Verifiable Computing in Avionics for Assuring Computer-Integrity without Replication

Johannes Reinhart*, Bastian Luettig*, Nicolas Huber†, Julian Liedtke† and Bjoern Annighoefer*
ORCID: 0000-0002-3512-5220, 0000-0002-9358-1611, 0000-0001-6905-3571, 0000-0002-8289-4970, 0000-0002-1268-0862
*Institute of Aircraft Systems, University of Stuttgart
†Institute of Information Security, University of Stuttgart

***Abstract*—Safety-critical digital systems such as Fly-by-wire control have demanding integrity and availability requirements which significantly exceed the occurrence rates of random hardware faults observed in digital computers. As a result, system designers need to employ reliable fault detection and mitigation techniques. Until now, the only method to achieve sufficiently reliable fault detection for systems that can cause hazardous or catastrophic events, is to replicate computer lanes and detect faults by comparing outputs. However, this comes with a large overhead in development cost, computing resources and additional requirements towards the application. We propose to apply a novel cryptographic technique to reliably detect faults and thereby assure integrity of avionics computers: Succinct Non-Interactive Arguments of Knowledge allow components to quickly verify computations without repeating the computation. We present a novel concept for building high-integrity avionics systems and set up a laboratory demonstrator for a simplified pitch control system. Our major results include the successful demonstration of the first self-proving and self-verifying cyber-physical system in a laboratory environment.**

***Index Terms*—Fault Detection, Integrity, Safety, Succinct Non-Interactive Argument of Knowledge, Systems Architectures, Verifiable Computing**

I. Introduction

Avionics systems whose failures can lead to hazardous or catastrophic events have challenging integrity and availability requirements. The European Aviation Safety Agency demands catastrophic failure conditions to be extremely improbable for large civil aircraft (CS 25.1309), which translates to $10^{-9}\,\mathrm{h}^{-1}$. Complex aviation computers can fail due to random hardware faults at rates of about $10^{-4}\,\mathrm{h}^{-1}$ to $10^{-5}\,\mathrm{h}^{-1}$: During operation, the A310 slat-flap computer had a failure rate of $8 \times 10^{-5}\,\mathrm{h}^{-1}$ [1], for the design phase of the Boeing Fly-By-Wire programme, a failure rate of $5 \times 10^{-5}\,\mathrm{h}^{-1}$ was assumed [2] and the Common Remote Data Concentrators in the Airbus A350 achieve failure rates of $10^{-5}\,\mathrm{h}^{-1}$. In order to satisfy the much lower required failure rates, system designers need to employ reliable fault detection and mitigation techniques. Standard methods for fault detection within a single computer lane include memory access checking, watchdog timers, checksum-based information redundancy or range and reasonableness checking [2], [3]. However, fault detection within a single lane covers only about $95\,\%$ [4] of all faults. This level of integrity is not sufficient for safety-critical systems. Until now, the only method to achieve sufficiently reliable fault detection is to replicate computer lanes and detect faults by comparing outputs of computer pairs [4], [5], triples [6] or quadruples [7]. Unfortunately, replicating computer lanes comes with a large overhead in development cost, resources and additional requirements towards the application. It usually requires replica management that includes ensuring synchronicity and data reconciliation among the computer lanes [3]. There are more efficient fault tolerance techniques for simple functions, but they usually come with the disadvantage of imprecise monitoring limits and therefore high numbers of false positives.

This work was supported by the Terra Incognita Programme of the University of Stuttgart.

We propose to apply a novel technique from the fields of cryptography to reliably detect faults and thereby assure integrity of avionics computers: Verifiable Computing [8] allows computations to be checked for correctness without requiring the computation to be carried out twice. This method became practical only about ten years ago, when the Pepper Project [9], Pinocchio [10] and TinyRAM [11], [12] emerged as the first practically usable implementations of such systems. A refinement of the original Pinocchio protocol by Groth [13] is considered the state of the art system in terms of verification speed. Newer systems include Bulletproofs [14], Ligero [15], STARK [16], Plonk [17], and Halo [18].

Our contributions are the introduction of a new concept using Verifiable Computing for building avionics systems with high integrity requirements. Compared to current systems, they will not require computer lanes to be replicated and, therefore, might be lighter and easier to develop. The new form for proving correctness comes with actual dissimilarity compared to standard monitoring implementations. Furthermore, we present a laboratory demonstrator representing a pitch control system as a proof-of-concept.

The remaining parts of the paper are structured as follows: In Section II, we introduce related work. In section III, we present a detailed description of our concept and the involved algorithms. Section IV shows how we implemented the concept, which tools we used and what modifications we applied to existing methods. In section V, we present our proof-of-concept laboratory demonstrator and in the following section VI we show, how the concept can be applied to typical avionic systems of large aeroplanes.

II. Related Work

So far, most applications of Verifiable Computation are related to the Web3 domain encompassing distributed ledger and blockchain technology. They include the privacy preserving digital currency ZCash [19], so-called Roll-ups for improving the efficiency of existing blockchains [20] and new blockchain protocols such as Mina [21]. Other possible applications ranging from healthcare services with sensitive personal data [22] to private voting systems [23] have been proposed. In the domain of aviation, Lorünser et. al [24] have presented a framework that uses Verifiable Computing to carry out non-centralized verifiable calculations for air traffic management. As to the authors' knowledge, Verifiable Computing has not yet been considered as a method for ensuring integrity in safety-critical systems.

In the railway domain, fail-safe systems with an extensive use of information redundancy and software flow checks for error detection have been certified and are used among others in the Metro Systems of Paris and Lyon [25]. The *vital coded microprocessor* presented by Forin in [26] achieves high fault coverage by encoding operands and defining operations that directly carry out calculations on the encoded operands without en- and decoding. Correctly encoded values have special properties throughout the computation, which are likely to be changed, should a fault occur, and thus can be checked at the end of a computation. This process is supported by special hardware. This method has been revisited and modified in more recent works [27], [28]. There are some parallels to our concept, however this approach makes many assumptions about hardware failures and their propagation in a microprocessor, which might be difficult to prove in a rigid qualification process, such as required for safety-critical systems in aviation.

III. Concept

We use the following notation throughout this paper: Lowercase letters denote values or data items, they can have a superscript index t denoting the current time step[1]. Subscript indeces are used for distinguishing data items with similar names or, if it is a number or i, for elements of an array of similar items. Functions are denoted with upper case letters. A function denoted with letter A is defined by an algorithm with an optional state, which appears both as an input and as an output to the function. A function denoted with letter C is an arithmetic circuit, used to represent a computation within a Verifiable Computing scheme. A relation $\mathcal{R}$ is a set of values or data items which satisfy a given system of equations.

Our goal is to realize a law on an avionics computer using Verifiable Computing, such that components using the results can reliably detect erroneous output with high probability. The law (Fig. 1) can be any system function, such as a control law, a monitoring application or some other data processing function. It can be formally defined as a function $(x_{out}, s_{law}^t) \leftarrow \mathrm{L}(x_{in}, s_{law}^{t-1})$ with internal state s_{law}, taking input values x_{in} and generating output x_{out}. The law is executed repeatedly such that the law's output depends on the input and the state from the previous time step.

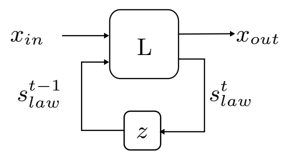

Fig. 1. Visualization of the law to be realized: the computer executes the law L every time step t. It takes input x_{in} and generates output x_{out}, with an internal state s_{law} stored in memory z.

If hardware would not fail, this could be achieved by implementing L as a computer program on the computing device. The device receives and sends input and output data via one or several databusses and stores its state in memory. Taking unreliable hardware into account, such a system could produce erroneous outputs indistinguishable from correct outputs. Therefore, extra effort is required to ensure integrity. The conventional approach uses two computing lanes running the same program. The lanes cross-compare the outputs and detect an error when the outputs differ. The error can be detected but not necessarily attributed to one of the two lanes, reducing reliability. Sensor values are checked in a similar fashion. The measurements of multiple independent sensors are compared to detect erroneous input values.

In our approach, we do not duplicate the computing device, but instead, we make use of a Verifiable Computing scheme with knowledge soundness. A Verifiable Computing scheme is a non-interactive proof system consisting of a prover algorithm, which produces a proof attesting to a correct evaluation of a circuit C for a statement Φ, and a verifier algorithm verifying the prover's claim. C can represent function evaluations and arithmetic constraints in general. In other words, the Verifiable Computing scheme allows proving the correct execution of a computer program represented as C. The circuit has input, output and intermediate wires that correspond to the input and output of the computer program and results of intermediate steps in the calculation. These can be either part of the public statement Φ or the hidden witness ω. Additionally, we make use of a digital signature scheme resistant against existential forgery and a hash function resistant against second preimage computation. These cryptographic primitives allow us to construct a system which satisfies integrity requirements without replicating the computing device. Fig. 2 depicts our construct. Each input message $m_{in,i} = (x, c, \sigma)_{in,i}$ consists of a value $x_{in,i}$, a message counter $c_{in,i}$ and a signature $\sigma_{in,i}$. The device evaluates the circuit C and runs the prover algorithm proving correct evaluation of C. C consists of verifying the signatures of every input message ($\mathrm{C}_{\mathrm{V_s}}$) and checking the message counter c (C_{mv}). The law L is also represented in the circuit (C_{L}), taking s_{law}^{t-1} and $x_{in} = (x_{in,1}, x_{in,2}, ...)$ as inputs and evaluating the updated state s^t and the output value x_{out}. The entire state $s = (s_{law}, c)$ contains the law's state s_{law} as well as the current message counter c, which

[1]For better readability, index t is omitted when its presence is easily inferred from the context.

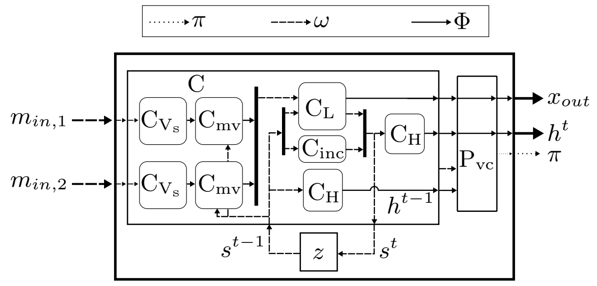

Fig. 2. Digital computing device fulfilling integrity requirements without replication. The device computes circuit C with statement Φ (solid lines) and witness ω (dashed lines). The state s is stored in memory z and the prover algorithm produces proof π (dotted lines).

increments in each time step (C_{inc}). The updated state is stored in the computing device's memory z. The device commits to its state by evaluating the state's hash h before and after the computation (C_H). The device finally outputs $m_{out} = (x_{out}, h^t, \pi)$ consisting of the law's output x_{out}, the proof π, and the hash of the updated state h^t.

The verifier can now check whether x_{out} has been calculated according to the law L by running the verifier algorithm on x_{out}, h^t, h^{t-1} and π. h^t must be stored for verifying the output at the next time step. If the computation were corrupted due to a hardware failure, the proof verification would fail. Here, we provide an intuitive explanation, why this is the case. A more formal argument is given later.

A. Failure types

There are four failure types possibly affecting the computation:

HW-FAIL-1: There could be any error in the evaluation of the circuit C. For example, the computer does not correctly compute the law. The verifier detects such errors according to the properties of the non-interactive proof system.

HW-FAIL-2: The state could have been changed between two time steps. As the hash function is resistant against second preimage computation, the device does not know another state evaluating to the same hash. Therefore, the hash value of the modified state will be different from the hash value of the state, which the system has committed to in the previous time step, causing the verification to fail.

HW-FAIL-3: The circuit including the law could have been evaluated correctly, but with input values not from the actual received message. In this case, the computing device must know a signature of such a value, which is impossible due to the signature scheme being secure against existential forgery.

Type 4 results from corrupted input messages due to a failed sensor and contains a corrupted value, but a correct signature. The conventional approach of using multiple similar, independent sensors resolves this issue. We assume, that independent sensors do not fail in a similar manner at the same time. Hence a deviation between redundant values will be an indicator for a fault. We do not consider this type further, as we will not solve it using Verifiable Computing.

Following, we provide a more formal description of the components and their properties.

B. The Verifiable Computation Scheme

For proving and verifying a computation in a Verifiable Computing scheme, the computation is represented as a circuit $C : \mathbb{H}^n \times \mathbb{H}^h \to \{0, 1\}, (\Phi, \omega) \mapsto C(\Phi, \omega)$ taking statement Φ and witness ω and outputting 1, if the circuit is satisfied, meaning the program was executed correctly, or 0 otherwise. The elements of the statement and witness are defined over the domain $\mathbb{H}$, which depends on the chosen proof system. Given a subset of ω and Φ as input elements, it is possible to evaluate the circuit, which means to efficiently compute the remaining elements of Φ and ω. As an example, our notation "$\omega_L, x_{out}, s^t_{law} \leftarrow$ evaluate $C_L(x_{in}, s^{t-1}_{law})$" means, that given the inputs x_{in} and s^{t-1}_{law}, which are part of the witness and statement of the circuit C_L, we calculate the remaining parts of the witness and statement, which is $(\omega_L, x_{out}, s^t_{law})$. The circuit C defines the relation $\mathcal{R} = \{(\Phi, \omega) : C(\Phi, \omega) = 1\}$. We use a Succinct Non-Interactive Argument of Knowledge (SNARK), which allows a computation to be proven with just a single short piece of data, called the proof π. Following the definition of Bitansky et al. [29], a Non-Interactive Argument of Knowledge is the set of three algorithms (G_{vc}, P_{vc}, V_{vc}):

- $(\kappa_{vc}, \tilde{\kappa}_{vc}) \leftarrow G_{vc}(C)$: The generator algorithm generates a prover-key κ_{vc} and a verifier-key $\tilde{\kappa}_{vc}$ given the circuit C.
- $\pi \leftarrow P_{vc}(\kappa_{vc}, \Phi, \omega)$: The proving algorithm takes the prover-key and computes a proof π given the statement and witness.
- $v \leftarrow V_{vc}(\tilde{\kappa}_{vc}, \pi, \Phi)$: The verifying algorithm takes the verifier-key, the statement and the proof and produces the verification result $v \in \{0, 1\}$ indicating either a pass (1) or a rejection (0).

They satisfy the following properties:

Completeness: If the output is the correct result of the computation, the verifier algorithm approves with a probability (Pr) of 100%:

$$\Pr\left[V_{vc}(\tilde{\kappa}_{vc}, \pi, \Phi) = 1 \;\middle|\; \begin{array}{c} (\kappa_{vc}, \tilde{\kappa}_{vc}) \leftarrow G_{vc}(C) \\ \pi \leftarrow P_{vc}(\kappa_{vc}, \Phi, \omega) \\ \forall (\Phi, \omega) \in \mathcal{R} \end{array} \right] = 1. \tag{1}$$

Computational knowledge soundness: If a prover does not know a witness ω or outputs a value, which is not a result of the computation, then the verifier accepts with negligible probability (ε). Formally, for every efficient[2] adversary A, there exists an efficient extractor E, such that

[2]An efficient algorithm takes polynomial execution time and memory. Informally, the basic argumentation is, that by increasing the problem size, there will be a point, where an efficient algorithm can be executed with bounded resources, while an inefficient algorithm (i.e. its complexity is more than polynomial) cannot be executed any more.

$$\Pr\left[\begin{array}{c} \mathrm{V_{vc}}(\tilde{\kappa}_{vc},\pi,\Phi)=1 \\ (\Phi,a)\notin\mathcal{R} \end{array}\middle| \begin{array}{c} (\kappa_{vc},\tilde{\kappa}_{vc})\leftarrow \mathrm{G_{vc}}(\mathrm{C}) \\ (\pi,\Phi)\leftarrow \mathrm{A}(z,\kappa_{vc}) \\ a\leftarrow \mathrm{E}(z,\kappa_{vc}) \end{array}\right]=\varepsilon. \tag{2}$$

Knowledge soundness is necessary for hiding the signed input values and the state of the computation as part of the witness, reducing the information to be transmitted between prover and verifier.

C. Signature Verification

We use an asymmetric signature scheme $(\mathrm{G_s},\mathrm{S_s},\mathrm{V_s})$ consisting of a key generator $(\tilde{\kappa}_s,\kappa_s)\leftarrow \mathrm{G_s}()$, a signing algorithm $\sigma\leftarrow \mathrm{S_s}(\kappa_s,m)$ and a verification algorithm $v\leftarrow \mathrm{V_s}(\tilde{\kappa}_s,\sigma,m),\ v\in\{0,1\}$ with properties:

Completeness: Messages signed with the correct key are accepted:

$$\Pr\left[\mathrm{V_s}(\tilde{\kappa}_s,\sigma,m)=1\middle| \begin{array}{c} (\tilde{\kappa}_s,\kappa_s)\leftarrow \mathrm{G_s}() \\ \sigma\leftarrow \mathrm{S_s}(\kappa_s,m) \end{array}\right]=1. \tag{3}$$

Resistance against existential forgery: Signatures to new messages cannot be created without the private key, even after seeing arbitrarily many valid signatures. For any adversary A and chosen messages m_i:

$$\Pr\left[\begin{array}{c} \mathrm{V_s}(\tilde{\kappa}_s,\sigma,m)=1 \\ m\notin\{m_i\} \end{array}\middle| \begin{array}{c} (\tilde{\kappa}_s,\kappa_s)\leftarrow \mathrm{G_s}() \\ \sigma_i\leftarrow \mathrm{S_s}(\kappa_s,m_i) \\ (m,\sigma)\leftarrow \mathrm{A}(\{m_i,\sigma_i\}) \end{array}\right]=\varepsilon. \tag{4}$$

D. Hash function

We use a hash function $h\leftarrow \mathrm{H}(m)$ which is resistant against second preimage attacks.

Resistance against second preimage attacks: Given a message and it's hash, a second different message with the same hash cannot be found efficiently. For any efficient adversary A:

$$\Pr\left[\begin{array}{c} h=\mathrm{H}(m_2) \\ m_2\neq m \end{array}\middle| \begin{array}{c} h=\mathrm{H}(m) \\ m_2=\mathrm{A}(h,m) \end{array}\right]=\varepsilon. \tag{5}$$

E. Formal Failure Model

We model the behaviour of a digital computer as the composition of the computer program and the hardware transfer function $\mathfrak{H}^t$. The hardware transfer function $\mathfrak{H}^t$ describes the effect of the unreliable computing device. At time steps, when the hardware works correctly, it is the identity function E.

F. Algorithms

Our proposed system requires three algorithms: The input conditioning algorithm (A_{sensor}) for signing the input values runs on devices providing inputs for the law. This could be sensors or other upstream computing devices. The computing algorithm ($A_{computing}$) runs the actual law and generates the proof. The verification algorithm ($A_{actuator}$) runs on downstream devices or actuators. The algorithms may have a state that is stored between consecutive invocations, e.g. for using $A_{computing}$, our notation is: $m_{out},s^t = A_{computing}(m_{in},s^{t-1})$.

The input conditioning increases a message counter and signs an input value together with the message counter.

Algorithm 1 Input Conditioning A_{sensor}

```
Input: $x_{in}$
Output: $m_{in}$
State: $c$
1: $c \leftarrow c+1$
2: $\sigma \leftarrow \mathrm{S_s}(\kappa_s,(x_{in},c))$
3: $m_{in} \leftarrow$ pack $(x_{in},c,\sigma)$
```

The computing algorithm ($A_{computing}$) evaluates the circuit C, finding assignments to the the hash function relations

$$\mathcal{R}_\mathrm{H}=\{(\omega_\mathrm{H},s,h):\mathrm{H}(s)=h\},$$

signature verification relations

$$\mathcal{R}_{\mathrm{V_s}}=\{(\omega_{\mathrm{V_s}},x,c,\sigma):\mathrm{V_s}(\tilde{\kappa}_s,\sigma,(x,c))=1\},$$

the message counter check relations

$$\mathcal{R}_\mathrm{mv}=\{(\omega_\mathrm{mv},c,c_{in}):c=c_{in}\},$$

the message counter increase relation

$$\mathcal{R}_\mathrm{inc}=\{(\omega_\mathrm{inc},c^{t-1},c^t):c^t=c^{t-1}+1,\},$$

and the law relation

$$\begin{aligned}\mathcal{R}_\mathrm{L}=\{(\omega_\mathrm{L},x_{in},x_{out},s^t_{law},s^{t-1}_{law}):\\ \mathrm{L}(x_{in},s^{t-1}_{law})=(x_{out},s^t_{law})\}.\end{aligned}$$

The statement $\Phi=(x_{out},h^t,h^{t-1})$ consists of the law's output, and the two hash values. The input values, and intermediate values of the circuit assignment are part of the witness. Note that the evaluations of the hash circuit ($\mathrm{C_H}$), law circuit ($\mathrm{C_L}$) and message check circuit ($\mathrm{C_{inc}}$) generate valid assignments for the corresponding relations, while the evaluations of $\mathrm{C_{V_s}}$ and $\mathrm{C_{mv}}$ can only yield valid assignments, if the signature and the message counter are correct.

Algorithm 2 Computing $A_{computing}$

```
Input: $m_{in}$
Output: $m_{out}$
State: $s$
1: $\omega_\mathrm{H},h^{t-1} \leftarrow$ evaluate $\mathrm{C_H}(s)$
2: for all $m_{in,i}$ in $m_{in}$ do
3:     $(x_{in,i},c_i,\sigma_i) \leftarrow$ unpack $m_{in,i}$
4:     $\omega_{\mathrm{V_s},i} \leftarrow$ evaluate $\mathrm{C_{V_s}}(x_{in,i},c_i,\sigma_i)$
5:     $\omega_\mathrm{mv} \leftarrow$ evaluate $\mathrm{C_{mv}}(c,c_{in,i})$
6: end for
7: $\omega_\mathrm{inc},c^t \leftarrow$ evaluate $\mathrm{C_{inc}}(c^{t-1})$
8: $\omega_\mathrm{L},x_{out},s^t_{law} \leftarrow$ evaluate $\mathrm{C_L}(x_{in},s^{t-1}_{law})$
9: $\omega_\mathrm{H},h^t, \leftarrow$ evaluate $\mathrm{C_H}(s)$
10: $\pi \leftarrow \mathrm{P_{vc}}(\kappa_{vc},\omega,\Phi)$
11: $m_{out} \leftarrow$ pack $(x_{out},h_t,\pi)$
```

The third algorithm checks the validity of an output message and returns the output value x_{out} and a status value v indicating the values validity:

```
Algorithm 3 Verification $A_{actuator}$
Input: $m_{out}$
Output: $x_{out},\ v$
State: $h^{t-1}$
1: $(x_{out}, h^t, \pi) \leftarrow$ unpack $m_{out}$
2: $v \leftarrow \mathrm{V_{vc}}(\tilde{\kappa}_{vc}, \pi, (x_{out}, h^{t-1}, h^t))$
3: $h^{t-1} \leftarrow h^t$
```

We formally define our notion of integrity for a system and show that the construction satisfies the definition. We assume, that the system behaves correctly for the first $t-1$ time steps and look at the system at time step t. The device has correct state s^{t-1}. The device receives an input message for each input value: $m_{in,i} \leftarrow A_{sensor}(x_{in,i})$, $m_{in} = (m_{in,1}, m_{in,2}, ...)$.

System Integrity:

1. If there is no failure in the hardware (the transfer function is the identity function, $\mathfrak{H} = E$), the system verification algorithm accepts and outputs the value according to L:

$$\Pr\left[\begin{array}{c|c} v = 1 & \mathfrak{H} = E \\ x_{out}, s^t_{law} = & m_{out}, s^t \leftarrow \mathfrak{H}(A_{computing}(m_{in}, s^{t-1})) \\ \mathrm{L}(x_{in}, s^{t-1}_{law}) & x_{out}, v \leftarrow A_{actuator}(m_{out}) \end{array}\right] = 1. \tag{6}$$

2. If the system outputs erroneous values, the probability that $A_{actuator}$ accepts is negligible:

$$\Pr\left[\begin{array}{c|c} & m_{out}, s^t \leftarrow \mathfrak{H}(A_{computing}(m_{in}, s^{t-1})) \\ v = 1 & z \leftarrow A_{actuator}(m_{out}) \\ & x_{out} \leftarrow \text{ unpack } m_{out} \\ & x_{out}, s^t_{law} \neq \mathrm{L}(x_{in}, s^{t-1}_{law}) \end{array}\right] = \varepsilon. \tag{7}$$

We first show, that (6) holds. In the case that $\mathfrak{H} = E$, then due to (3), $(\omega_{\mathrm{V_s}}, x, c, \sigma) \in \mathcal{R}_{\mathrm{V_s}}$, and as both sensor and device have increased the message counter, $(\omega_{\mathrm{mv}}, c, c_{in}) \in \mathcal{R}_{\mathrm{mv}}$. Therefore for the whole circuit $(\omega, \Phi) \in \mathcal{R}$. Then, according to (1), $\mathrm{V_{vc}}$ outputs 1.

Next, we show that (7) holds for our system.

We split the condition $x_{out}, s^t_{law} \neq \mathrm{L}(x_{in}, s^{t-1}_{law})$ into the three failure types from III-A:

1) $x_{out}, s^t_{law} \notin \{(x, s_2) : x, s_2 = \mathrm{L}(x_{in}, s_1)\ \forall x_{in}, s_1\}$
2) $x_{out}, s^t_{law} = \mathrm{L}(x_{in}, s_1) \wedge s_1 \neq s^{t-1}_{law}$
3) $x_{out}, s^t_{law} = \mathrm{L}(x, s^{t-1}) \wedge x \neq x_{in}$

HW-FAIL-1: In this case, $(\omega_{\mathrm{L}}, x_{in}, x_{out}, s^t_{law}, s^{t-1}_{law}) \notin \mathcal{R}_{\mathrm{L}}$ and therefore $(\Phi, \omega) \notin \mathcal{R}$. According to (2), the probability that $\mathrm{V_{vc}}$ accepts is negligible.

HW-FAIL-2: Either $\mathcal{R}_{\mathrm{H}}$ would not be satisfied, resulting in the verifier rejecting due to (2), or if the relation was satisfied, then a state must be extractable from the system, such that $h = \mathrm{H}(s) = \mathrm{H}(s^{t-1})$ with $s \neq s^{t-1}$. According to (5), this is improbable.

HW-FAIL-3: Here, $(\omega_{\mathrm{V_s}}, x, c, \sigma) \notin \mathcal{R}_{\mathrm{V_s}}$ and therefore $(\Phi, \omega) \notin \mathcal{R}$ as in case 1. Otherwise, due to (2), a signature must be extractable from the device, such that $\mathrm{V_s}(\tilde{\kappa}_s, \sigma, (x_{in}, c_{in})) = 1$. As c_{in} changes in each iteration, (x_{in}, c_{in}) is not in the list of messages seen by the system, therefore (4) applies, stating that this case has negligible probability.

IV. Implementation

For instantiating the Verifiable Computing scheme, we use a modified version of the Groth16 preprocessing pairing-based SNARK presented in [13]. This system is considered state-of-the-art in terms of verification speed and proof size and is well established in several Web3 applications, for example ZCash [30] and Mina [21] use a slightly modified version of this SNARK. As Groth16 is pairing-based, the domain $\mathbb{H}$ is the prime field $\mathbb{F}_r$ with r being the order of the prime subgroup $\mathbb{G}_1$ of points on a pairing-friendly elliptic curve. The relation $\mathcal{R}$ is a rank one constraint system (R1CS). We will provide details on the constraints in our setup later in this chapter. As a basis for our implementation, we used the open-source C++ library *libsnark*[3], which implements a number of pairing-based SNARK systems and offers a choice of pairing-friendly curves. We also made use of the open-source library *ethsnarks*[4], a project originally intended for enabling SNARKs on the Ethereum blockchain. It conveniently provides implementations of R1CS circuits compatible with *libsnark* including circuits for hash functions and EdDSA signature verification.

We made the following modifications to the original Groth16 proof system: 1.) We removed the randomization factors in the prover algorithm, which are only needed for the zero-knowledge property of the SNARK. Information exposure is not of concern in our application. 2.) Additional to *libsnark*'s Barreto-Naehring curve with prime order subgroup size of 254 bits (presented in [12]), we used a Barreto-Naehring curve with a 124 bit prime order subgroup size and a twisted Edwards curve with a 61 bit prime order subgroup size for speeding up operations over the field $\mathbb{F}_r$ and the elliptic curve group $\mathbb{G}_1$. We implemented both curves taking the existing curve implementations as a template. Using curves with small subgroup size is justified, as our system is not required to be secure against intentional abuse, but must rather protect against randomly occurring hardware faults, which significantly lower the requirements on the group size. 3.) Some smaller performance improvements include removing the representation of the circuit from the proving key in *libsnark* and using an FFT algorithm tweaked for better cache usage. These two improvements were taken from *Loopring*'s[5] fork of *libsnark*, see [31] for details.

For the digital signature algorithm, we chose EdDSA, which allows fast signature verification when encoded in R1CS. EdDSA signature verification consists of operations on group

[3]https://github.com/scipr-lab/libsnark
[4]https://github.com/HarryR/ethsnarks
[5]*Loopring* is an application on the Ethereum blockchain using SNARKs.

elements of elliptic curves over prime fields, therefore it can be adapted to SNARK circuits by selecting a so-called SNARK-friendly elliptic curve (inner curve), whose definition field has the same order as the prime order group of the SNARK curve (outer curve). This a common optimization for cryptographic primitives embedded in SNARKs [32]–[36]. However, it requires a different inner curve for each outer curve.

We added the following modifications to the EdDSA signature verification circuit from *ethsnarks*: We replaced the circuits for elliptic curve point addition and multiplication with those used by ZCash [30] to reduce the number of constraints. Additionally, as the inner curve provided by *ethsnarks* only fits to the *libsnark* Barreto-Naehring curve, we implemented inner curves for the two additional outer curves.

A. Choice of Security Parameters

In our application, we want to reliably detect computing faults due to random hardware failures. In order for the system to be certified, it must be shown, that the probability of an undetected failure is sufficiently low, depending on the failure effect. For that, we use a conservative failure model, that provides an upper bound to the undetected failure probability. We model the malfunctioning device as a computationally bounded adversary, trying to cause an undetected failure of the system. Using cryptanalytic tools, we can then estimate the adversary's success probability. This model is justified, as a device breaking a cryptographic scheme due to some random malfunction cannot have a higher probability of success than the same device using an efficient algorithm to break the scheme. The security parameter allows to set the upper bound for the probability of an undetected failure: The larger the parameter, the less probable an undetected failure becomes. However, computational effort also increases with the security parameter.

Here we provide a rough sketch on how to chose the security parameter. The details in a real system will depend on the system's purpose, requirements, environment, etc. First, a functional hazard assessment usually yields a failure effect, to which a targeted maximum failure probability can be assigned. On large transport aircraft, for failures causing catastrophic events, this is typically on the order of $10^{-9}\,\mathrm{h}^{-1}$. Dividing it by the device's failure probability results in the maximal probability for not detecting the failure, which would be $10^{-9}\,\mathrm{h}^{-1}/10^{-5}\,\mathrm{h}^{-1} = 10^{-4}$ for a modern robust aviation computer.

If we assume that a device's failure must be detected within $\Delta t = 1\,\mathrm{s}$, we need to show, that an algorithm running for $1\,\mathrm{s}$ has a probability of less than 10^{-4} of generating a fake proof. As evident from the qualitative analysis in chapter III, there are several ways to produce a fake proof, such as forging a signature, finding a second-preimage to the hash function or directly targeting the proof system. The parameters of each of the cryptographic primitives should be chosen, such that a breach is equally impossible. For sake of brevity, we will only consider the direct targeting of the proof system and chose parameters deemed equivalent for the other cases. A more rigid analysis should consider every case individually.

We note that the security of the Groth16 proof system is based on the hardness of solving the discrete logarithm problem on the underlying elliptic curve. That is, if one is able to fake Groth16 proofs, one is also able to compute discrete logarithms on that curve. Therefore, in the following, we will analyse the hardness of the discrete logarithm problem in our setting. The best known generic algorithm for solving this problem is Pollard's rho algorithm. The success probability of this algorithm depends on the number of iterations k it is executed and on the security parameter, which is the size of the group $N = |\mathbb{G}|$ (see e.g. [37]):

$$\Pr = \sum_{i=1}^{k} \frac{i}{N} \exp(-i^2/(2N)). \tag{8}$$

For the curves mentioned above, we have $\log_2(|\mathbb{G}_1|) = 254$ and 124 for the Barreto-Naehring curves (BN254, BN124) and 61 for the Edwards curve (ED61). The other groups have size $\log_2(|\mathbb{G}_2|) = 2\log_2(|\mathbb{G}_1|), \log_2(|\mathbb{G}_T|) = 12\log_2(|\mathbb{G}_1|)$ for the Barreto-Naehring curve and $\log_2(|\mathbb{G}_2|) = 3\log_2(|\mathbb{G}_1|), \log_2(|\mathbb{G}_T|) = 6\log_2(|\mathbb{G}_1|)$ for the Edwards curve.

Each iteration of Pollard rho involves one group addition operation. We profile group additions for each of our curves on our target computing device (Intel i5-12500T processor, restricted to a single core) to estimate k for $\Delta t = 1\,\mathrm{s}$. Using (8) gives the maximal success probabilities in table I for the selected curves.

TABLE I
SUCCESS PROBABILITY OF POLLARD RHO ALGORITHM FOR $\Delta t = 1\,\mathrm{s}$ ON OUR TARGET COMPUTING DEVICE.

Curve	**ED61**	**BN124**	**BN254**
$\mathbb{G}_1$	8×10^{-6}	9×10^{-45}	2×10^{-96}
$\mathbb{G}_2$	1×10^{-26}	1×10^{-66}	4×10^{-218}
$\mathbb{G}_T$	6×10^{-65}	9×10^{-143}	$<2 \times 10^{-308}$

As we can see, the significant restricition of the computing capability allows to choose significantly lower security parameters than recommended for security applications, as these usually try to protect against an attacker with a large cluster of high-performing special hardware and several years of runtime. With our assumptions and model, the Edwards curve would be sufficient. It should be noted, however, that if the malfunctioning device is modelled such that it is not restricted to just generic group operations, more efficient algorithms, such as the General Number Field Sieve for solving the discrete logarithm on the target group $\mathbb{G}_T$, exist. This example should not undermine the fact, that finding a suitable security parameter can be quite difficult [38] and requires a more careful and thorough analysis than what would fit into this paper. However, keep in mind, that we conservatively assume that a random hardware fault leads to an efficient algorithm for faking a proof.

B. Implementation of the Circuit

While most computations on digital avionics computers use integers, fixed-point or floating-point numbers, the circuits for our SNARK proofs are defined over a finite field $\mathbb{H} = \mathbb{F}_r$ corresponding to the curve's prime order group $\mathbb{G}_1$. Therefore, finite field elements must be used to represent the actual data types. Addition and multiplication of finite field elements correspond to addition and multiplication of integers modulo the field order. We represent unsigned integers $i_u \in [0, n]$ and signed integers $i_s \in [-n, n]$ with a single prime group element $g \in \mathbb{F}_r$. Range checks of g at appropriate locations in the circuit enforce that overflows cannot occur. Note that the maximal range is limited by $|\mathbb{F}_r| = |\mathbb{G}_1|$, such that with our smallest curve, integers are restricted to at most 61 bits. Negative integers are represented by the additive inverse of the field $-|i| = r - |g|$. Therefore, addition and multiplication are trivial. Integer division (with remainder) is more expensive in terms of circuit size, as this operation requires several range constraints. For representing decimal numbers we did not implement floating-point arithmetic but instead use simpler fixed-point arithmetic. We represent fixed-point numbers d as a tuple of a field element g and an exponent e: $d = g \cdot 2^{-e}$. The exponent is not a variable in the circuit but a fixed constant associated to the value. It is publicly known as part of the circuit. Addition of a number with the same exponent is achieved by adding the field elements. For an addition of a number with a different exponent, the numbers are first converted to have a common exponent. Multiplication of a fixed-point value to another is achieved by first multiplying the field element and then dividing by 2^e.

A central part of the circuit is the law L. In control applications, the essential part is typically a discretized linear controller. In our proof-of-concept demonstration, we use a PT_1 controller with time-constant T_1, steady-state gain P and sample time T_s. In discretized state-space form, it is

$$\begin{aligned} x^t &= ax^{t-1} + (1-a)u^t \\ y^t &= Px^{t-1} \\ a &= \exp(-T_s/T_1). \end{aligned} \tag{9}$$

For sufficiently reducing rounding errors, the internal state x is implemented as a fixed point number $x = \tilde{x} \cdot 2^{-e}$ with exponent of $e = 16$. The constraints for the circuit are[6]:

$$\begin{aligned} \lfloor a \cdot 2^e \rfloor \tilde{x}^t &= 2^e q_1 + q_2 \\ q_1 + \lfloor (1-a)2^e \rfloor u &= \tilde{x}^{t+1} \\ \lfloor P \rfloor \tilde{x}^t &= 2^e y + q_3 \end{aligned} \tag{10}$$

Additionally, range constraints need to be set for some of the variables:

$$\begin{aligned} 0 \le q_2 < 2^e, \quad -2^e \le q_1 < 2^e \\ 0 \le q_3 < 2^e, \quad -2^e \le y < 2^e. \end{aligned} \tag{11}$$

[6] $\lfloor .. \rfloor$ indicates cutting off decimal places.

Each of these range constraints translates to e R1CS constraints, see [30] for details.

The state x^{t-1}, x^t is input of the hash function, as it is part of the whole system's state s. u is an input value, and y is the output value. For redundant input values, a monitoring function is implemented, which in our example is a simple range check between the two input values with a fixed threshold Δ, and a passthrough of one of the inputs, which is cheaper in terms of constraints than an average:

$$\begin{aligned} -\Delta \le u_1 - u_2 < \Delta \\ u = u_1. \end{aligned} \tag{12}$$

V. Demo Example

We chose a simplified pitch controller that deflects an elevator to demonstrate our initial concept. Fig. 3 shows the architecture and exchanged signals and Fig. 4 is a photograph of the laboratory setup. The system consists of a stick, an input/output module (IOM) that reads the sensor value, a computing module (CPM) that computes the target elevator deflection and the IOM2 that controls the elevator. The demonstrator shows three different aspects: (1) message signing (IOM), (2) law execution of proof computation (CPM), and (3) proof verification (IOM2).

A. General architecture and operation

The IOM reads the current stick position $\tilde{x}_\Phi$, executes the sensor-application (A_{sensor}), which generates the corresponding signature σ_ϕ, and transmits the data inside message $m_{sensor} = (x_\phi, c_\phi, \sigma_\phi)$ to the Core Processing Module (CPM).

The CPM executes the computing application ($A_{computing}$), which checks the signature and computes the control algorithm. The CPM generates a proof π. It then sends $m_{command} = (x_\eta, \mathrm{h}, \pi)$ to the outbound IOM2.

IOM2 executes the actuator application ($A_{actuator}$), checks if the proof matches the command and controls the actuator accordingly. If the proof fails, IOM2 indicates an error and fixes the actuator position.

All computers, IOM, CPM, and IOM2, utilize an Intel Core i5-12500T CPU and 16GByte RAM. They connect to the same private Ethernet network. The modules operate on a standard Debian GNU/Linux Minimal, we did not use a real-time Kernel and did not add any specific configuration. The system did not execute any GUI and was remotely accessed using SSH over Ethernet via a dedicated network.

During a one-time compile action, each module compiles its own application. The keys are then distributed to each device.

B. Demonstration Scenario

We defined three demonstration scenarios, one shows normal operation (i), one an error within the CPM (ii), and the third a discrepancy in the sensor signals (iii).

For *scenario (i)*, we started the applications on IOM (A_{sensor}), CPM ($A_{computing}$) and IOM2 ($A_{actuator}$) and observed sensor data, computation and actuator. First, the stick was deflected and the actuator was expected to follow. For

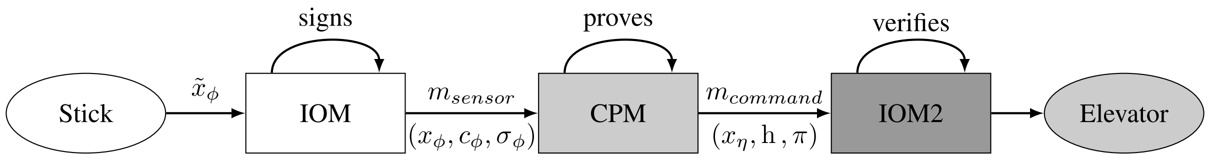

Fig. 3. Demonstration concept with exchanged data

scenario (ii), a hardcoded fault in the CPM software was triggered, and the elevator was expected to stop its movement. IOM2 should indicate this condition. For *scenario (iii)*, we adapted scenario (i) and connected two sensors with IOM to the CPM. The CPM application ($A_{computing}$) compares the sensor values. Due to an out-of-limit discrepancy, the application is expected to throw an error.

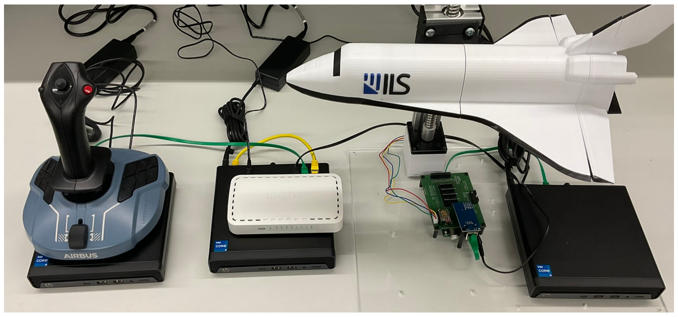

Fig. 4. Laboratory demonstrator of a simplified pitch control system with a real-time Verifiable Computing controller

C. Results

The demonstration scenarios performed correctly. Faults within the signature, the message counter, and the command were properly detected and correctly attributed. The first scenario even revealed an integer overflow bug that we unintentionally implemented. The scenarios (ii) and (iii) show that the failures are detected as intended. The verifiable computing approach ensures that sensor monitoring works correctly and enables correct failure attribution. The latter becomes much more relevant with more sensors in the system and applications on the CPM. Then the verifier can passivate select commands instead of the entire module.

For the given hardware and setup, the system operated at 10 Hz. The main limiting factor in terms of computational runtime is the proof generation on the CPM computer.

VI. Application in Avionics Systems

In the following section, we discuss, how we imagine the concept to be applied to typical avionics systems. We consider the two types of system architectures typically found in large transport aircraft: In a federated architecture, the different system functions are implemented on separate hardware devices, each responsible for a limited number of tasks. Typical communication buses between devices are ARINC 429 and CAN. Additionally, computing devices have input and output interfaces for directly interacting with sensors or actuators using analog or discrete signals. In our concept, sensor values are signed. This can either be carried out by the sensor itself, if it is equipped with appropriate electronics and communicates via a field bus. In the case that it only provides discrete or analog signals, the computing device interfacing the sensor can digitally sign the values. Actuator electronics or the computing device interfacing it verifies the command. The outlined concept ensures the integrity of computations in between. As an example, we consider an autopilot inspired by the Airbus A340 autoflight system. The central part of the autoflight system in the classical concept are four flight guidance computer lanes (FGC) arranged in pairs, that allow a fail-operational, fail-passive behaviour as required for automatic landings. The autopilot gets the pilot's input via the Flight-Control Unit (FCU), the Flight-Management System (FM), Navigation Computers (NAV) and Air Data Inertial Reference Units (ADIRU). Autoflight commands are passed to the Primary (PRIM) Flight Control Computers, which together with additional Secondary Flight Control Computers (SEC) command the control surface actuators. With our concept, the two duplex guidance computers could be reduced to two simplex guidance computers (Fig. 5) while still having fail-operational, fail-passive behaviour, as the original system. The FCU, FMs, ADIRUs and NAVs would in this case sign their data using A_{sensor}. The simplex flight guidance computers can then compute their commands and prove the computation's correctness with $A_{computing}$. The FCPCs can run the verifier algorithm $A_{actuator}$. If one of the FGCs failed, this would be detected by the verification algorithm, and the FCPCs would accept commands from the remaining FGC. There are still many obstacles for realizing such a system, as discussed later, with one of the biggest challenges being the long runtime of the proving algorithm.

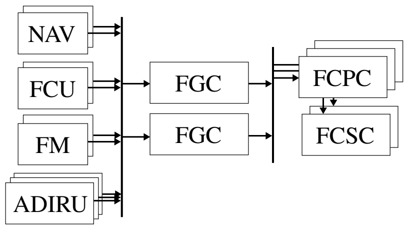

Fig. 5. Proposal of an autoflight system with fail-operational, fail-passive behaviour: Instead of two duplex guidance computers, two single lane guidance computers can be used if failures of a single lane are detected using Verifiable Computing.

Our concept may also be applicable in the setting of Integrated Modular Avionics (IMA). IMA systems communicate via a common Data Network, such as ARINC 664 [39] or ARINC 629 [40]. There are special interfacing devices for multiplexing and formatting raw sensor data onto the network and for controlling actuators called Remote Data Concentrators (RDC) [41] or input/output modules (IOM) [40]. The computation is usually carried out on several central computers which act as a shared computing resource, reducing the total number of computing devices. In such a system architecture, signing the data could be taken over by the interfacing devices. This would have the advantage, that not every datum requires its own signature, but several data could be signed together, reducing the number of signature checks in the prover algorithm. Verification can be carried out on the interfacing devices directly controlling the actuators.

VII. Discussion

Our proposed technique allows for new system architectures addressing the challenges of current fault tolerant architectures: 1.) The redundancy management in current systems introduces high complexity. In order to reliably detect random hardware faults by comparing outputs of replicated systems, the computer replica must produce similar outputs (so-called agreement) in the case that no fault occurred and different outputs in the case of faults. Lala and Harper claim that in order to achieve $100\,\%$ coverage of faults using the comparison method, replicas must exhibit exact agreement [3], which means that their outputs must be bitwise identical. Otherwise, in the case of only approximate agreement, thresholds for the comparison must be selected, which are usually found empirically and constitute a compromise between good fault coverage and low number of false positives [3], [5]. It is questionable, whether approximate agreement systems can scale to more complex applications, such as image recognition or audio processing, as finding a threshold might become impossible. On the other side, exact agreement systems also face significant challenges: First, replicated systems must be similar, if not identical, in order to produce bitwise comparable outputs. This increases susceptibility to common mode failures. Second, exact agreement must be enforced explicitly. This includes synchronization of redundant computers for limiting the time skew between corresponding operations [3], [42], and establishing agreement between input data and state data. Establishing agreement amongst several computers is non-trivial, as asymmetric behavior of faulty components, also known as the byzantine generals problem [43], must be addressed properly [44], [45]. 2.) There is a large overhead in resources in the final system. In the Airbus A340 Primary Flight Control System, computers are duplicated according to the Control-Monitoring strategy resulting in 2x5 computer lanes allocated only for flight control [46]. The Boeing 777 Primary Flight Control consists of a triple-triple redundant system with a fully-verifiable quad-simplex backbone system accounting to a total of 13 computers allocated for flight control only [47]. If integrity was ensured by Verifiable Computing, redundancy would only be required for availability, and not for integrity any more. This new approach therefore has the potential to reduce redundancy management complexity and the number of hardware resources required.

On the other hand, the application of Verifiable Computing for safety in avionics is completely new and exhibits potential drawbacks. For instance, there is a lack of mature tooling and industrial experience for applying this method. Our impressions from building circuits with the *libsnark* library are that this is a very cumbersome and error prone process. There are some software packages simplifying the development of SNARK circuits [48], [49], but for a rigid development process, further tooling, e.g. for formally proving the correctness of circuits and for traceability to requirements are necessary. Another drawback is the large overhead in computing time, specifically for the proving algorithm. Fortunately, there is current research focussing on optimizing the size of constraint systems e.g. [49]–[51] and on new proof systems with better performance charactersitics [52], [53].

VIII. Conclusion

We proposed to use Verifiable Computing as a technique to reliably detect hardware faults in avionics computers as an alternative to ensuring integrity by computer replication. This allows for new system architectures overcoming some of the deficiencies of today's architectures. We carried out the first proof of concept demonstration that Verifiable Computing can be used for avionics applications such as simple control tasks. We provided a formal argument, why our construction ensures integrity and outlined a method for conservatively selecting an appropriate security parameter corresponding to a given maximum failure probability. Major challenges are the large overhead in computing time and the immaturity of the available tools, but current developments show promising results.

References

[1] A. Hills, "Digital fly-by-wire experience," in *AGARD Lecture Series No. 143*, 1985.

[2] A. Hills and N. Mirza, "Fault tolerant avionics," in *Digital Avionics Systems Conference*, 1988.

[3] J. H. Lala and R. E. Harper, "Architectural principles for safety-critical real-time applications," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 25–40, 1994.

[4] R. Hammett, "Design by extrapolation: An evaluation of fault tolerant avionics," *IEEE Aerospace and Electronic Systems Magazine*, vol. 17, no. 4, pp. 17–25, 2002.

[5] D. Brière and P. Traverse, "Airbus a320/a330/a340 electrical flight controls - a family of fault-tolerant systems," in *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE, 1993, pp. 616–623.

[6] E. Hitt, "Fault-tolerant avionics," in *Digital avionics handbook*, C. R. Spitzer, U. Ferrell, and T. Ferrell, Eds. CRC Press, 2017.

[7] D. J. Popp and R. L. Kahler, "C-17 flight control systems software design," in *Proceedings / IEEE/AIAA 11th Digital Avionics Systems Conference*. New York: IEEE, 1992, pp. 580–585.

[8] M. Walfish and A. J. Blumberg, "Verifying computations without reexecuting them," *Communications of the ACM*, vol. 58, no. 2, pp. 74–84, 2015.

[9] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish, "Making argument systems for outsourced computation practical (sometimes)," in *19th Annual Network and Distributed System Security Symposium*, 2012.

[10] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 238–252.

[11] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Advances in Cryptology - CRYPTO 2013*. Springer, Berlin, Heidelberg, 2013, pp. 90–108.

[12] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct non-interactive zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*. Berkeley, Calif.: USENIX Association, 2014, pp. 781–796.

[13] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 305–326.

[14] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[15] S. Ames, C. Hazay, Y. Ishai, and M. Venkitasubramaniam, "Ligero: Lightweight sublinear arguments without a trusted setup," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY: ACM, 2017, pp. 2087–2104.

[16] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[17] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[18] S. Bowe, J. Grigg, and D. Hopwood, "Recursive proof composition without a trusted setup," *Cryptology ePrint Archive*, 2019.

[19] E. Ben-Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 459–474.

[20] C. Sguanci, R. Spatafora, and A. M. Vergani, "Layer 2 blockchain scaling: a survey," *arXiv preprint*, 2021.

[21] J. Bonneau, I. Meckler, V. Rao, and E. Shapiro, "Mina: Decentralized cryptocurrency at scale." [Online]. Available: https://docs.minaprotocol.com/assets/technicalWhitepaper.pdf

[22] D. A. Luong and J. H. Park, "Privacy-preserving blockchain-based healthcare system for iot devices using zk-snark," *IEEE Access*, vol. 10, pp. 55 739–55 752, 2022.

[23] N. Huber, R. Kuesters, T. Krips, J. Liedtke, J. Mueller, D. Rausch, P. Reisert, and A. Vogt, "Kryvos: Publicly tally-hiding verifiable e-voting," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2022, pp. 1443–1457.

[24] T. Loruenser, F. Wohner, and S. Krenn, "A verifiable multiparty computation solver for the linear assignment problem," in *Proceedings of the 2022 on Cloud Computing Security Workshop*. New York, NY, USA: ACM, 2022, pp. 41–51.

[25] O. Patrick, "The coded microprocessor certification," *IFAC Proceedings Volumes*, vol. 25, no. 30, pp. 185–190, 1992.

[26] P. Forin, "Vital coded microprocessor principles and application for various transit systems," *IFAC Proceedings Volumes*, vol. 23, no. 2, pp. 79–84, 1990.

[27] U. Wappler and M. Müller, "Software protection mechanisms for dependable systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, 2008, pp. 947–952.

[28] M. Süßkraut, A. Schmitt, and J. Kaienburg, "Safe program execution with diversified encoding," in *Proceedings of the 13th embedded world conference*, 2015.

[29] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ACM Conferences. New York, NY: ACM, 2012, pp. 326–349.

[30] D. Hopwood, S. Bowe, T. Hornby, and N. Wilcox, "Zcash protocol specification: Version 2022.3.8 [nu5]." [Online]. Available: https://zips.z.cash/protocol/protocol.pdf

[31] B. Devos, "Loopring's zksnark prover optimizations - loopring protocol," *Loopring Protocol*, 01.03.2020. [Online]. Available: https://medium.loopring.io/zksnark-prover-optimizations-3e9a3e5578c0

[32] A. Kosba, Z. Zhao, A. Miller, Y. Qian, T.-H. H. Chan, C. Papamanthou, R. Pass, A. Shelat, and E. Shi, "C∅c∅: A framework for building composable zero-knowledge proofs," *Cryptology ePrint Archive*, 2015.

[33] C. Costello, C. Fournet, J. Howell, M. Kohlweiss, B. Kreuter, M. Naehrig, B. Parno, and S. Zahur, "Geppetto: Versatile verifiable computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015.

[34] M. Bellés-Muñoz, B. Whitehat, J. Baylina, V. Daza, and J. L. Muñoz-Tapia, "Twisted edwards elliptic curves for zero-knowledge circuits," *Mathematics*, vol. 9, no. 23, 2021.

[35] Y. El Housni and A. Guillevic, "Families of snark-friendly 2-chains of elliptic curves," in *Advances in Cryptology–EUROCRYPT 2022*. Springer, Cham, 2022, pp. 367–396.

[36] D. F. Aranha, Youssef El Housni, and Aurore Guillevic, "A survey of elliptic curves for proof systems," *Designs, Codes and Cryptography*, 2022.

[37] J. Hoffstein, J. Pipher, and J. H. Silverman, *An introduction to mathematical cryptography*, 2nd ed., ser. Undergraduate Texts in Mathematics. New York and Heidelberg and Dordrecht and London: Springer, 2014.

[38] A. Menezes, P. Sarkar, and S. Singh, "Challenges with assessing the impact of nfs advances on the security of pairing-based cryptography," in *Paradigms in Cryptology–Mycrypt 2016. Malicious and Exploratory Cryptology: Second International Conference*, 2016, pp. 83–108.

[39] T. Gaska, C. Watkin, and Y. Chen, "Integrated modular avionics - past, present, and future," *IEEE Aerospace and Electronic Systems Magazine*, vol. 30, no. 9, pp. 12–23, 2015.

[40] B. Witwer, "Systems integration of the 777 airplane information management system (aims)," *IEEE Aerospace and Electronic Systems Magazine*, vol. 11, no. 4, pp. 17–21, 1996.

[41] H. Butz, "Open integrated modular avionic (ima): State of the art and future development road map at airbus deutschland," in *Proceedings of the 1st International Workshop on Aircraft System Technologies*, O. von Estorff, Ed., 2007, pp. 211–222.

[42] D. L. Palumbo and R. W. Butler, "Measurement of sift operating system overhead."

[43] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, 1982.

[44] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems."

[45] M. Barborak, A. Dahbura, and M. Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, 1993.

[46] C. Favre, "Fly-by-wire for commercial aircraft: the airbus experience," *International Journal of Control*, vol. 59, no. 1, pp. 139–157, 1994.

[47] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Applications Conference. Proceedings*, vol. 1, 1996, pp. 293–307.

[48] J. Eberhardt and S. Tai, "Zokrates - scalable privacy-preserving off-chain computations," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 1084–1091.

[49] A. Kosba, C. Papamanthou, and E. Shi, "xjsnark: A framework for efficient verifiable computation," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 994–961.

[50] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A new hash function for zero-knowledge proof systems," in *Proceedings of the 30th USENIX Security Symposium*. Berkeley, CA: USENIX Association, 2021, pp. 519–535.

[51] E. Albert, M. Bellés-Muñoz, M. Isabel, C. Rodríguez-Núñez, and A. Rubio, "Distilling constraints in zero-knowledge protocols," in *Computer Aided Verification*. Cham: Springer International Publishing and Imprint Springer, 2022, pp. 430–443.

[52] J. Lee, S. Setty, J. Thaler, and R. S. Wahby, "Linear-time and post-quantum zero-knowledge snarks for r1cs," *Cryptology ePrint Archive*, 2021.

[53] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Advances in Cryptology - EUROCRYPT 2023*, 2023, pp. 499–530.