated by Python.

We implement the permutation π_c (with $T \in \{8, 16\}$ rounds) on PLC via the Algorithm 5.

C.2 Pseudo-code of SPECK

The encryption function $\mathsf{SPECK}_{\mathrm{enc}}(m)$ of SPECK would first divide the input message into two words each of which has length l_e. In the implementation, we consider a word length $l_e \in \{16, 32, 64\}$. $\mathsf{SPECK}_{\mathrm{enc}}(m)$ is realized by Algorithm 6.

Algorithm 4: Evaluation of Chaskey

Input: DINT ARRAY $m[\ell * \mathrm{DN}_k]$ where $\mathrm{DN}_k = 4$ (for 128-bit).
Output: DINT ARRAY $\tau[\mathrm{DN}_k]$.

1: Assign hard-coded keys $k||K_1||K_2$ to $\{K[i]\}_{i\in[3\mathrm{DN}_k]}$;
2: $\{\mathrm{ST}[i]\}_{i\in[\mathrm{DN}_k]} := \{k[i]\}_{i\in[\mathrm{DN}_k]}$; // Init ST
// Absorb message blocks and permute them
3: **for** $j := 0$ to $\ell - 1$ by 1 **do**
4: Absorb the j-th message block $m[j * \mathrm{DN}_k + i]$ into $\mathrm{ST}[i]$ for $i \in [\mathrm{DN}_k]$, i.e., $\mathrm{ST}[i] := \mathrm{ST}[i]$ XOR $m[j * \mathrm{DN}_k + i]$;
5: $\{\mathrm{ST}[i]\}_{i\in[\mathrm{DN}_k]} := \pi_c(\{\mathrm{ST}[i]\}_{i\in[\mathrm{DN}_k]})$;
6: **end for**
7: If the last message block is not 128-bit, pad the last message block, and set $\{L[\mathrm{DN}_k]\}_{i\in[\mathrm{DN}_k]} := K_2[\mathrm{DN}_k]_{i\in[\mathrm{DN}_k]}$; Otherwise $\{L[\mathrm{DN}_k]\}_{i\in[\mathrm{DN}_k]} := K_1[\mathrm{DN}_k]_{i\in[\mathrm{DN}_k]}$;
8: Absorb the last message block into $\{\mathrm{ST}[i]\}_{i\in[\mathrm{DN}_k]}$ as above;
//Squeeze the state to generate the MAC tags
9: $\{\mathrm{ST}[i]\}_{i\in[\mathrm{DN}_k]} := \pi_c(\{\mathrm{ST}[i]\}_{i\in[\mathrm{DN}_k]})$;
10: $\tau[i] := \mathrm{ST}[i]$ XOR $L[i]$ for $i \in [\mathrm{DN}_k]$;
Clear $\{K[i]\}_{i\in[3\mathrm{DN}_k]}$;
11: **return** $\{\tau[i]\}_{i\in[\mathrm{DN}_k]}$;

Algorithm 5: Permutation π_c of Chaskey

Input: DINT ARRAY $v[4]$, internal state.
Output: DINT ARRAY $v[4]$.

1: **for** $i := 0$ to $T - 1$ by 1 **do**
2: $v'[0] := ((v[1]\hat{+}v[2])\lll 16)\hat{+}(v[3]\lll 8)$ XOR $(v[2]\hat{+}v[3])$;
3: $v'[1] := (((v[1]\lll 5)$ XOR $(v[1]\hat{+}v[0]))\lll 7)$ XOR $((v[3]\hat{+}v[2])\hat{+}((v[1]\lll 5)$ XOR $(v[1]\hat{+}v[0])))$;
4: $v'[2] := (((v[3]\hat{+}v[2])\hat{+}((v[1]\lll 5)$ XOR $(v[1]\hat{+}v[0])))\lll 16)$;
5: $v'[3] := (((v[1]\hat{+}v[0])\lll 16)\hat{+}(v[3]\lll 8))$ XOR $(v[3]\hat{+}v[2])$ XOR $(v[3]\lll 8)$ XOR $(v[3]\hat{+}v[2])\lll 13)$;
6: $\{v[i]\}_{i\in[4]} := \{v'[i]\}_{i\in[4]}$;
7: **end for**
8: **return** $\{v[i]\}_{i\in[4]}$;

Algorithm 6: Encryption of SPECK

Input: DINT ARRAY $m[2]$, each storing l_e-bit message.
Output: DINT ARRAY $c[2]$ storing ciphertext.

Assign the hard-coded $k_0||\ldots||k_{T-1}$ to KEY array $\{K[i]\}_{i\in[T]}$;
$c[0] := m[0]$; $c[1] := m[1]$;
$\alpha := 8$; $\beta := 3$;
if $l_e = 16$ **then**
    $\alpha := 7$; $\beta := 2$;
end if
for $i := 0$ to $T-1$ by 1 **do**
    $c[1] := ((c[1] \hat{\gg} \alpha) \hat{+} c[0])$ XOR $K[i]$;
    $c[0] := (c[0] \hat{\ll} \beta)$ XOR $c[1]$;
end for
Clear $\{K[i]\}_{i\in[T]}$;
return $\{c[i]\}_{i\in[2]}$;

C.3 Pseudo-code of SIMON

The implementation of SIMON is similar to that of SPECK, which mainly relies on our hard-coded shift/rotate function. Algorithm 7 shows the pseudo-code of $\mathsf{SIMON}_{\mathrm{enc}}(m)$.

Algorithm 7: Encryption of SIMON

Input: DINT ARRAY $m[2]$, each storing l_e-bit message.
Output: DINT ARRAY $c[2]$.

Assign the hard-coded $k_0||\ldots||k_{T-1}$ to KEY array $\{K[i]\}_{i\in[T]}$;
$c[0] := m[1]$; $c[1] := m[0]$;
for $i := 0$ to $T-1$ by 1 **do**
    $tmp := c_0$;
    $c[0] := c[1]$ XOR $((c[0] \hat{\ll} 1)$ AND $(c[0] \hat{\ll} 8))$
        XOR $(c[0] \hat{\ll} 2)$ XOR $K[i]$;
    $c[1] := tmp$;
end for
Clear $\{K[i]\}_{i\in[T]}$;
return $\{c[i]\}_{i\in[2]}$;

C.4 Pseudo-code of PRF and PRG

Algorithm 8 shows the evaluation function of PRF/PRG on PLC. One could adjust the output length through the parameter l_r to obtain the specific functionality of either PRF or PRG. Concerning PRF, we assume that the input message x could be any value, and the output is just two ciphertext values, i.e., $\mathrm{DN}_r = 1$. As for PRG, we assume that the input message x should be counter values in which the first message $x[0]$ is used to differentiate two PRG evaluations. We shall control the second one $x[1]$ incrementally to generate the long enough random numbers (determined by l_r). As the implementation of other algorithms, we do the initialization of both kinds of schemes on PC with Python, i.e., sampling random keys for them. Let $\mathsf{Enc} \in \{\mathsf{PRESENT}_{\mathrm{enc}}, \mathsf{SPECK}_{\mathrm{enc}}\}$ be one of the encryption schemes.

Algorithm 8: Evaluation of PRF and PRG

Input: DINT ARRAY $x[2]$.
Output: DINT ARRAY $R[\mathrm{DN}_r]$, where $\mathrm{DN}_r = \frac{l_r}{2|\mathsf{CIPHERTEXT}|}$.
1: Assign round encryption keys $k_0||\ldots||k_{T-1}$ to KEY array $\{K[i]\}_{i\in[T]}$;
2: **for** $i := 0$ to $\mathrm{DN}_r - 1$ by 2 **do**
3: $\quad \{R[i+j]\}_{j\in[2]} := \mathsf{Enc}(\{x[j]\}_{j\in[2]})$;
4: $\quad x[1] := x[1] + 1$;
5: **end for**
6: Clear $\{K[i]\}_{i\in[T]}$;
7: **return** $\{R[i]\}_{i\in[\mathrm{DN}_r]}$;

C.5 Pseudo-codes of PHOTON

PHOTON mainly leverages on a sponge-like construction (adopted by AES) as internal unkeyed permutation denoted by π_p which deals with $l_t = l_c + l_r$-bit input and has $T = 12$ rounds, where l_c is the capacity (security parameter) and l_r is the bit-length (bitrate) of a message block. The input message $m \in \{0,1\}^{l_m}$ of PHOTON will be divided into ℓ message blocks for hash evaluation, each of which has l_r bits, i.e., $l_m = \ell l_r$. The output hash value $z \in \{0,1\}^{l_h}$ of PHOTON is represented by ℓ' chunks each of which has a bit-length (bitrate) $l_{r'}$, where $l_h \in \{80, 128, 160, 224, 256\}$. π_p would apply a l_s-bit SBOX, where $l_s = 8$ when $l_h = 256$ and $l_s = 4$ otherwise. We may append the concrete value of l_h to PHOTON to differentiate each version.

The evaluation algorithm of PHOTON on PLC is shown by Algorithm 9, which mainly runs the permutation algorithm π_p to generate the hash value. We present π_p in Algorithm 10. To get the least significant l_s-bit from the SCShRMCS table look-up result Tv (≥ 32 bits) to update the internal l_s-bit state), we apply the optimization idea **B-RW** to develop a function $\mathsf{GetByte}(Tv, \mathrm{start}, l_s)$ that can get each l_s-bit block value from Tv, where start indicates the starting bit to operate. That is, the returned value res of $\mathsf{GetByte}(Tv, \mathrm{start}, l_s)$ is computed by a few assignments and additions $res.[i] := Tv.[\mathrm{start} + i]$ for $i \in [l_s]$. All other steps are realized following the specification.

C.6 Pseudo-codes of SPONGENT

The main skeleton of evaluation algorithm is briefly presented in Algorithm 11.

To implement the permutation function π_s, we first pre-compute round-dependent LSFR constants $\mathsf{VI}[i]$ and $\mathsf{IV}[i]$ for $i \in [T]$, unlike the reference implementation of SPONGENT [BKL+12] which, for example, computes $\mathsf{IV}[i]$ and $\mathsf{VI}[i]$ (for 88/178/88 version) on the fly as

$$\mathsf{IV}[i] = ((\mathsf{IV}[i-1] << 1)|(((0x80\&\mathsf{IV}[i-1]) >> 7) \oplus ((0x08\&\mathsf{IV}[i-1]) >> 3) \oplus ((0x04\&\mathsf{IV}[i-1]) >> 2) \oplus ((0x02\&\mathsf{IV}[i-1]) >> 1)))\&0x\mathrm{FF};$$

and

$$\mathsf{VI}[i] = (((\mathsf{IV}[i]\&0x01) << 7)|((\mathsf{IV}[i]\&0x02) << 5)|((\mathsf{IV}[i]\&0x04) << 3)|((\mathsf{IV}[i]\&0x08) << 1)| ((\mathsf{IV}[i]\&0x10) >> 1)|((\mathsf{IV}[i]\&0x20) >> 3)|((\mathsf{IV}[i]\&0x40) >> 5)|((\mathsf{IV}[i]\&0x80) >> 7)).$$

That is, we try to avoid such an expensive computation (without hardware support) during evaluation, so that we load them in the initialization phase as constants. The SBOX is loaded as constants as well, so we can realize the SLayer by lookup operations. In addition, the PLayer is implemented the similar idea in the implementation of PRESENT.

Algorithm 9: Evaluation of PHOTON

Input: DINT ARRAY $m[\mathrm{DN}_r]$ where $\mathrm{DN}_r = \frac{l_m}{8}$.
Output: DINT ARRAY $z[\mathrm{DN}_r{}']$ where $\mathrm{DN}_r{}' = \frac{l_h}{8}$.

Load the hard-coded parameters $\{\mathrm{RC}[i,j]\}_{i\in[l_d],j\in[T]}$, $\{\mathsf{PHTable}[i,j,t]\}\}_{i\in[l_d],j\in[2^{l_s}],t\in[2]}$, initial state $\{\bar{\mathsf{C}}[i,j]\}_{i\in[l_d],j\in[l_d]}$;
//Compress the first $l_m - l_r$-bit message
MBitIdx := 0;
while MBitIdx <= $l_m - l_r$ **do**
  Absorb each message block by xoring it into matrix $\bar{\mathsf{C}}$.
  $\{\bar{\mathsf{C}}[i,j]\}_{i\in[l_d],j\in[l_d]} := \pi_p(\{\bar{\mathsf{C}}[i,j]\}_{i\in[l_d],j\in[l_d]})$;
  MBitIdx := MBitIdx + l_r;
end while
Pad the last padded message block, and absorb it as before;
//Squeeze data blocks
hashbitlen := 0;
while hashbitlen <= l_h **do**
  Append $\frac{l_{r'}}{8}$ state blocks into digest z;
  $\bar{\mathsf{C}}[i,j]\}_{i\in[l_d],j\in[l_d]} := \pi_p(\{\bar{\mathsf{C}}[i,j]\}_{i\in[l_d],j\in[l_d]})$;
  hashbitlen := hashbitlen + $l_{r'}$;
end while
return $\{z[j]\}_{j\in[\mathrm{DN}_r{}']}$;

Algorithm 10: Permutation π_p of PHOTON

Input: DINT ARRAY $\bar{\mathsf{C}}[l_d, l_d]$, internal state matrix.
Output: DINT ARRAY $\bar{\mathsf{C}}[l_d, l_d]$.

for $v := 0$ to $T-1$ by 1 **do**
  AddConstant: add round-dependent constants to each cell of the first column of $\bar{\mathsf{C}}$, i.e., $\bar{\mathsf{C}}[i,0] := \bar{\mathsf{C}}[i,0]$ XOR RC[v] XOR IC[i];
  // *SubCell*, *ShiftRows*, and *MixColumnsSerial* based on PHTable:
  Duplicate $\bar{\mathsf{C}}$ to yield a copy $\bar{\mathsf{C}}'$
  for $j := 0$ to $l_d - 1$ by 1 **do**
    $Tv := 0$; //Init the SCShRMCS table lookup result
    for $i := 0$ to $l_d - 1$ by 1 **do**
      $u := \bar{\mathsf{C}}'[i, (i+j) \text{ MOD } l_d]$; //Table look-up index
      $Tv := Tv$ XOR $\mathsf{PHTable}[i,u]$;
      Start := 0;
      //Update state with right-most $l_d \times l_s$ bits of Tv
      for $\delta := 1$ to l_d by 1 **do**
        $\bar{\mathsf{C}}[l_d - \delta, j] := \mathsf{GetByte}(Tv, \text{Start}, l_s)$;
        Start := Start + l_s;
      end for
    end for
  end for
end for
return $\{\bar{\mathsf{C}}[i,j]\}_{i\in[l_d],j\in[l_d]}$;

Algorithm 11: Evaluation of SPONGENT

Input: DINT ARRAY $m[\mathrm{DN}_r]$ where $\mathrm{DN}_r = \frac{l_m}{8}$.
Output: DINT ARRAY $z[\mathrm{DN}_r{}']$ where $\mathrm{DN}_r{}' = \frac{l_m}{8}$.

Load the hard-coded parameters, incl. $\{\mathsf{IV}[i]\}_{i\in[T]}$, $\{\mathsf{VI}[i]\}_{i\in[T]}$, and $\{\mathsf{SBOX}[i]\}\}_{i\in[\mathrm{NS}}$, where $\mathrm{NS} = \frac{l_c+l_r}{8}$;
//Absorb available message blocks
while databitlen >= l_r **do**
  Absorb each l_r-bit message block into the state St.
  $\{\mathrm{St}[i]\}_{i\in[\mathrm{NS}]} := \pi(\{\mathrm{St}[i]\}_{i\in[\mathrm{NS}]})$;
  databitlen := databitlen-l_r;
end while
Pad the last message block, and absorb it as before;
//Squeeze data blocks
while hashbitlen < l_h **do**
  Append the first l_r-bit of St into the digest z.
  $\{\mathrm{St}[i]\}_{i\in[\mathrm{NS}]} := \pi(\{\mathrm{St}[i]\}_{i\in[\mathrm{NS}]})$;
  hashbitlen:=hashbitlen+l_r;
end while
return $\{z[j]\}_{j\in[\mathrm{DN}_r{}']}$;

Algorithm 12: Permutation π_s of SPONGENT

Input: DINT ARRAY St[NS], where $\mathrm{NS} = \frac{l_c+l_r}{8}$.
Output: DINT ARRAY St[NS].

for $j := 0$ to $T-1$ by 1 **do**
  St[0] := St[0] XOR $\mathsf{IV}[j]$ AND 16#FF;
  St[1] := St[0] XOR (($\mathsf{IV}[j]$/258) AND 16#FF);
  St[NS − 2] := St[NS − 2] XOR $\mathsf{VI}[j]$ AND 16#FF;
  St[NS − 1] := St[NS − 1] XOR (($\mathsf{VI}[j]$/258) AND 16#FF);
  //SLayer
  for $i := 0$ to NS − 1 by 1 **do**
    St[i] := SBOX(St[i]);
  end for
  //PLayer with hard-coded assignments
  ptmp[0].[0] := St[0].[0];
    ⋮
  ptmp[6].[5] := St[4].[6];
    ⋮
  ptmp[NS].[7] := St[NS].[7];
  Copy the PLayer result *ptmp* to St;
end for
return $\{\mathrm{St}[i]\}_{i\in[\mathrm{NS}]}$;

C.7 Pseudo-codes of PoA

We implement the proof generation of PoA by Algorithm 13 and the replenishment algorithm is shown by Algorithm 14.

Algorithm 13: Proof Generation of PoA

Input: DINT Counters P, S and Idx, and DINT ARRAY $\tau[\mathrm{DN}_k]$ storing the MAC of tags being protected, where Idx is an index of the next proof to be computed.

Output: DINT ARRAY $pw[\mathrm{DN}_x]$ storing the proof to be sent, DINT P, S and Idx, and DINT ARRAY $\tau[\mathrm{DN}_k]$

1: Check if $\{\tau[i]\}_{i\in[\mathrm{DN}_k]} = \mathsf{Chaskey}(\mathrm{P}||\mathrm{S}||\mathrm{Idx})$;
2: $\{pw[i]\}_{i\in[\mathrm{DN}_x]} := \mathsf{PRG}(\mathrm{P}||\mathrm{S}+l_r)$; //compute p_0^{S}
3: **if** Idx=0 **then**
4: Idx := N; // The value of N is hard-coded
5: S := S + 2^{23}; // S is stored at leftmost 9-bit of DINT
6: **if** S=M **then**
7: P := P + 1; // switch to new protocol instance
8: **end if**
9: **end if**
10: **for** $i := 0$ to Idx − 1 by 1 **do**
11: $\{pw[i]\}_{i\in[\mathrm{DN}_x]} := \mathsf{F}_{\mathrm{sss}}(\{pw[i]\}_{i\in[\mathrm{DN}_x]})$; //proof gen.
12: **end for**
13: Idx := Idx − 1; //update the index for next proof
14: $\{\tau[i]\}_{i\in[\mathrm{DN}_k]} := \mathsf{Chaskey}(\mathrm{P}||\mathrm{S}||\mathrm{Idx})$;
15: **return** $\{pw[i]\}_{i\in[\mathrm{DN}_x]}$, P, S, Idx and $\tau[\mathrm{DN}_k]$;

Algorithm 14: Proof Replenishment of PoA

Input: DINT counters P and S, and DINT ARRAY $\tau[\mathrm{DN}_k]$ storing the MAC of tags being protected.

Output: DINT ARRAY $\mathrm{Tail}[M, \mathrm{DN}_x]$ storing the new tail nodes, DINT P, S and Idx, and DINT ARRAY $\tau[\mathrm{DN}_k]$, and DINT ARRAY $\mathrm{SIG}[128, \mathrm{DN}_x]$ storing signature

1: Check if $\{\tau[i]\}_{i\in[\mathrm{DN}_k]} = \mathsf{Chaskey}(\mathrm{P}||\mathrm{S}||\mathrm{Idx})$;
2: P := P + 1; S := 0; Idx := 0;
//Compute the i-th tail node
3: **for** $i := 0$ to $M - 1$ by 1 **do**
4: $\{\mathrm{Tail}[i,\mu]\}_{\mu\in[\mathrm{DN}_x]} := \mathsf{PRG}(\mathrm{P}||i)$;
5: Repeat N times of $\{\mathrm{Tail}[i,\mu]\}_{\mu\in[\mathrm{DN}_x]} := \mathsf{F}_{\mathrm{sss}}(\mathrm{Tail}[i,\mu]\}_{\mu\in[\mathrm{DN}_x]})$ to generate the i-th tail node;
6: **end for**
//Lamport OTS, select 128 heads according to m
7: $\{m[\iota]\}_{\iota\in[128]} \leftarrow \mathsf{H}_{sss}(\mathrm{Tail}[i,j]\}_{i\in[256],j\in[\mathrm{DN}_x]})$; //m is 128-bit
8: **for** $i := 0$ to 127 by 1 **do**
9: $\mathrm{SIG}[i,j]\}_{j\in[\mathrm{DN}_x]} := \mathsf{PRG}(\mathrm{P}||(2i + m[i])2^{23} + l_r)$;
10: **end for**
11: $\{\tau[i]\}_{i\in[\mathrm{DN}_k]} := \mathsf{Chaskey}(\mathrm{P}||\mathrm{S}||\mathrm{Idx})$; //protect the tags
12: **return** $\{\mathrm{Tail}[i,j]\}_{i\in[256],j\in[\mathrm{DN}_x]}$, P, S, Idx, $\{\tau[i]\}_{i\in[\mathrm{DN}_k]}$, and $\mathrm{SIG}[i,j]\}_{i\in j\in[128],j\in[\mathrm{DN}_x]}$;

D Implementation and Performance of Big-integer Operations

D.1 Pseudo-codes of Big-integer Operations

Here we focus on describing the implementation of big-integer operations with a particular base $b = 2^{l_b}$ that can be optimized via ST's bit-wise operability. For other bases, e.g., $b = 10$, they can be implemented analogously following the specification. We represent each operand of big-integer operation with l_b-bit digits. The implementation of multiple-precision addition $\mathsf{Add}(x, y)$ is shown by Algorithm 15. Since the carry is only a one-bit value, we can obtain it from the sum of two digits for free, e.g., $c := w[i].[l_b]$, and the mod base operation can be straightforwardly realized by setting the l_b-th bit to be zero.

Algorithm 15: Big-integer Multiple-precision Addition

Input: Positive integers x and y are represented as DINT ARRAY $x[l_a]$ and $y[l_a]$, i.e., each operand has l_a digits with base $b = 2^{l_b}$.
Output: DINT ARRAY $w[l_a + 1]$ storing the sum $w = x + y = (w[l_a] \dots w[0])_b$.

1: $c := 0$; //Initialize carry c to zero
2: **for** $i := 0$ to $l_a - 1$ by 1 **do**
3: $w[i] := x[i] + y[i] + c$; // add each digit
4: $c := w[i].[l_b]$;
5: $w[i].[l_b] := 0$; // mod 2^{l_b}
6: **end for**
7: $w[l_a] := c$;
8: **return** $\{w[i]\}_{i \in [l_a+1]}$;

Algorithm 16 shows the implementation of multiple-precision subtraction $\mathsf{Sub}(x, y)$. Here an AND operation is used to realize the mod $b = 2^{l_b}$ operation, e.g., $w[i] := w[i]$ AND $0^{32-l_b}||1^{l_b-1}$, since PLC adopts two's complement to represent a negative number consisting of many ones that should be removed.

Algorithm 16: Big-integer Multiple-precision Subtraction

Input: Positive integers x and y are represented as DINT ARRAY $x[l_a]$ and $y[l_a]$, i.e., each operand has l_a base $b = 2^{l_b}$ digits.
Output: DINT ARRAY $w[l_a]$ storing the difference $w = x - y = (w[l_a - 1] \dots w[0])_b$.

1: $c := 0$; //Initialize carry c to zero
2: **for** $i := 0$ to $l_a - 1$ by 1 **do**
3: $w[i] := (x[i] - y[i] + c)$; // substract each digit
4: **if** $w[i].[31] = 1$ **then**
5: $c := -1$; // $w[i].[31]$ is the sign bit
6: **end if**
7: $w[i] := w[i]$ AND $0^{32-l_b}||1^{l_b-1}$; // mod 2^{l_b}
8: **end for**
9: **return** $\{w[i]\}_{i \in [l_a]}$;

The multiple-precision multiplication algorithm $\mathsf{Mul}(x, y)$ is implemented with Algorithm 17, which mainly leverages on the above new mod b operation approach and our PLC shift function introduced before.

The implementation of multiple-precision division $\mathsf{Div}(x, y)$ is shown by Algorithm 18. To compare two equal-length big-integers, we define a comparison function $\mathsf{Comp}(\{x[i]\}_{i \in [l_a]}, \{y[i]\}_{i \in [l_a]})$ that two takes as input big-integers' digit representation and outputs the

Algorithm 17: Big-integer Multiple-precision Multiplication

Input: Positive integers x and y are represented as DINT ARRAY $x[l_a]$ and $y[l_y]$, i.e., x has l_a base $b = 2^{l_b}$ digits and y has l_y base b digits.
Output: DINT ARRAY $w[l_a + l_y]$ storing the product $(w[l_a + l_y - 1] \dots w[0])_b$.

1: $\{w[i]\}_{i\in[l_a+l_y]} := 0$;
2: **for** $i := 0$ to $l_y - 1$ by 1 **do**
3: $\quad c := 0$;
4: $\quad$ **for** $j := 0$ to $l_a - 1$ by 1 **do**
5: $\quad\quad \hbar := w[i + j] + y[j] \cdot x[i] + c$; $\quad //\hbar = (uv)_b$
6: $\quad\quad w[i + j] := \hbar$ AND $0^{32-l_b}||1^{l_b-1}$; $\quad //w[i + j] := v$
7: $\quad\quad c := h/b$; $\quad //c := u$
8: $\quad$ **end for**
9: $\quad w[i + j] := c$;
10: **end for**
11: **return** $\{w[i]\}_{i\in[l_a+l_y]}$;

digital-wise comparison result $r \in \{0, 1\}$, where 1 indicates $\{x[i]\}_{i\in[l_a]} \geq \{y[i]\}_{i\in[l_a]}$ and 0 otherwise. And we say that $\{x[i]\}_{i\in[l_a]} \geq \{y[i]\}_{i\in[l_a]}$ if and only if $x[i] \geq y[i]$ for all $i \in [l_a]$.

During the division, we need to shift left a big-integer with a few digits, e.g., the result of yb^{i-l_y-1} (as illustrated in Section A.5). To so do, we define a function LShiftB that takes as input a big-integer $\{x[i]\}_{i\in[l_a]}$ and the number of digits $pos(< l_a)$ that it will shift, it returns the left-shifted result $\{y[i]\}_{i\in[l_a]}$. That is, $y[i] := 0$ for $i \in [pos]$, and $y[j] := x[j - pos]$ for $j \in [pos, l_a]$.

By Algorithm 19, we implement multiplication in a additive group $\mathbb{Z}_m$, i.e., AMul. Since scanning the bits of the input e is free, AMul can be realized by running our multiple-precision addition following the specification. In the meantime, if some intermediate result is larger than $2m$, we do the modular reduction of the value by subtracting it with m.

D.2 Performance of Big-integer Operations

Big-integer operations are implicitly used in our *PLCrypto*, while mathematical operations are carried with operands over 30 bits. To show the performance of big-integer operations on PLC, we implement five typical big-integer operations as presented in Appendix A.5 and Appendix D.1, i.e., multiple-precision addition (Add), subtraction (Sub), multiplication (Mul), and division (Div), and additive multiplication (AMul).

We benchmark the big-integer operations with three types of bases, i.e., $b = 2^{15}$, $b = 2^{30}$ and $b = 10$. However, base $b = 2^{30}$ is used for benchmarking the addition and the subtraction only, since 30-bit multiplication would exceed the range of DINT. Note that the big-integer addition/subtraction with base $b = 2^{30}$ is implicitly used in the implementation of SPECK. Note that modulo 2^{31} addition (used by subset-sum based OWF) is similar to that of modulo 2^{30}. We thus omit it here for simplicity. Also, the 32-th bit of a DINT variable is the sign bit that cannot be used to store the multiplication result, so the base $b = 2^{15}$ is the largest base and most efficient one for multiplication that we could test.

In Figure 9, we show the performance of five types of big-integer functions. The Add and Sub are not efficient enough comparing to the corresponding results on other platforms. So the Mul and the Div which are realized based on Add and Sub are costly. Since the group formed by the points on elliptic curve cryptography (ECC) over a finite field unitizes additive notation, our result regarding Algorithm 19 implies that ECC might be feasible on PLC. For example, AMul with 512-bit multipliers costs around 400 ms. Also, we have experimentally verified that the DINT multiplication is constant-time on our platform, regardless of the given operands. In this work, we just made some preliminary attempts

Algorithm 18: Big-integer Multiple-precision Division

Input: Positive integers x and y are represented as DINT ARRAY $x[l_a]$ and $y[l_y]$ ($l_a \geq l_y \geq 1$ and $y \neq 0$), i.e., x and has l_a base $b = 2^{l_b}$ digits, y has l_y base b digits.

Output: DINT ARRAY $q[l_a - l_y + 1]$ and $r[l_y]$, storing quotient $q = (q[l_a - l_y], \ldots, q[0])_b$ and reminder $r = (r[l_y - 1], \ldots, r[0])_b$.

1: $\{q[i]\}_{i\in[l_a-l_y+1]} := 0$;
2: $\{z[i]\}_{i\in[l_a]} := \mathsf{LShiftB}(\{y[i]\}_{i\in[l_y]}, l_y)$;
//Compute the highest digit of the quotient
3: **while** $\mathsf{Comp}(\{x[i]\}_{i\in[l_a]}, \{z[i]\}_{i\in[l_a]}) = 1$ **do**
4: $q[l_a - l_y] := q[l_a - l_y] + 1$;
5: $\{x[i]\}_{i\in[l_a]} := \mathsf{Sub}(\{x[i]\}_{i\in[l_a]}, \{z[i]\}_{i\in[l_a]})$;
6: **end while**
//Compute other digits of the quotient
7: **for** $i := l_a - 1$ to l_y by -1 **do**
8: **if** $x[i] = y[l_y - 1]$ **then**
9: $q[i - l_y] := b - 1$;
10: **else**
11: $q[i - l_y] := \frac{x[i]b+x[i-1]}{y[l_y-1]}$);
12: **end if**
13: $\theta := y[l_y - 1]b + y[l_y - 2]$;
14: **while** $q[i - l_y]\theta > x[i]b^2 + (x[i - 1]b + x[i - 2]$ **do**
15: $q[i - l_y] := q[i - l_y] - 1$;
16: **end while**
17: $\{r[i]\}_{i\in[l_a]} := \mathsf{LShiftB}(\{y[i]\}_{i\in[l_y]}, l_a - (i - l_y))$;
18: $\{w[i]\}_{i\in[l_a]} := \mathsf{Mul}(q[i - l_y], \{r[i]\}_{i\in[l_a]})$;
19: $\{x[i]\}_{i\in[l_a]} := \mathsf{Sub}(\{x[i]\}_{i\in[l_a]}, \{w[i]\}_{i\in[l_a]})$;
20: **if** $\mathsf{Comp}(\{x[i]\}_{i\in[l_a]}, 0) = 0$ **then**
21: $\{x[i]\}_{i\in[l_a]} := \mathsf{Add}(\{x[i]\}_{i\in[l_a]}, \{r[i]\}_{i\in[l_a]})$;
22: $q[i - l_y] := q[i - l_y] - 1$;
23: **end if**
24: **end for**
25: $\{r[i]\}_{i\in[l_y]} := \{x[i]\}_{i\in[l_y]}$;
26: **return** $\{q[i]\}_{i\in[l_a-l_y+1]}$ and $\{r[i]\}_{i\in[l_y]}$;

to realize big-integer operations and put an emphasis on what we need to implement symmetric cryptographic algorithms. It is an open question to implement the big-integer operations on PLC efficiently, and then realize public-key cryptography on PLC. We leave out this as one of the future works.

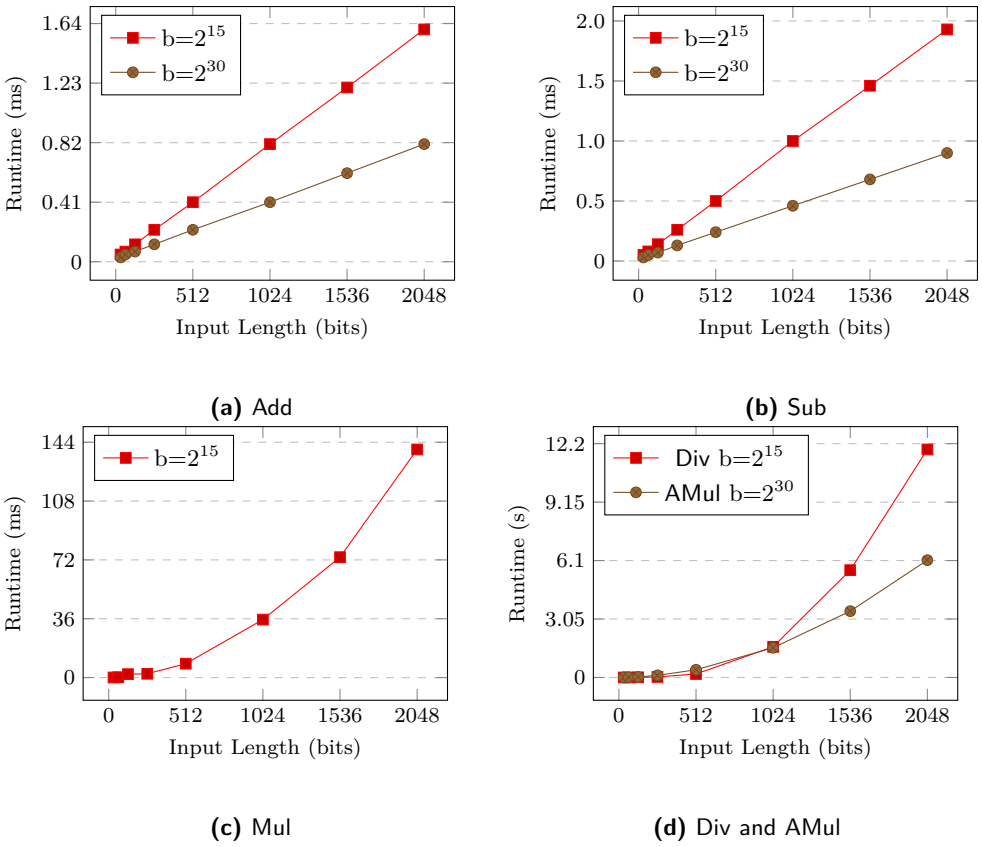


Figure 9: Runtimes of Big-integer Operations.

Algorithm 19: Big-integer Multiplication in an Additive Group

Input: Positive integers $(e, g) \in \mathbb{Z}_m$ which are represented as DINT ARRAY $x[l_a]$ and $y[l_a]$, i.e., each operand has l_a base $b = 2^{l_b}$ digits.

Output: DINT ARRAY $w[l_a]$ storing the product $w = e \cdot g = (w[l_a - 1] \dots w[0])_b \in \mathbb{Z}_m$.

1: $\{w[i]\}_{i\in[l_a]} := 0$;
2: **for** $i := 0$ to $l_a - 1$ by 1 **do**
3: $j := l_b$;
4: **while** $j >= 0$ **do**
5: $\{w[i]\}_{i\in[l_a]} := \mathsf{Add}(\{w[i]\}_{i\in[l_a]}, \{w[i]\}_{i\in[l_a]})$;
6: **if** $e[i].[j] = 1$ **then**
7: $\{w[i]\}_{i\in[l_a]} := \mathsf{Add}(\{w[i]\}_{i\in[l_a]}, \{g[i]\}_{i\in[l_a]})$;
8: **end if**
9: **if** $\mathsf{Comp}(\{w[i]\}_{i\in[l_a]}, 2m) = 1$ **then**
10: $\{w[i]\}_{i\in[l_a]} := \mathsf{Sub}(\{w[i]\}_{i\in[l_a]}, m)$; //$w \bmod m$
11: **end if**
12: $j := j - 1$;
13: **end while**
14: **end for**
15: **return** $\{w[i]\}_{i\in[l_a]}$;