Chair of Distributed Systems & Operating Systems
Department of Computer Science // TUM School of Computation, Information and Technology
Technical University of Munich

Kernel Functions of a Trusted NIC Architecture

Guided Research

Robert Schambach ✉

Chair of Distributed Systems & Operating Systems,
Department of Computer Science // TUM School of Computation, Information and Technology,
Technical University of Munich

✉ scha@in.tum.de

June 5, 2023

Abstract — Due to the widespread use of untrusted third-party cloud computing infrastructure, service providers must account for individual compute nodes acting in arbitrary ways, or *Byzantine*. To this end, service providers employ *Byzantine Fault Tolerance* (BFT) protocols to detect Byzantine nodes. However, traditional BFT protocols require $3f + 1$ replicas to tolerate f Byzantine nodes, posing a large hurdle for in-field use.

Recently, trusted hardware-based BFT protocols emerged, which reduce the number of required replicas to $2f + 1$ by preventing nodes from making conflicting statements, or *equivocating*. Yet, these solutions neglect the *transferable authentication* of messages, which is necessary for the $2f + 1$ bound. Further, these solutions use trusted hardware outside of the network path, adding additional overhead.

To address these issues, we introduce kernel functions for a smart-NIC architecture, which address transferable authentication via asymmetric cryptography while being on the network path due to the kernels' placement on a smart-NIC. The kernel functions perform similar to existing software and FPGA-based solutions, while being moderately slower due to the use of asymmetric rather than symmetric cryptography.

1 Introduction

Due to the onset of cloud-computing, we increasingly depend on internet services. Hence, systems consisting of these interconnected services must retain high availability, even if individual services may fail. To this end, cloud service providers employ *Crash Fault Tolerant* (CFT) replication protocols. These protocols thereby exclusively handle crash-stop failures; the protocols expect services to only fail by crashing. In particular, such protocols assume services to not omit faulty behavior [1]. However, this assumption does not hold in practice. Software bugs, hardware errors, operator mistakes as well as malicious attacks often cause services to act in ways unintended by the service designer.

To this end, *Byzantine Fault Tolerance* (BFT) [2] protocols address services exhibiting arbitrary behavior, which we refer to as *Byzantine Faults*. Previously, such protocols were unsuitable for in-field use, and remained of theoretical interest. This incompatibility was due to the protocols' complexity [3], their poor performance, and critically, their number of required replicas. To illustrate, BFT protocols required $3f + 1$ replicas to tolerate f faults, for instance PBFT [4]. In contrast, CFT protocols only need $2f + 1$ replicas for f failures [1].

Recently, BFT protocols have emerged which ameliorate the need for a high amount of replicas by using trusted subsystems. For instance, such protocols use *Trusted Platform Module*s (TPMs), smart cards, and *Field-Programmable Gate Array*s (FPGAs) to reduce the required number of replicas to $2f + 1$ [5]–[7]. These protocols thereby use their subsystems to prevent Byzantine nodes from making conflicting statements, which we refer to as *equivocation*. However, preventing equivocation alone does not suffice to reduce the number of replicas to $2f + 1$. Such protocols also require the transferability of the authentication of messages [8]. Moreover, the developer using these BFT protocols must explicitly call the trusted subsystem to use the protocol. This non-transparency introduces additional complexity and makes room for bugs.

We address these aspects by introducing two FPGA kernel functions for attestation and validation, which we design to be integrated into a trusted *smart Network Interface Card* (smart-NIC) architecture for BFT. The functions thereby use asymmetric cryptography to ensure the transferability of authentication. Furthermore, as the functions may easily be integrated into a smart-NIC architecture, the functions prevent non-equivocation transparently. The kernel functions are thereby on the network path. The smart-NIC executes the functions, rendering them invisible to the program-

mer.Moreover, the use of the smart-NIC is independent of the host platform. As such, heterogeneous hosts may use the same trusted hardware subsystem while possessing different platforms.

Summarizing, the contributions of this work are as follows:

- We design and implement two FPGA kernel functions to prevent equivocation and enable transferability of authentication.
- We demonstrate how the kernel functions can be integrated into a trusted smart-NIC architecture to enable BFT.
- We evaluate the kernel functions, showing their overhead when we execute the functions on an FPGA.

2 Background

In this section, we give an overview of the theory behind BFT as well as the hardware which executes the kernels. We thereby cover equivocation, and present traditional as well as modern approaches to BFT. Moreover, we explain why non-equivocation is insufficient to enable $2f + 1$ replicas. Finally, we provide context for the use of FPGAs and smart-NICs for BFT.

2.1 Preventing Equivocation

A critical issue which BFT systems must counter is when nodes make conflicting statements, or *equivocate*. The canonical *Byzantine Generals Problem* [2] demonstrates the severity of equivocation. Given three parties attempting to reach a consensus, a single faulty party may prevent agreement, specifically because of equivocation.

Traditional BFT protocols, such as PBFT [4], counter such Byzantine behavior by acting as replicated state machines, thereby ensuring the linearizability of client requests. Hence, the system processes and responds to client requests in a consistent, total order. The system achieves this aspect via a designated replica leader, which proposes the order to execute received requests. For the leader may be Byzantine, the protocol necessitates the follower replicas to additionally synchronize their designated action with one another.

Modern protocols reduce the required number of replicas to $2f + 1$ by specifically preventing equivocation via trusted subsystems. Moreover, these subsystems remove the need for synchronization between the followers [5]–[7]. For instance, *Attested Append-Only Memory* (A2M) [5] uses trusted hardware to facilitate trusted logs for every protocol participant. These logs record the transmitted protocol messages. As trusted hardware stores and manages the logs, participants can trust the logs as well. Hence, followers can independently validate received messages, preventing equivocation of a Byzantine leader.

2.2 The Need for Transferable Authentication

As [8] demonstrates, non-equivocation alone is insufficient to reduce the number of required replicas to $2f + 1$. This result is due to the fact that non-transferable authentication allow Byzantine processes to violate the BFT protocol's *integrity*.

As we model processes in a BFT protocol as state machines, we define integrity as follows: if a correct process receives a message m, then m is a valid state machine message. However, without transferable authentication, a Byzantine process can send an invalid non-equivocated message to a correct process. Due to its faulty behavior, the Byzantine process thereby can send this message regardless of receiving correct inputs. The correct process can only validate the messages authenticity regarding its sender, and that the message does not conflict with another message sent by the Byzantine process. As such, a Byzantine process can send an invalid message to a correct receiver, which the receiving process perceives to be valid, thereby validating the integrity property [8].

In contrast, with transferable authentication, processes can prove the validity of their output messages. Such a process must thereby attach the transferable authentication tokens, for instance signatures, of the corresponding inputs which produced the output message. The receiving process must then verify the output by reproducing this result via the authenticated inputs. This verification produces a trust chain of authenticated messages, thereby ensuring the BFT protocol's integrity [8].

2.3 Trusted Hardware for BFT

Previous hybrid BFT protocols, such as [5]–[7], use trusted hardware to reduce the number of required replicas to $2f + 1$. These protocols reduce the required replicas by delegating mechanisms which prohibit equivocation to their respective trusted hardware

subsystem. Unlike [5], [6] yet similar to [7], an FPGA within a smart-NIC executes the kernel functions.

Having an FPGA execute the functions entails several advantages. One such convenience is a small *Trusted Computing Base* (TCB). As the FPGA implements the kernel function via configurable logic blocks, the implementation requires no Linux kernel or hypervisor [9]. Therefore, the absence of such lower-level software reduces the attack surface as well as the amount of code required to be trusted.

Further, to be compatible with a distributed systems sending thousands of messages per second, trusted hardware for BFT must have good performance. Hence, TPMs or smart cards are too slow, as these often have high latency for single operations. In contrast, FPGA functions are highly performant, due to the FPGA's ability to process large amounts of data in parallel as well as not needing to jump between the program and memory. Therefore, FPGAs are well suited to handle a high volume of incoming- and outgoing messages [9].

Besides having a small TCB, FPGAs serve well as a trusted subsystem as FPGAs execute separately from the BFT participant. Moreover, the participant is unable to read the memory of the FPGA. As such, we can program protected secrets into the FPGAs to establish trust between these subsystems in the BFT protocol [9].

Smart-NICs can use FPGAs to execute the kernel functions directly on the network path. Such smart-NICs consist of an on-board processor, a NIC, as well as an accelerator, such as in our case an FPGA. Hence, smart-NICs with a programmed FPGA can offer a transparent and trusted API for BFT operations. The BFT participant thereby does not have to manually attest and validate messages, as the participants offloads this work to the smart-NIC [10].

3 Design

In this section, we present the high-level overview of the kernels within the context of the trusted smart-NIC architecture. In particular, we first examine the encompassing smart-NIC structure and the role of the kernels in this composition. We then conclude by inspecting the attest and verify kernels conceptual function and by explaining how the kernels ensure BFT.

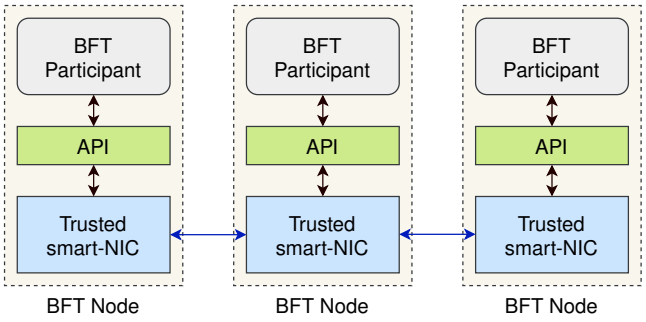

Figure 1 Trusted smart-NIC System Overview

3.1 Trusted smart-NIC Architecture

We show the overarching design of a trusted smart-NIC system in Figure 1. Therein, the figure presents three BFT nodes communicating via the smart-NIC. The host BFT participant thereby only communicates to the other participants via an API. Hence, the API hides the smart-NIC from the BFT participant. Further, the API relays messages sent by the participant to the trusted smart-NIC. In turn, the smart-NIC relays the message to other target smart-NICs. The NICs thereby use the kernel functions to ensure BFT.

We present an overview of the high-level architecture of the smart-NIC with the respective kernel functions in Figure 2. The conceptual smart-NIC design consists of four major components; the attest, verify, and network stack kernel, which comprise the FPGA functions, and the NIC. Of the FPGA functions, the smart-NIC uses the attest and verify kernels to ensure BFT. We show how these kernels function specifically in Section 3.2. Further, the network stack kernel ties the logic within the smart-NIC together. The network stack kernel integrates the logic of the attest and verify kernels into the specific BFT protocol.

In particular, the network stack kernel applies the attest kernel to outgoing messages. The BFT participant thereby sends the message via the API, which delegates the message to the network stack kernel. The network stack kernel then applies the attest kernel, and forwards the message as well as the attest kernel's output to the NIC for transmission. Moreover, the network stack kernel applies the verify kernel to incoming messages. The network stack kernel thereby reverses the outgoing process to pass the message back to the receiving BFT participant.

3.2 Attest & Verify Kernels

We present the attestation- and verify kernel functions in Figures 3, 4, respectively. The attestation kernel

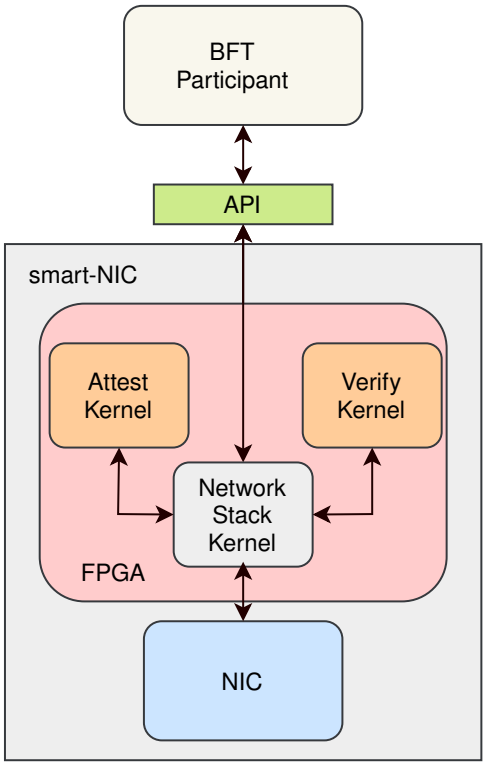

Figure 2 Trusted smart-NIC Architecture

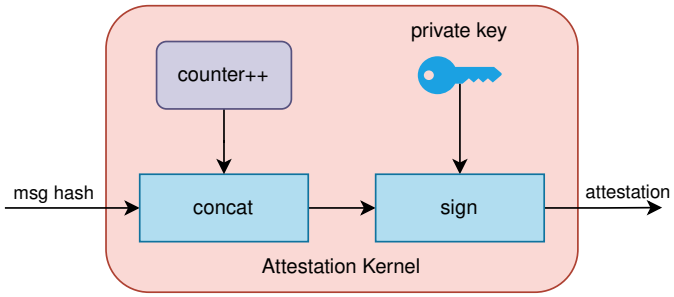

Figure 3 Attestation Kernel Overview

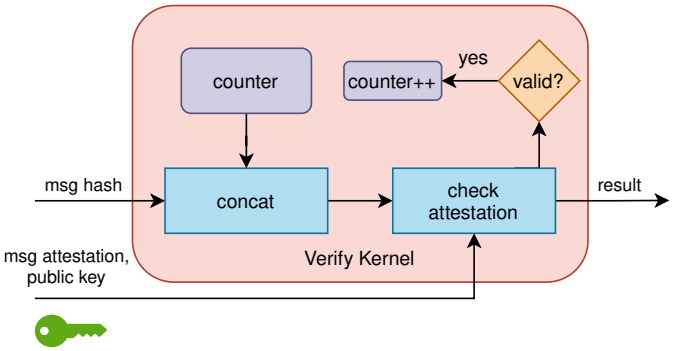

Figure 4 Verify Kernel Overview

thereby takes a message hash as input and produces an attestation using a private key. Further, the verify kernel receives a candidate message hash as well as a message attestation and public key, and produces an attestation result.

To enable BFT, we require non-equivocation and transferability of authentication, as we mention in Sections 2.1 and 2.2. As we show in this section, we prevent equivocation via an increment-only counter. The attest kernel binds the counter to the attestation and thereby increments the counter. As such, the attest kernel assigns each message an individual counter value. The verify kernel also possesses a counter, and increments the counter upon every successful verification. If a participant attempts to equivocate, then the attestation kernel binds a new counter value to the equivocation message. The verify kernel in turn detects the counter value mismatch, and returns a failed verification result.

We enable transferable authentication via the kernels' cryptography primitives. The kernels thereby use asymmetric cryptography, specifically, public and private keys. The kernels must exchange public keys beforehand, such that the verify kernels possess the public keys of all attestation kernels. Further, the kernels produce the attestations via signing functions; the attestations are signatures. As such, any verify kernel may validate a given message attestation via the corresponding public key. Hence, the kernels provide transferable authentication.

We now briefly explain how the kernels function, as we show in Figures 3 and 4. The attest kernel initially concatenates the input hash of the outgoing message to the internal counter value. The kernel then signs the concatenation with its private key, producing an attestation. After performing the signing operation, the kernel increments its counter value and returns the attestation.

Analogous to the attestation kernel, the verification kernel initially concatenates a candidate message hash with its internal counter value. The kernel then verifies the provided message attestation using the public key and the concatenation. If the attestation is valid, the kernel increments the counter. Finally, the verify kernel returns the verification result.

4 Implementation

This section presents the implementation of the smart-NIC kernels. In particular, we first state which languages and frameworks we use to create the kernels and how we implement the functions on which FPGA. Further, we inspect the kernels' cryptographic functions.

We implement the kernels in C++17 [11]. Further, we synthesize the code to the FPGA using the Xilinx Vitis software platform [12] version 2022.2 [13]. We execute these synthesized functions on an Alveo U280 Data Center Accelerator Card FPGA [14]. Moreover, the host executes the FPGA functions using the *Xilinx Runtime Library* (XRT) version 2022.2 [15].

To implement the cryptographic signing and verification operations, we use the Monocypher cryptography library version 4.0.1 [16]. We apply Monocypher due to its simplicity, small size, and cryptography algorithm choice. Moreover, the library's small size does not unnecessarily prolong the already drawn-out time required to synthesize the FPGA code. However, we nevertheless remove all of Monocypher's logic besides signing and verification to enable hardware synthesization. Finally, Monocypher uses the *Edwards-Curve Digital Signature Algorithm* (EdDSA) [17] for its public key signatures. Due to EdDSA's small key and signature sizes as well as good performance, EdDSA is well-suited for execution in a constrained and performance critical environment such as the FPGA [17].

5 Evaluation

This section discusses the effectiveness and performance of the kernel functions. Specifically, we present the experimental setup, including the metrics we use to evaluate the function execution. Moreover, we inspect the resulting implications as we contextualize the results via benchmarks on similiar BFT-enabling FPGA functions.

5.1 Experimental Setup

For the benchmark, we specifically measure the execution time of the kernel functions including the PCIe overhead as well as the XRT API call to the FPGA. To account for the overhead, we also measure the execution time of a kernel function without any logic, which we refer to as the empty kernel. As we mention in Section 4, we execute the functions on a Xilinx U280 FPGA [14].

We benchmark the kernel functions by averaging the time difference before and after the XRT API call to the FPGA. We thereby measure 20000 kernel executions and compute the corresponding average kernel execution time. Before we perform the measurements, we execute the kernel function beforehand to warm up the FPGA.

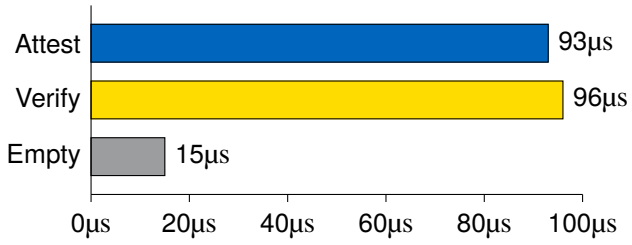

Figure 5 Kernel Function Benchmark

5.2 Benchmark Results

We present the results of the kernel function benchmark in Figure 5. The figure thereby shows the results of the attestation, verification, and empty kernel functions. As we explain before in this section, the empty kernel displays the FPGA kernel function call overhead. Hence, the difference between the attest as well as the verify function benchmarks and the empty kernel is the actual execution time of the kernel functions.

As the empty kernel benchmark in Figure 5 shows, the overhead accounts for a substantial amount of the execution time. This overhead is due to the time required to call the FPGA function as well as the PCIe transmission delay. Further, the benchmarks of the attest and verify kernels are fairly similar. This minimal difference is due to the functions' cryptographic procedure being fairly similar. In both procedures, the kernels compute a signature using a key, whereby the verify kernel validates the produced signature afterwards.

5.3 Benchmark in Context

As in [7], we contextualize the performance of the kernel functions of the smartNIC architecture via three subsystems which assign counter values to messages. All input message hashes are of size 32 B. We show this comparison in Figures 6 and 7 for the attestation and verification functions, respectively. The subsystems are as follows:

- **SoftLib**: the host authenticates and verifies messages via a software library. As the same process executes SoftLib which participates in a distributed system, the overhead is minimal. Further, as the BFT participant host itself executes this process, SoftLib does not provide BFT. The subsystem serves only for comparison. Further, [7] benchmarks SoftLib using an 8-core machine (2.3 GHz, 8 GB) [7].
- **Secure Sockets Layer** (SSL): the host executes an OpenSSL [18] server process. As this is process

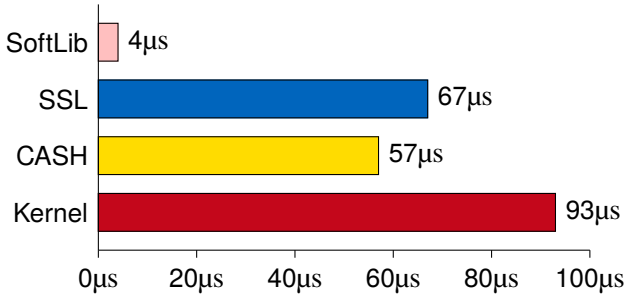

Figure 6 Subsystem Attest Benchmark Comparison [7]

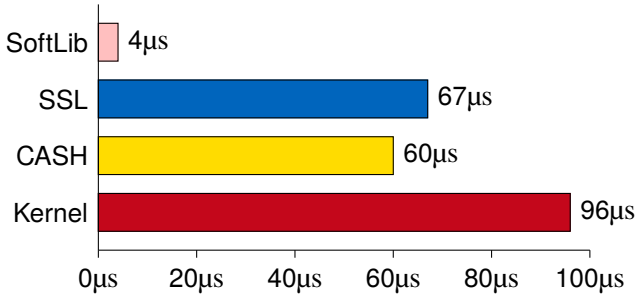

Figure 7 Subsystem Verify Benchmark Comparison [7]

is separate from the participating process, SSL incurs a larger overhead. Moreover, the same BFT-limts as well as benchmarking environment as in SoftLib apply.

- **Counter Assignment Service in Hardware** (CASH): an FPGA executes a trusted counter-assigning subsystem. This subsystem thereby uses symmetric rather than asymmetric encryption. As trusted hardware runs CASH, this subsystem also enables BFT. [7] runs CASH using a Xilinx Spartan-3 XC3S1500 FPGA.

To allow for comparability between the subsystems and the smartNIC kernel functions, all benchmarks execute on the same input message hash sizes of 32 B. Notably, subsystems themselves do not compute the message hash. We instead precompute this hash. Hence, for the attestation benchmark, the subsystems receive the input message hash, and produce an attestation. [7] measures the subsystems' functions as we do with our kernel functions, as we explain in Section 5.1. As such, for the attestation benchmark in Figure 6, the subsystems receive the message hash as input and produce an attestation. For the verification benchmark in Figure 7, the subsystems receive a candidate message hash, a corresponding attestation, and if applicable, a public key.

As [7] explains, the overhead for SSL is due to the inter-process communication via sockets. FreeLib does not have these overheads, and as such, is much faster. As CASH and our kernel functions require communicating with the FPGA, the functions also have a larger overhead than SoftLib. However, the FPGA subsystems do fall into the broad overhead range of SSL. Notably, CASH performs better than the smart-NIC functions. This speed difference is mainly due to the FPGA subsystem's cryptographic operations; CASH uses symmetric cryptography, while the cryptography of the smart-NIC functions is asymmetric. Asymmetric cryptography is known to be slower than symmetric cryptography, hence the performance difference. However, the asymmetric cryptography allows for the transferability of authentication, as we explain in Section 3.

6 Related Work

In this section, we discuss how this work relates to three BFT-enabling trusted subsystems.

A2M [5] implements a trusted log in external hardware enable protocol BFT. Similar to A2M, this work verifies messages by relying on a trusted hardware component. However, while A2M uses a trusted log to prevent equivocation, this work relies on a trusted counter.

More in-line to this work, TrInc [6] ensures non-equivocation by using a trusted counter on a smart card. While the trusted counter abstraction matches the technique we use in this work, we use an FPGA in a smart-NIC to process possibly large amounts of messages.

Even more closely related to this work, CheapBFT [7] operates via a trusted FPGA-based subsystem which uses the counter abstraction to enable BFT. While our work mostly aligns with CheapBFT, we design this work specifically in the context of a trusted smart-NIC architecture. Moreover, this work uses asymmetric encryption instead of CheapBFT's symmetric encryption to enable transferable authentication, as we explain in Section 2.

7 Conclusion

The FPGA kernel functions enable BFT with $2f + 1$ replicas within the context of a trusted NIC architecture. The functions thereby prevent equivocation and enable transferable authentication via trusted hardware and public-key cryptography, respectively. As the functions are within a smart-NIC, these kernel functions are part of the network path, reducing overheads

in the function's application. Further, the kernel functions perform similar to related subsystems. However, the functions execute moderately slower due to the use of asymmetric cryptography rather than symmetric cryptography.

References

[1] C. Delporte-Gallet, H. Fauconnier, F. C. Freiling, L. D. Penso, and A. Tielmann, "From crash-stop to permanent omission: Automatic transformation and weakest failure detectors," in *Distributed Computing*, A. Pelc, Ed., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2007, pp. 165–178, ISBN: 978-3-540-75142-7. DOI: 10.1007/978-3-540-75142-7_15.

[2] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, New York, NY, USA: Association for Computing Machinery, Oct. 4, 2019, pp. 203–226, ISBN: 978-1-4503-7270-1. [Online]. Available: https://doi.org/10.1145/3335772.3335936 (visited on 11/07/2022).

[3] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," *ACM Transactions on Computer Systems*, vol. 32, no. 4, 12:1–12:45, Jan. 20, 2015, ISSN: 0734-2071. DOI: 10.1145/2658994. [Online]. Available: https://doi.org/10.1145/2658994 (visited on 05/07/2023).

[4] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, pp. 398–461, Nov. 2002, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/571637.571640. [Online]. Available: https://dl.acm.org/doi/10.1145/571637.571640 (visited on 05/06/2023).

[5] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: Making adversaries stick to their word," p. 16,

[6] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," p. 14,

[7] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel, "CheapBFT: Resource-efficient byzantine fault tolerance," in *Proceedings of the 7th ACM european conference on Computer Systems*, Bern Switzerland: ACM, Apr. 10, 2012, pp. 295–308, ISBN: 978-1-4503-1223-3. DOI: 10.1145/2168836.2168866. [Online]. Available: https://dl.acm.org/doi/10.1145/2168836.2168866 (visited on 04/27/2023).

[8] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues, "On the (limited) power of non-equivocation," in *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, Madeira Portugal: ACM, Jul. 16, 2012, pp. 301–308, ISBN: 978-1-4503-1450-3. DOI: 10.1145/2332432.2332490. [Online]. Available: https://dl.acm.org/doi/10.1145/2332432.2332490 (visited on 04/25/2023).

[9] Advanced Micro Devices, Inc. "Programming an FPGA: An introduction to how it works," Xilinx. (May 7, 2023), [Online]. Available: https://www.xilinx.com/products/silicon-devices/resources/programming-an-fpga-an-introduction-to-how-it-works.html (visited on 05/08/2023).

[10] I. Synposys. "What is a SmartNIC? | data center network architecture," New Horizons for Chip Design. (Jul. 12, 2022), [Online]. Available: https://blogs.synopsys.com/from-silicon-to-software/2022/07/12/what-is-a-smartnic/ (visited on 05/08/2023).

[11] "C++17 - cppreference.com." (May 6, 2023), [Online]. Available: https://en.cppreference.com/w/cpp/17 (visited on 05/22/2023).

[12] "Vitis unified software platform documentation: Application acceleration development," 2022.

[13] "Downloads," Xilinx. (), [Online]. Available: https://www.xilinx.com/support/download.html (visited on 05/22/2023).

[14] "Alveo u280 data center accelerator card," Xilinx. (), [Online]. Available: https://www.xilinx.com/products/boards-and-kits/alveo/u280.html (visited on 05/22/2023).

[15] "Xilinx runtime library (XRT)," Xilinx. (), [Online]. Available: https://www.xilinx.com/products/design-tools/vitis/xrt.html (visited on 05/22/2023).

[16] "Monocypher." (Apr. 6, 2023), [Online]. Available: https://monocypher.org/ (visited on 05/22/2023).

[17] S. Josefsson and I. Liusvaara, “Edwards-curve digital signature algorithm (EdDSA),” Internet Engineering Task Force, Request for Comments RFC 8032, Jan. 2017, Num Pages: 60. DOI: 10.17487/RFC8032. [Online]. Available: https://datatracker.ietf.org/doc/rfc8032 (visited on 05/22/2023).

[18] The OpenSSL Project. “OpenSSL.” (), [Online]. Available: https://www.openssl.org/ (visited on 05/23/2023).