Towards Provable Security in Industrial Control Systems Via Dynamic Protocol Attestation

December 2024

Changing the World's Energy Future

Max Taylor, Lance Gregory Joneckis, Trevor Kann, Arthur Amorim

Idaho National Laboratory

Towards Provable Security in Industrial Control Systems Via Dynamic Protocol Attestation

Max Taylor, Lance Gregory Joneckis, Trevor Kann, Arthur Amorim

December 2024

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

Towards Provable Security in Industrial Control Systems Via Dynamic Protocol Attestation

Arthur Amorim
University of Central Florida
Florida, USA
arthur.amorim@ucf.edu

Trevor Kann
Carnegie Mellon University
Pittsburgh, USA
tkann@cmu.edu

Max Taylor
Idaho National Laboratory
Idaho Falls, USA
maxwell.taylor@inl.gov

Lance Joneckis
Idaho National Laboratory
Idaho Falls, USA
lance.joneckis@inl.gov

Abstract—**Industrial control systems (ICSs) increasingly rely on digital technologies vulnerable to cyber attacks. Cyber attackers can infiltrate ICSs and execute malicious actions. Individually, each action seems innocuous. But taken together, they cause the system to enter an unsafe state. These attacks have resulted in dramatic consequences such as physical damage, economic loss, and environmental catastrophes. This paper introduces a methodology that restricts actions using protocols. These protocols only allow safe actions to execute. Protocols are written in a domain specific language we have embedded in an interactive theorem prover (ITP). The ITP enables formal, machine-checked proofs to ensure protocols maintain safety properties. We use dynamic attestation to ensure ICSs conform to their protocol even if an adversary compromises a component. Since protocol conformance prevents unsafe actions, the previously mentioned cyber attacks become impossible. We demonstrate the effectiveness of our methodology using an example from the Fischertechnik Industry 4.0 platform. We measure dynamic attestation's impact on latency and throughput. Our approach is a starting point for studying how to combine formal methods and protocol design to thwart attacks intended to cripple ICSs.**

Index Terms—**Formal methods, Correctness proofs, Industrial automation, Industrial control, Software and System Safety.**

1. Introduction

Industrial control systems (ICSs) are large-scale, distributed, cyber-physical systems controlled by software. ICSs are increasingly targeted in cyber attacks. For example, the Stuxnet worm used malicious commands to destroy centrifuges [36]. In 2014, a similar attack caused immense damage to a blast furnace in a steel mill [29]. Blast furnaces must be shut down according to a specific protocol or risk damage. Attackers infiltrated the steel mill's ICSs and violated the shutdown protocol.

Figure 1 shows an attack against an ICS controlling a high bay warehouse storing red, white, and blue items. This ICS contains a human machine interface (HMI), a control unit, and low-level device drivers. These components interact by executing remote procedure calls (RPCs). At the
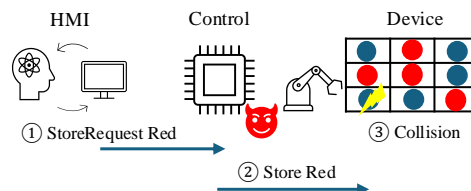

Figure 1: Example attack. An adversary infiltrates the control unit to cause a collision in the warehouse.

start of this attack, the warehouse is full. Then, the operator executes a **StoreRequest** RPC. Alone, this RPC is benign: the controller is under no obligation to store an item in a full warehouse. But in this controller resides an attacker who previously infiltrated the ICS. The attacker seizes their opportunity to execute a **Store** RPC on the hardware drivers. Since there is no empty bay available, a collision occurs.

An ICS's software components can become compromised at any moment. Compromises are increasing with the rise of "Industry 4.0." Industry 4.0 integrates manufacturing and digital technology [22]. Often, digital technology uses the public internet. Presence on the internet exposes new and existing vulnerabilities to attackers. Attackers exploit vulnerabilities to compromise ICS components. Compromised components can execute arbitrary RPCs. Hence, we assume an attacker can execute arbitrary, malicious RPCs. Malicious RPCs can lead to unsafe behavior, as in Figure 1.

It is possible, albeit challenging, to show an ICS behaves safely using formal methods. Formal methods are a family of mathematical techniques for reasoning about systems. Formal methods have two popular techniques: model checking and theorem proving. Model checking has been applied in previous research efforts to obtain safety poofs for ICSs [31], [35], [46], [27], [12]. However, model checking suffers from the state explosion problem. This problem prevents model checking from scaling to larger ICSs, as shown in [12].

Scaling safety proofs to larger ICSs is possible using theorem proving. However, theorem proving notoriously requires significant effort. For example, it cost 20 person-years to verify seL4, an OS written in $\approx$ 8000 lines of C [24]. But

this effort was only necessary because the totality of seL4 was verified. All other factors remaining equal, verifying less software requires less effort.

Our main idea is to restrict sequences of RPCs to a protocol. We prove protocol conformance implies the ICS behaves safely. Thus, unsafe behavior can only ever occur as a result of executing a non-conformant RPC. We keep non-conformant RPCs from executing using a dynamic protocol attestation system. When the attestation system encounters a non-conformant RPC, it triggers a fail-safe. Fail-safes are always safe to execute.

In our approach, engineers describe their system's RPCs using an interactive theorem prover (ITP). ITPs are programs that help engineers write formal, machine-checked proofs. This paper uses F* [38], a state-of-the-art ITP with sophisticated proof automation. Inside F*, we have embedded a domain specific language (DSL). Our DSL helps engineers design RPC protocols. Engineers use F*'s proof automation to prove that protocol conformance implies safety.

We provide automation to generate code for RPC frameworks from our DSL. Although ITPs like F* can help verify designs, they are not suitable for writing ICS software. ICS software is often written in embedded languages like C or C++. Embedded languages are usually not memory or type safe. This lack of safety makes RPC implementations more vulnerable to attacks. We can reduce the risks posed by this attack vector by leveraging off-the-shelf RPC frameworks. RPC frameworks use high-level message specifications to generate "boilerplate" code. With boilerplate code automatically generated, software engineers can concentrate on essential application logic. Our automation extracts message specifications for RPC frameworks directly from an engineer's F* specification.

Critically, our approach makes formally verifying all software in an ICS unnecessary. Instead, we only need to verify software that interfaces with physical mechanisms. For example, the low-level software that drives a servo or reads a thermometer. It is easy to identify these software components. In our formal model, they directly interact with the system state. Our proofs show that the ICS behaves safely if RPC semantics are accurately modeled; i.e., if the software implementation matches the F* semantics.

To summarize our contributions, we:

- **Propose a new methodology for developing ICS software.** Our methodology enables us to construct new communication protocols in F*, a state-of-the-art ITP. We use F* to prove that protocols maintain critical safety invariants.
- **Provide automation to extract RPC framework inputs from F*.** This generates the code necessary to create implementations of the protocol.
- **Show how to use dynamic protocol attestation to reduce the verification burden.**
- **Provide experimental results that highlight the potential of our approach.** Dynamic attestation checking adds moderate overhead. In many applica-

tions, the overhead will be acceptable. Future work will consider how to further reduce the overhead.

2. Background

2.1. Fischertechnik

2.1.1. Overview. Industry and academia use Fischertechnik's embedded platforms [13] to address the challenges posed by Industry 4.0 [22]. Industry 4.0 increases the complexity and concurrency of ICS software. These increases have lead to vulnerabilities not previously found in embedded systems. Members of industry and academia are mitigating these vulnerabilities with new technical solutions, often validated on Fischertechnik's platforms.

The Fischertechnik was initially used in studies involving controller verification with model checking [8], [7], [15], [7], in an example implementation of model-driven systems engineering [43], [3], and in a program analysis-based PLC code vetting study [44].

Researchers have also used Fischertechnik platforms to study ICS security. For example, researchers used Fischertechnik platforms to study runtime detection of improper parameters in ICSs [44]. They were also used to study decentralized monitoring of linear temporal logic (LTL) formulae [1]. Finally, a contract-based hierarchical resilience management framework which improves resilience when dealing with component failures used a Fischertechnik platform to perform its evaluation [17].

2.1.2. High Bay Warehouse. The Fischertechnik factory stores manufactured items in its high bay warehouse (HBW). The HBW is shown in Figure 2. The HBW provides an interface that displays items to customers. The HBW consists of nine bays that can store at most one item. The HBW interacts closely with the vacuum gripper. The vacuum gripper delivers and retrieves items from storage. The HBW also has an arm that moves items into and out of bays. We chose to implement our approach on the HBW due to its intuitive functionality and straightforward interactions with other components.
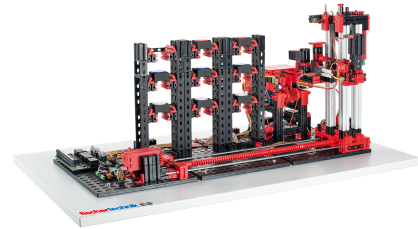

Figure 2: Image of the Fischertechnik's High Bay Warehouse from [14]

2.1.3. Critical Safety Properties. The high bay warehouse is vulnerable to two dangerous scenarios:

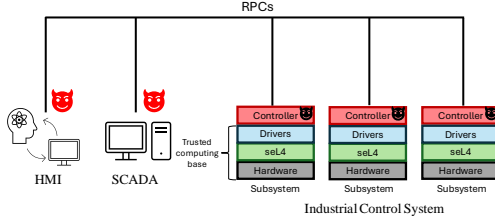1) Storing a new item when the high bay warehouse is already full.

Figure 3: Overview of an ICS. The demon icon is located adjacent to components that we assume an attacker controls.

   2)   Attempting to retrieve an item that is not stored in a bay.

Therefore, an ICS's software implementation must enforce three invariants. First, that the HBW is not full before it attempts to store an item. Second, that the HBW only processes a customer's order if it contains the desired item. Third, that the state of the bays is accurately tracked. In this example the safety property is that none of these invariants are violated.

2.2. Attack Model

2.2.1. Threats. We consider a threat model where adversaries seek to compromise ICS controllers. ICS controllers use low-level hardware drivers to manipulate the physical world. ICS controllers support emerging industry 4.0 trends by interfacing with the public internet. Interfacing with the internet jeopardizes the security of an ICS controller. Therefore, we assume an ICS controller may become compromised at any moment. Compromised ICS controllers may execute RPCs that place the ICS at-risk.

2.2.2. Attacker. We assume the attacker is able to execute arbitrary RPCs on any controller in the ICS. Figure 3 shows possible attack vectors. ICS controllers communicate with low-level device drivers. Thus, the attacker is able to execute malicious actions from compromised controllers. The attacker may also execute malicious RPCs on uncompromised controllers connect by a network. By trusting input from malicious RPCs, uncompromised controllers can also take a system to an unsafe state.

2.2.3. Trusted Computing Base. We assume attackers cannot control the operating system, low-level hardware drivers, or hardware. High assurance operating systems like seL4 [41] have formally verified security properties. These properties enable strong isolation of software components. We assume that engineers have verified the behavior of hardware drivers. We trust that hardware functions as intended. We also assume attackers have not compromised the RPC framework. We further assume the RPC framework always executes RPCs. Finally, we assume attackers have not compromised our dynamic protocol checker.

2.3. Interactive Theorem Provers

Interactive theorem provers are tools that help engineers construct computer-verified mathematical proofs. Engineers write proofs in a language the ITP provides. Often, ITPs provide purely functional languages based on the Calculus of Inductive Constructions [34]. The Calculus of Inductive Constructions (CIC) is a type theory for expressing proofs. Proofs in CIC are constructed following the Curry-Howard isomorphism [21]. The Curry-Howard isomorphism treats types as propositions and values as proofs. Writing a proof of a proposition amounts to writing a term whose type corresponds to the proposition. Type-checking verifies that the proof is free from mistakes in reasoning.

ITPs provide automation that helps users write terms. Writing a term that inhabits a type is often awkward. As the type's corresponding proposition grows more complex, the situation becomes worse. To help remedy the situation, ITPs often provide tactic systems. Tactics are examples of metaprograms: programs that write programs. Tactics construct terms by exposing commands to engineers. These commands more closely match how humans think about proofs. Commands need not even be correct because the term a command produces is ultimately type-checked. Users can extend ITPs with their own tactics without compromising the soundness of their proofs.

F* [38] is a new ITP that makes it even easier for engineers to write proofs. Tactic systems require users to explicitly justify each step in their reasoning. But many steps in reasoning are obvious: equality is symmetric, addition is commutative, etc. Often times, humans make these steps without ever saying so. F* mimics human reasoning by integrating the Z3 SMT solver with its tactic system [10]. SMT solvers are tools that automatically prove propositions. Since it is impossible to automatically prove all propositions, SMT solvers may fail. When Z3 succeeds, an engineer is relieved of the burden. When Z3 fails, the engineer can use a combination of programming and tactics to write the proof. Z3 still fills gaps between steps, thus tremendously reducing the proof burden.

3. Methodology

This section describes how to construct protocol specifications and prove that safety properties hold on the protocol. Communication protocols describe legal sequences of RPCs. Ultimately, a communication protocol is a formal language inhabited by sequences of RPCs called *traces*. RPCs have semantics that our methodology formally models in F*. In the semantics, an RPC that causes the ICS to behave unsafely causes the state to become **Wrong**. We embed a DSL for describing communication protocols in F*. Using F*, we can verify conformance to a protocol implies the ICS does not go **Wrong**.

An attacker can cause traces of the ICS to diverge from the specified protocol. To account for this possibility, we implement dynamic protocol attestation. Dynamic protocol

attestation checks that the ICS's trace conforms to the protocol. Dynamic protocol attestation prevents non-conforming RPCs from taking the system to a **Wrong** state.

3.1. DSL Design

We describe communication protocols using a DSL embedded in F*. Our methodology provides a general purpose protocol description language. However, the protocol-specific parameters listed in Table 1 must be provided by the engineer.

Parameter	Description
$\mathbb{D}(V_m)$	Values used by $State$ and $ExtCom$
AP	Propositions that reflect the ICS's state
$ExtCom$	RPCs executed in the ICS
Inputs	Commands received by the HMI

TABLE 1: Description of protocol-specific parameters.

3.1.1. Syntax. Our DSL draws a distinction between RPCs and terms that describe protocol behavior. In our DSL, RPCs are called *external commands*. Terms that describe protocol behavior are called *internal commands*.

$$v_{0,j} \in \mathbb{D}(V_0)$$
$$\vdots \in \ldots$$
$$v_{m,j} \in \mathbb{D}(V_m)$$
$$ap \in AP$$
$$state \in State$$
$$EC \in ExtCom$$
$$IC \in IntCom$$

$$
\begin{aligned}
V_0 \quad &::= \quad v_{0,0} \mid \ldots \mid v_{0,n} \\
\vdots \quad &::= \quad \ldots \\
V_m \quad &::= \quad v_{m,0} \mid \ldots \mid v_{m,n} \\
state \quad &::= \quad \sigma \mid \textbf{Wrong} \\
EC \quad &::= \quad E_1\ v_{m_1,n_1}\ \ldots\ v_{m_k,n_k} \mid \ldots \mid \\
&\qquad E_j\ v_{m_1,n_1}\ \ldots\ v_{m_k,n_k} \\
IC \quad &::= \quad \textbf{Skip} \mid \textbf{Seq}\ IC_1\ IC_2 \mid \textbf{If}\ ap\ IC_1\ IC_2 \mid EC
\end{aligned}
$$

$ExtCom$ is the set of all external commands exchanged in the system. Commands are represented as $E_j\ v_{m_1,n_1}\ \ldots\ v_{m_k,n_k}$. Each v_{m_k,n_k} represents a value of the command's corresponding argument. A value is part of a domain $\mathbb{D}(V_m)$. Domains can be standardized (e.g., $\mathbb{N}$, bool) or user-defined. A trace is a sequence of external commands. External commands are the only mechanism to modify the ICS's state. Therefore, to assure the safety of the system is to enforce that only traces that satisfy safety guarantees are executed. In our methodology, this can be accomplished by defining a provably secure RPC protocol. Atomic propositions, AP, are used to reason about the ICS's internal state. The labeling function L: State $\rightarrow 2^{AP}$ maps a state to a set of atomic propositions. An atomic proposition p evaluates to true in state σ iff $p \in L(\sigma)$.

States can be *valid* or *invalid*. A valid state is a tuple $\sigma = \langle s1,\ s2,\ \ldots,\ s_k,\ input \rangle$; σ contains k internal variables that comprise the internal state. A valid state σ also keeps track of SCADA and HMI input. Although not part of the syntax, inputs are also protocol-specific parameters. An invalid state is labeleled as a **Wrong** state. Non-conformant RPCs take the semantics of the ICS to **Wrong**. Engineers define a protocol using internal commands. We emphasize the **If** internal command, as it represents the control flow for creating safe protocol specifications. **If** commands are conditioned by atomic propositions.

3.1.2. Semantics. The semantics in Figure 4 formalize the ICS's behavior. A command that takes a system to a valid state is a *valid* command. Each external command has two semantic rules. The first rule states the effects of performing the operations denoted by a valid command. The second rule states which commands are *invalid*. For example, the **Store** command is invalid when the control unit executes a **Store** command when the high bay warehouse is already full. Invalid commands lead the system to a **Wrong** state. The big-step semantics for external commands takes a state and a command and returns a new state.

$$\langle \text{ExtCom},\ \text{State} \rangle \Rightarrow_{EC} \text{State}$$

The semantics for internal commands also uses a big-step semantics. In the absence of recursion, all command-state pairs evaluate to a state without the need to consider intermediate steps.

$$\langle \text{IntCom},\ \text{State} \rangle \Rightarrow_{IC} \text{State}$$

In order to define a protocol specification for a given system, the user must add the system's external commands to the DSL. Furthermore, the user must define their underlying semantics. There are two functions that achieve these goals.

The semantic function $\mathcal{E}$: **ExtCom** $\rightarrow$ **State** $\rightarrow$ **State** implements $\Rightarrow_{EC}$ for all combinations of external commands and states. This function allows the user to define the semantics for all RPCs in the system. First, the semantics allows the system to define which $State \times ExtCom$ combinations are valid. Second, the semantics formalize how executing a given command changes the internal state. Third, the exhaustive property of F* functions forces users to consider all possible scenarios. However, some of the control flow may require procedures to check properties of the internal state.

The precondition function Φ : ExtCom $\rightarrow$ State $\rightarrow$ bool determines if a command is valid in a valid state. Commands that are not valid evaluate to **Wrong**. We note that Φ and atomic propositions are strictly orthogonal ideas. Φ is used to define the semantics of commands. Atomic propositions are used to define protocols.

3.1.3. Safety Condition. In an unrestricted environment, both valid and invalid sequences of RPCs can be executed. To generate only valid RPCs, we restrict accepted traces to a

$$\frac{\Phi(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \sigma) \quad \frac{\Phi(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \sigma)}{(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \sigma) \Rightarrow_{EC} \sigma'}}{(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \sigma) \Rightarrow_{EC} \sigma'} \text{(ExtCom Valid)} \qquad \frac{}{(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \mathbf{Wrong}) \Rightarrow_{EC} \mathbf{Wrong}} \text{(Wrong)}$$

$$\frac{\neg\Phi(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \sigma)}{(E_j v_{m_1,n_1} \ldots v_{m_k,n_k}, \sigma) \Rightarrow_{EC} \mathbf{Wrong}} \text{(ExtCom Invalid)} \qquad \frac{ap \in L(s) \quad (Ic1, s) \Rightarrow_{IC} s'}{(\mathbf{If}\ ap\ IC_1\ IC_2, s) \Rightarrow_{IC} s'} \text{(If true)} \qquad \frac{}{(\mathbf{Skip}, s) \Rightarrow_{IC} s} \text{(Skip)}$$

$$\frac{ap \notin L(s) \quad (IC_2, s) \Rightarrow_{IC} s''}{(\mathbf{If}\ ap\ IC_1\ IC_2, \sigma) \Rightarrow_{IC} s''} \text{(If false)} \qquad \frac{(Ec1, s) \Rightarrow_{IC} \mathbf{Wrong}}{(\mathbf{Seq}\ IC_1\ IC_2, s) \Rightarrow_{IC} \mathbf{Wrong}} \text{(Seq Invalid)} \qquad \frac{(IC_1, \sigma) \Rightarrow_{IC} \sigma' \vee (IC_2, \sigma) \Rightarrow_{IC} \sigma''}{(\mathbf{Seq}\ IC_1\ IC_2, \sigma) \Rightarrow_{IC} \sigma''} \text{(Seq Valid)}$$

Figure 4: Semantics of our DSL.

Framework	Overhead	Embedded Language Support			Custom Networking?	Security Practices	
		C?	C++?	Rust?		Code Scanning?	Fuzzing?
Cap'n Proto [6]	Low	✗	✓	✓	✗	✗	✓
COAP [37]	High	✓	✓	✓	✗	✗	✗
eRPC [11]	Low	✓	✗	✗	✓	✗	✗
gRPC [16]	Medium	✗	✓	✓	✗	✗	✓

TABLE 2: RPC frameworks we considered adopting in our prototype.

subset of all possible traces. Our DSL introduces syntax for restricting RPCs, given underlying semantics of the RPCs. Using the DSL, users write a term (i.e., called *spec*) that specifies a communication protocol. F* can be used to prove different safety properties about the protocol. Given the attack model, the following safety condition must be proven:

SAFETY CONDITION

Applying the big-step semantics $\Rightarrow_{IC}$ to *spec* in a state $\neq$ **Wrong** always yields a state' $\neq$ **Wrong**.

Formally:

$$\forall \sigma \in \text{State}.\langle spec, \sigma \rangle \not\Rightarrow_{IC} \mathbf{Wrong}$$

If this condition is proven, then all sequences of commands generated by the *spec* are safe command sequences. Thus, *spec* generates a subset of all RPCs that cannot go **Wrong**.

3.2. Translation to RPC Framework

We provide automation that translates terms from F* into an RPC framework. The translation occurs via procedures written using Meta-F* [28]. Meta-F* is a framework for metaprogramming in F*. Metaprogramming involves writing procedures that write programs. Metaprogramming allows us to inspect terms in F* and emit new terms that generate code in an RPC framework. This allows us to use F* specifications to generate code for an RPC framework.

3.2.1. RPC Frameworks. Table 2 shows the RPC frameworks we considered adopting in our prototype. The "Custom Networking" column indicates if the framework supports additional network technologies. There are a large number of network technologies found in embedded systems. For example, an embedded system might use CAN, UART, SPI, and ethernet to connect its subsystems. Ideally,

an embedded framework allows for extending its network layer to support these protocols.

RPC frameworks introduce overhead. Overhead originates from serializing and deserializing data. We classified overhead as either *Low*, *Medium*, or *High*. Cap'n Proto and eRPC are low-overhead RPC frameworks. They use binary representations of data that matches how a compiler arranges data structures. This reduces overhead while serializing and deserializing data. We consider COAP to have high overhead. While COAP itself is light-weight, it is not a true RPC framework. Creating an RPC framework in COAP introduces additional overhead beyond the other RPC frameworks.

We evaluated the security practices of candidate RPC frameworks with two criteria. First, *does the project use any static analysis tools?* Static analysis tools like Coverity [9] can identify vulnerabilities during development [2]. Second, we considered *does the project use fuzzing?* Fuzzing is a software testing strategy that uses random inputs to try to find unexpected outputs or invariant violations [30]. Fuzzing has identified thousands of defects in software [26], [47]. The "Security Practices" columns in Table 2 reports our findings. gRPC performed static analysis using Coverity, but stopped in 2015. Both gRPC and Cap'n Proto use fuzzing. Fuzzing is effective, having found a critical security vulnerability in Cap'n Proto in 2022 [5].

We chose to initially implement support for the Cap'n Proto RPC framework based on our evaluation. Cap'n Proto is lightweight, supports many embedded languages, and has good security practices. Although Cap'n Proto does not support C, C and C++ have good interoperability.

In the future, we plan to support additional RPC frameworks besides Cap'n Proto. Cap'n Proto lacks the ability to use common embedded networking technologies. We will eventually fill this gap using eRPC. In the long term, we plan to create our own RPC framework. We will use formal methods to prove its security.

3.3. Dynamic Protocol Attestation

StoreRequest Not_Full Store(blue)
StoreRequest Not_Full Store(white)
StoreRequest Not_Full Store(white)
RetrieveRequest(white) No_Color Retrieve(white)
RetrieveRequest(white) No_Color Retrieve(white)
RetrieveRequest(blue) No_Color Retrieve(blue)
StoreRequest Not_Full Store(white)
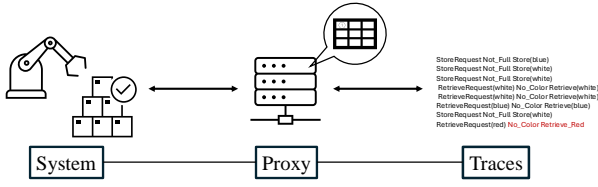RetrieveRequest(red) No_Color Retrieve_Red

Figure 5: Dynamic protocol attestation.

We make no assumptions about the correctness of components implementing the RPC protocol beyond the low-level device drivers. Component defects may produce invalid traces. Thus, we introduce dynamic protocol attestation to ensure the protocol is followed at runtime. Consequently, it provides assurance that the system does not reach an unsafe state. To prevent illegal runtime traces from causing the ICS to go wrong, we must decide if the current trace is generated by *spec*. A decider procedure acts as a recognizer for the safe RPC language. The decider takes the ongoing system trace, *spec*, and current state as an input. As output, the decider returns true if *spec* generates the trace and false otherwise. This procedure is implemented as a proxy. The proxy monitors ICS traces at runtime, as shown in Figure 5. For performance reasons, we implemented the proxy by hand. But, it can be automatically generated. This automation is left for future work.

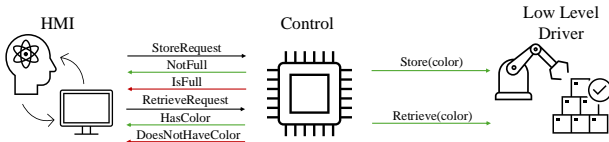## 4. Applying Methodology to Fischertechnik

Figure 6: The warehouse's communication protocol.

To demonstrate our metholodology, we use the Fischertechnik factory as shown in Figure 6. A customer can request an item of a certain color via a **RetrieveRequest** *color* RPC. Likewise, a supplier can request to store an item using the **StoreRequest** *color* RPC. The HBW controller receives and processes these RPCs. The effect of RPCs is contingent on the current internal state of the system. The controller can either return a rejection or an acknowledgement to the sender. The **RetrieveRequest** *color* RPC triggers either a **HasColor** or a **DoesNotHaveColor** response. The **StoreRequest** *color* RPC can trigger a **IsFull** or **NotFull** response. The controller then forwards the accepted requests to the low-level hardware drivers. After processing an RPC, the controller updates the current dynamic state of the the bays.

The controller makes all policy decisions. The low-level drivers faithfully execute all RPCs they receive from the controller. The safety of this communication is critically important. An attacker is able to execute arbitrary RPCs. Thus, an attack stemming from the controller could trigger unsafe driver commands. These unsafe commands pose a physical threat to workers, and even the ICS itself. Since the protocol describes all safe executions, nonconforming traces can cause unsafe behavior. Non-conformant traces that lead to unsafe behavior are *invalid traces*. A invalid trace will break at least one logical assertions defined in §2.1.3. This can lead to outcomes like collisions when attempting to store an item in an occupied bay.

4.1. Syntax and Semantics

The protocol-specific syntax is:

$$
\begin{array}{lll}
color & ::= & \textbf{empty} \mid \textbf{red} \mid \textbf{white} \mid \textbf{blue} \\
input & ::= & \textbf{RetrieveRequest } color \mid \textbf{StoreRequest } color \\
EC & ::= & \textbf{IsFull} \mid \textbf{NotFull} \mid \textbf{HasColor} \mid \textbf{Store } color \\
 & & \mid \textbf{Retrieve } color \mid \textbf{DoesNotHaveColor} \\
ap & ::= & \textbf{APBaysFull} \mid \textbf{APHasRetrieveRequest} \\
 & & \mid \textbf{APContainsWhite} \mid \textbf{APContainsBlue} \\
 & & \mid \textbf{APHasStoreRequest} \mid \textbf{APRequestRed} \\
 & & \mid \textbf{APContainsRed} \mid \textbf{APRequestBlue} \\
 & & \mid \textbf{APRequestWhite}
\end{array}
$$

Some external commands take *color* as arguments. We represent a valid state as a pair $\sigma = \langle \text{bays, input} \rangle$; σ keeps track of the current state of the nine bays and the current user input. We model the bays as a record of nine slots:

```
type bays = { bay1 : color; ... ; bay9 : color; }
```

To access each bay, we define a function:

```
bf : bays → ℕ → color
```

that takes the bays and a number n between 1 and 9 as arguments, and returns the current state of bay n.

EC is the set of valid commands that can be sent in the network. Figure 6 demonstrates the intended communication protocol. The example in appendix §A.4 shows how a trace is broken down into small sequences of commands that can be interpreted by the decider. Each part is evaluated by the dynamic attestation checker against the current state. The example contains only valid commands that take the HBW from a valid state to another valid state in accordance to the HMI input. On the other hand, §A.5 illustrates a few patterns of invalid traces. These traces contain at least one invalid command, with takes the ICS's state to **Wrong**. The invalid command either violates the HBW's safety properties listed in §2.1.3 or does not correspond to the intended action given by the HMI input.

Using our DSL, we write a term called *spec* (which is shown in Figure 8) that specifies the RPC protocol for the high bay warehouse.

Next, we must prove that following the protocol described by *spec* ensures that the safety condition holds. For

$$\frac{\text{find color}(s,1,\text{empty}) = \text{None}}{(\textbf{IsFull}, s) \Rightarrow_{EC} s}\text{(IsFull Valid)} \quad \frac{\text{find color}(s,1,\text{empty}) = \text{Some}}{(\textbf{IsFull}, s) \Rightarrow_{EC} \textbf{Wrong}}\text{(IsFull Invalid)} \quad \frac{\text{find color}(s,1,\text{empty}) = \text{Some}}{(\textbf{NotFull}, s) \Rightarrow_{EC} s}\text{(NotFull Valid)}$$

$$\frac{\text{find color}(s,1,\text{empty}) = \text{None}}{(\textbf{NotFull}, s) \Rightarrow_{EC} \textbf{Wrong}}\text{(NotFull Invalid)} \quad \frac{\text{find color}(\sigma,1,\text{empty}) = \text{None}}{(\textbf{Store } color, \sigma) \Rightarrow_{EC} \textbf{Wrong}}\text{(Store color) Invalid)} \quad \frac{\text{find color}(\sigma,1,color) = \text{None}}{(\textbf{Retrieve } color, \sigma) \Rightarrow_{EC} \textbf{Wrong}}\text{(Retrieve Color Invalid)}$$

$$\frac{\sigma.\text{inp} = \textbf{RetrieveRequest } color \quad \text{find color}(\sigma,1,c) = \text{None}}{(\textbf{DoesNotHaveColor}, \sigma) \Rightarrow_{EC} \sigma}\text{(DNHC Valid)} \quad \frac{\sigma.\text{inp} = \textbf{RetrieveRequest } color \quad \text{find color}(\sigma,1,c) = \text{Some}}{(\textbf{HasColor}, \sigma) \Rightarrow_{EC} \sigma}\text{(HasColor Valid)}$$

$$\frac{\sigma.\text{inp} = \textbf{RetrieveRequest } color \quad \text{find color}(\sigma,1,c) = \text{Some}}{(\textbf{DoesNotHaveColor}, \sigma) \Rightarrow_{EC} \textbf{Wrong}}\text{(DNHC Invalid)} \quad \frac{\sigma.\text{inp} = \textbf{RetrieveRequest } color \quad \text{find color}(\sigma,1,c) = \text{None}}{(\textbf{HasColor}, \sigma) \Rightarrow_{EC} \textbf{Wrong}}\text{(HasColor Invalid)}$$

$$\frac{\text{find color}(\sigma,1,\text{empty}) = \text{Some}}{(\textbf{Store } color, \sigma) \Rightarrow_{EC} \sigma[color \mapsto \text{bf}(\sigma.\text{bays}, n)]}\text{(Store color Valid)} \quad \frac{\text{find color}(\sigma,1,color) = \text{Some } n}{(\textbf{Retrieve } color, \sigma) \Rightarrow_{EC} \sigma[\text{empty} \mapsto \text{bf}(\sigma.\text{bays}, n)]}\text{(Retrieve color Valid)}$$

Figure 7: Semantics of our command language.

```
1  spec =
2  If APHasStoreRequest
3   (If APBaysFull (IsFull)
4    (Seq
5      (NotFull)
6      (If APRequestRed (Store Red)
7        (If APRequestWhite (Store White)
8          (If APRequestBlue (Store Blue)
9            Skip)))))
10  (If APHasRetrieveRequest
11    (If APRequestRed
12      (If APContainsRed
13        (Seq (HasColor) (Retrieve Red)) (NoColor))
14      (If APRequestWhite
15        (If APContainsWhite
16          (Seq (HasColor) (Retrieve White))
17          (NoColor))
18        (If APRequestBlue
19          (If APContainsBlue
20            (Seq (HasColor) (Retrieve Blue))
21            (NoColor))
22          Skip)))
23    Skip)
```

Figure 8: Specification of the HBW's protocol.

that purpose, we define the semantic function for the RPC language.

To assist with the controller logic, we define the function:

$$\text{find\_color} : \text{color} \to \mathbb{N} \to \text{bays} \to \text{option } \mathbb{N}$$

which acts as the Φ function. `find_color empty b` finds the bay that will be used during a **Store** RPC. Similarly, `find_color c b` finds a item colored c. The function returns an `option` $\mathbb{N}$. Values of type `option a` are either `Some a` or `None`. The natural number inside the option cannot be used without accounting the possibility that the value is missing. With the help of the pre-condition function, we define the semantic function in §A.1. To define a semantic function, the user must match each external command to its valid and invalid semantic rules. As an example of conformant RPC behavior, an **IsFull** response can be sent when the bays are full. On the other hand,

an **IsFull** response sent when one more bays are available is invalid behavior. The semantic function formalizes these semantic rules. The semantic rules will serve as the basis for the safety condition proof.

The proof in §A.2 verifies our safety condition. The proof shows that, for any valid state σ, $\langle spec, \sigma \rangle \not\Rightarrow_{IC}$ **Wrong**. The proof verifies that every combination of atomic propositions allowed by $spec$ in a valid state cannot compute a **Wrong** state. A `match` construct in F* relates atomic propositions to the internal state. As an example, if the labeling function maps a state to APHasStoreRequest and APBaysFull, then the HBW must be full. Otherwise, APBaysFull means we sent an invalid command to the HMI. An `assert` construct in F* checks that to be true. Unlike other ITPs, F* can delegate the verification of boolean equations to SMT solvers. F* uses an SMT solver backend to alleviate some of the proof burden. Instead of writing hints to the proof checker, engineers write assertions they want to verify. These assertions are encoded as SMT formulae that can be checked by an SMT solver. The proof uses the flexibility of this approach. Assertions are checked at compile-time, before the program is executed. The proof is complete once all atomic propositions are checked to reach valid states when adhering to the $spec$ protocol.

Consider a situation where there is a request to retrieve a blue item. Then, a system that follows $spec$ denies the request by sending a **NoColor** RPC. The safety proof guarantees that there is no blue item in the high bay warehouse. The proof uses the the labeling function L to map the current state to the atomic propositions that are true at that state. Then it uses an assert statement to ask the Z3 solver to verify APContainsBlue $\in L(\sigma)$. The SMT backend makes this proof style more ergonomic then solely relying on type-checker.

We provide the implementation of a decider that checks traces are generated by a protocol in F*, as shown in §A.3. The decider uses the `eval_until_next_com` function to break down the trace into smaller command sequences. This function is responsible for breaking down traces into checkable RPC sequences.

Given the proven safety condition, *spec* cannot lead to wrong state. The decider then makes is possible to decide if a trace can be generated by *spec*. The decider will be used to implement the dynamic protocol attestation checker discussed in 4.3.

4.2. RPC Generation for the Fischertechnik

Figure 9a shows F* code that describes the RPCs comprising our protocol. Critically, line 17 shows how to use our metaprogramming system. The `by` syntax invokes F*'s tactic system. F*'s tactic system is an extensible mechanism to automate transforming terms. Individual term transformers are called *tactics*. `gen_rpcs` is a tactic that transforms an RPC framework and an F* type into an RPC encoder. RPC encoders contain code to emit inputs to an RPC framework. Emission occurs by calling the `dump` function. Figure 9b shows the inputs that are automatically generated from the code in Figure 9a.

The tactic system manipulates F*'s abstract syntax tree (AST) using `gen_rpcs`. `gen_rpcs` traverses the AST of the type supplied as its second argument. Several primitive types including machine integers and strings have simple, direct encodings. `gen_rpcs` supports composite types like records by recursively applying itself to struct members. Advanced features in F*'s type system like sum types and refinement types are unsupported.

The RPC framework's existing code generation tool transforms the framework inputs into code. We trust the RPC framework to emit correct code. The generated code contains logic to handle errors ranging from RPC cancellation to network errors. Since common errors are already handled, engineers only need to implement message semantics.

4.3. Dynamic Protocol Attestation

We implement dynamic protocol attestation as a proxy in C++ using Cap'n Proto. The proxy dynamically feeds the current trace to the decider procedure in F*. Since ICS never terminate, our approach ensures we always analyze finite traces. That is done through chopping the trace is smaller parts while keeping track of the state. Our approach was successful in finding non-comformant RPCs.

4.3.1. Performance Evaluation. To understand the effects of dynamic protocol attestation on RPC performance, we conducted a performance evaluation. To conduct our evaluation, we created a software implementation of the Fischertechnik HBW system. Our implementation has four components: a controller, low-level device drivers, a client, and the dynamic attestation system. The controller exposes a store/retrieve interface to the client. The low-level device drivers executes physical actions. The client randomly sends the controller messages that conform to the controller's protocol. We implemented each component in C++. We implemented RPCs using Cap'n Proto with the inputs our automation generated.

We considered two performance criteria in our evaluation. First, *what is the effect on latency?* Second, *what is the effect on message throughput?* We examined two measures of throughput: the number of messages sent, and the number of bytes sent. We evaluated two configurations: one without dynamic protocol attestation enabled, and one with it enabled. We evaluated each configuration for 30 seconds. Table 3 shows our numeric results.

Configuration	Avg. Latency (ms)	Throughput	
		Messages	KB
Without Attestation	0.1452 ± 0.0007	191,250	3,060
With Attestation	0.1903 ± 0.0007	145,054	2,321

TABLE 3: Numeric results from our evaluation.

There is moderate overhead imposed by protocol attestation. Protocol monitoring increases latency by $\approx 31\%$. Similarly, protocol monitoring decreases throughput by $\approx 32\%$. Some applications may tolerate protocol monitoring's performance overhead. Performance overhead can be reduced in future work.

To better understand dynamic protocol attestation's impact on latency, we examined the cumulative density functions (CDFs) of our two evaluation configurations. These CDFs are shown in Figure 10. We observe that protocol monitoring preserves the shape of the original CDF. In particular, 99th percentile latency also increases by $\approx 30\%$. Despite this increase, it is promising that protocol monitoring's worst-case performance appears tied to its average-case performance.

5. Limitations

In this section we will briefly discuss some of the known limitations of our work and possible mitigations for them. The protocols defined with our methodology provide formal guarantees of safety. However, these guarantees exist within a framework of assumptions.

5.1. Fail-Safes Are Not Always Safe

The methodology guarantees that non-conformant RPCs will not execute dangerous commands. Instead, the ICS executes a fail-safe command whenever an RPC would violate the protocol. Often these fail-safe commands are well-known and can be safely executed from any given state the ICS may be in. However, there may be ICSs for which such fail-safes do not exist. Moreover fail-safes may rely on the integrity of the devices that trigger them. Possibly, an alternative backup mechanism could be called when a fail-safe needs to be executed. One such mechanism is to alert a human operator to manually bring the ICS to a known safe state.

Additionally, we assume doing nothing is always safe. This assumption has the unintended consequence of requiring our controller to be sufficiently abstracted away from the drivers. Then, issuing no commands or issuing a fail-safe is always a safe operation. Some ICS subsystems rapidly

```
 1  type store_response = | NotFull | IsFull
 2
 3  type retrieve_response =
 4    | HasColor | DoesNotHaveColor
 5
 6  type color = | Red | White | Blue
 7
 8  type hbw_rpc = {
 9      store_request : color → store_response;
10      retrieve_request : color →
11          retrieve_response;
12      store : color → unit;
13      retrieve : color → unit;
14  }
15
16  let enc : encoder CapnProto hbw_rpc =
17      _ by (gen_rpcs (`CapnProto) (`hbw_rpc))
18
19  dump enc
```

(a) The RPC specification in F*.

```
 1  interface HbwRpc {
 2    storeRequest @0(_0:Color) -> StoreResponse;
 3    retrieveRequest
 4        @1(_0:Color) -> RetrieveResponse;
 5    store @2(_0:Color);
 6    retrieve @3(_0:Color);
 7  }
 8  struct RetrieveResponse {
 9    enum V {hasColor @ 0; doesNotHaveColor @ 1;}
10    v @ 0 : V;
11  }
12  struct StoreResponse {
13    enum V {notFull @ 0; isFull @ 1;}
14    v @ 0 : V;
15  }
16  struct Color {
17    enum V {red @ 0; white @ 1; blue @ 2;}
18    v @ 0 : V;
19  }
```

(b) Cap'n Proto inputs emitted from Figure 9a.

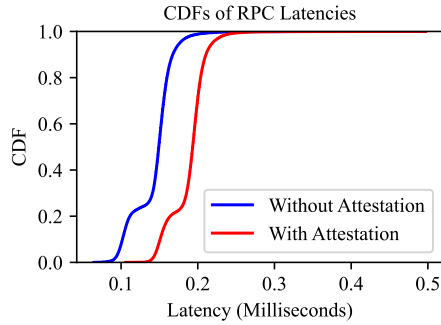Figure 9: Overview of the Fischertechnik's HBW RPC system.

Figure 10: CDFs of RPC latency, without and with dynamic protocol attestation.

issue commands turning another subsystem on and off at a precise interval (e.g., a pulse-width modulated controller). In the event the controller must execute a fail-safe command, it may not be safe to cease issuing other commands. The controller could have been actively preventing another component from causing damage. Our protocol makes no guarantees about the timeliness of the messages between devices. Thus, these time-dependent events should be pushed into the devices' drivers.

5.2. No Liveliness Guarantees; Denial of Service

We assume no guarantees about time within our system nor any liveliness guarantees. A compromised subsystem could continuously execute non-conformant RPCs. Consequently, the only executed commands are fail-safes. Alternatively, the compromised device could refuse to communicate at all, denying service to all devices requiring a response from it. A possible defense to this is to have an internal timeout process within the protocol. The timeout process

would issue a competing command to a controller, instructing the controller to issue a fail-safe within a reasonable amount of time. Such instruction can be implemented by refining the communication semantics.

5.3. Discretization

Our RPC functions allow any type within the function parameters. However, our type system can only reason about countable states and the protocol must be implemented on realizable hardware. This prevents us from reasoning about ICSs with possibly uncountable states or RPCs where infinite precision is required. However, we argue these limitations exist in the current ICS landscape. Furthermore, our solution drastically improves the assurances ICS operators can have on their system.

6. Future Work & Conclusion

6.1. Future Work

Our initial results demonstrate the viability of our approach. We envision four extensions to our work. First, the current DSL to define *spec* requires protocol-specific parameters. Our ultimate goal is to create a general DSL that can describe all protocols. Second, the dynamic checker's overhead might prevent its adoption in certain scenarios. Third, most of the framework in this paper was implemented by hand. Additional automation is needed to create the dynamic checker. We also desire proof automation. Finally, we desire supported for additional RPC frameworks.

6.1.1. Generalizing the DSL. To create DSLs for defining arbitrary protocols, we need a core language. Session types [18] are a typed foundation for the design of message-passing systems. Multiparty session types (MPST)[20] are

able to define properties in systems with multiple components. One of these properties is the ordering of messages. A relevant example is the application of MPST to specify the protocol for the Ocean Observatories Initiative (OOI) [33], [4], [19]. This research inspired the development multiparty session types with payload refinements [45], [42], [32]. Refinement multiparty session types (RMPSTs) are able to also reason about the contents of messages. A possible extension of our work is to embed RMPSTs in F*. This embedding allows users to write typed protocol specifications using RMPSTs. This can be seen a direct extension of [45]. Once having embedded RMPSTs in F*, we plan to integrate with additional type theories including theories of unit types and security type systems (e.g., [39], [25]).

6.1.2. Reducing Dynamic Attestation's Overhead. Our implementation of dynamic protocol attestation is naive. A single, global proxy adds performance overhead because it requires an extra round of serializing and deserializing RPCs. This extra round could be prevented if the proxy deserialized RPCs into shared memory. Deserializing RPCs into shared memory requires a distributed attestation system. We leave this as future work.

6.1.3. Extending Automation. To provide better automation for our approach, we could also benefit from session types. The use of a DSL that implements RMPST would mechanize the process in many ways. First, standard semantics are used for proofs. Second, F*'s type checker can be used to verify protocol implementations. Third, a few core proofs, such as absence of deadlocks, can be proven for all protocols implemented with the DSL. These proofs would be protocol-irrelevant. Users would get these benefits just by using the DSL.

We have shown F*'s use of SMT solvers automates some of the proof process. However, considerable knowledge about F* is needed to prove non-trivial properties of complex system. This is a drawback that precludes adoption of formal methods. We can help reduce proof burden by (1) integrating with model checkers when possible, (e.g., [23], [40]) and (2) providing new tactics and SMT patterns to help further automate proofs.

6.1.4. Additional RPC Framework Support. Currently, This approach currently only supports the Cap'n Proto RPC framework. We would like to expand our extraction of RPCs to handle additional RPC frameworks.

6.2. Conclusion

In this paper, we have presented a methodology that enables engineers to design communication protocols in F*. Communication protocols are provably safe using F*'s theorem proving capabilities. We have also shown how to translate the F* specification into RPCs. By extracting RPCs from the F* specification, we provide a single source of truth governing the ICS's RPCs. We implemented a dynamic attestation system that ensures runtime traces conform to the safe protocol. Critically, our approach limits the impact of compromised components without requiring heavy-weight verification. While these tradeoffs are not insignificant, many ICS could benefit from our proposal, especially when the cost of security incidents is high.

References

[1] Omar Al-Bataineh, Daniel Jun Xian Ng, and Arvind Easwaran. Monitoring cumulative cost properties. In *Proceedings - 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering, FormaliSE 2021*, pages 19–30. Institute of Electrical and Electronics Engineers Inc., 5 2021.

[2] Dejan Baca. Identifying security relevant warnings from static code analysis tools through code tainting. In *2010 International Conference on Availability, Reliability and Security*, pages 386–390. IEEE, 2010.

[3] Sebastian Biallas. Verification of programmable logic controller code using model checking and static analysis. Technical report, RWTH Aachen University, 2016.

[4] Laura Bocchi, Tzu Chun Chen, Romain Demangeon, Kohei Honda, and Nobuko Yoshida. Monitoring networks through multiparty session types. *Theoretical Computer Science*, 669:33–58, 3 2017.

[5] Cap'n Proto. Cve-2022-46149 security advisory, 2022. Accessed: 2024-09-03.

[6] Cap'n Proto Authors. Cap'n proto: Insanely fast data interchange format and rpc system. https://capnproto.org, 2024. Accessed: 2024-06-06.

[7] Thierry Cattel. Process control design using spin, 1995.

[8] Thierry Cattel. Using concurrency and formal methods for the design of safe process control, 1996.

[9] Coverity Scan. Coverity scan: Static analysis for open source projects, 2024. Accessed: 2024-09-03.

[10] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[11] eRPC Project. erpc: Embedded remote procedure call, 2024. Accessed: 2024-09-03.

[12] Borja Fernández Adiego, Ignacio D Lopez-Miguel, Frederic Havart, Enrique Blanco Viñuela, Tomasz Ladzinski, and Jean-Charles Tournier. Applying model checking to highly-configurable safety critical software: the sps-pps plc program. *JACoW*, pages 759–763, 2022.

[13] Fischertechnik. Fischertechnik. https://www.fischertechnik.de/en/, 2024. Accessed: 2024-06-06.

[14] Fischertechnik. Fischertechnik automated high bay warehouse 24v, 2024. Accessed: 2024-10-08.

[15] Vincent Gourcuff, Jean Marc Faure, and Olivier De Smet. Improving large-sized plc programs verification using abstractions. In *IFAC Proceedings Volumes (IFAC-PapersOnline)*, volume 17, 2008.

[16] gRPC Authors. grpc: A high performance, open source universal rpc framework. https://grpc.io, 2024. Accessed: 2024-06-06.

[17] Mohammad Shihabul Haque, Daniel Jun Xian Ng, Arvind Easwaran, and Karthikeyan Thangamariappan. Contract-based hierarchical resilience management for cyber-physical systems. *Computer*, 51:56–65, 11 2018.

[18] Kohei Honda. Types for dyadic interaction*. 1993. Accessed: 2024-09-03.

[19] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation.

[20] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008.

[21] William A Howard et al. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[22] IBM. Industry 4.0, n.d. Accessed: 2024-08-28.

[23] Saurabh Jha, Subho Banerjee, Timothy Tsai, Siva KS Hari, Michael B Sullivan, Zbigniew T Kalbarczyk, Stephen W Keckler, and Ravishankar K Iyer. Ml-based fault injection for autonomous vehicles: A case for bayesian fault injection. In *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 112–124. IEEE, 2019.

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[25] Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D Bond, and Zhiqiang Lin. Cocoon: Static information flow control in rust. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):166–193, 2024.

[26] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1:1–13, 2018.

[27] Ignacio D Lopez-Miguel, Jean-Charles Tournier, and Borja Fernandez Adiego. Plcverif: status of a formal verification tool for programmable logic controller. *arXiv preprint arXiv:2203.17253*, 2022.

[28] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, et al. Meta-f: Proof automation with smt, tactics, and metaprograms. In *European Symposium on Programming*, pages 30–59. Springer International Publishing Cham, 2019.

[29] Dover Microsystems. German steel mill cyberattack, 2024. Accessed: 2024-09-10.

[30] Barton P Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[31] Ramesh Neupane and Hoda Mehrpouyan. An ontology-based framework for formal verification of safety and security properties of control logics. In *2022 14th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pages 1–8. IEEE, 2022.

[32] Rumyana Neykova, Raymond Hu, Nobuko Yoshida, and Fahd Abdeljallal. A session type provider: compile-time api generation of distributed protocols with refinements in f. In *Proceedings of the 27th International Conference on Compiler Construction*, CC '18, page 128–138, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Rumyana Neykova, Nobuko Yoshida, and Raymond Hu. Spy: Local verification of global protocols, 2013.

[34] Christine Paulin-Mohring. Introduction to the calculus of inductive constructions. *All about Proofs, Proofs for All*, 55, 2015.

[35] Olivera Pavlovic and Hans-Dieter Ehrich. Model checking plc software written in function block diagram. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 439–448. IEEE, 2010.

[36] Michael Riley. The real story of stuxnet. *IEEE Spectrum*, 2014. Accessed: 2024-09-10.

[37] Z. Shelby, C. Bormann, L. Frank, S. Kasap, P. Kyzivat, and J. M. D. Morris. The constrained application protocol (coap), 2014. RFC 7252, June 2014. Accessed: 2024-09-03.

[38] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–270, 2016.

[39] Max Taylor, Johnathon Aurand, Feng Qin, Xiaorui Wang, Brandon Henry, and Xiangyu Zhang. Sa4u: practical static analysis for unit type error detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–11, 2022.

[40] Max Taylor, Haicheng Chen, Feng Qin, and Christopher Stewart. Avis: In-situ model checking for unmanned aerial vehicles. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 471–483. IEEE, 2021.

[41] UNSW. sel4 secure microkernel. https://sel4.systems/, 2013. Accessed: 2024-06-06.

[42] Martin Vassor and Nobuko Yoshida. Refinements for Multiparty Message-Passing Protocols: Specification-Agnostic Theory and Implementation. In Jonathan Aldrich and Guido Salvaneschi, editors, *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 41:1–41:29, Dagstuhl, Germany, 2024. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[43] Nat Andreas Wortmann. *Model-Driven Architecture and Behavior of Cyber-Physical Systems*. PhD thesis, RWTH Aachen University, 2021.

[44] Qingzhao Zhang, Xiao Zhu, Mu Zhang, and Z. Morley Mao. Automated runtime mitigation for misconfiguration vulnerabilities in industrial control systems. In *ACM International Conference Proceeding Series*, pages 333–349. Association for Computing Machinery, 10 2022.

[45] Fangyi Zhou, Francisco Ferreira, Raymond Hu, Rumyana Neykova, and Nobuko Yoshida. Statically verified refinements for multiparty protocols. In *OOPSLA*, 9 2020.

[46] Min Zhou, Fei He, Ming Gu, and Xiaoyu Song. Translation-based model checking for plc programs. In *2009 33rd Annual IEEE International Computer Software and Applications Conference*, volume 1, pages 553–562. IEEE, 2009.

[47] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

Appendix A.

A.1. Semantic Function $\mathcal{E}$ in §4.1

The semantic function is used to define the meaning behind executing each RPC. It must specify the valid and invalid cases for each external command defined in the syntax.

```
1  let 𝓔 (c : ExtCom) (s : state) : state =
2    match c, s with
3    | _, Wrong -> Wrong
4    | IsFull, Sigma bay inp ->
5        if None? (find_color' Empty 1 bay)
6        then Sigma bay inp else Wrong
7    | NotFull, Sigma bay inp ->
8        if Some? (find_color' Empty 1 bay)
9        then Sigma bay inp else Wrong
10   | HasColor, Sigma bay inp ->
11       match inp with
12       | None -> Wrong
13       | Some (RetrieveRequest color) ->
```

```
14          if Some? (find_color' color 1 bay)
15          then Sigma bay inp else Wrong
16    | _ -> Wrong
17  | NoColor, Sigma bay inp ->
18      match inp with
19      | None -> Wrong
20      | Some (RetrieveRequest color) ->
21          if None? (find_color' color 1 bay)
22          then Sigma bay inp else Wrong
23      | _ -> Wrong
24  | Store color, Sigma bay inp ->
25      match find_color' Empty 1 bay with
26      | Some n ->
27          Sigma (update_bay bay n color) inp
28      | _ -> Wrong
29  | Retrieve color, Sigma bay inp ->
30      match find_color' color 1 bay with
31      | Some n ->
32          Sigma (update_bay bay n Empty) inp
33      | _ -> Wrong
```

A.2. Proof of Safety Condition in §4.1

The proof shows that for any valid state σ, $\langle spec, \sigma \rangle \not\Rightarrow_{IC}$ **Wrong**. The proof uses the atomic propositions provided by the user to ensure that *spec* can only generate valid traces.

```
1  let lemma_session_good (σ : state{σ ≠ Wrong})
2  : Lemma (ensures σ spec ⇏IC Wrong) =
3  match spec, σ with
4  | If APHasStoreRequest c1 c2,
5      Sigma bays inputs → begin
6   if eval_atomic_prop APHasStoreRequest σ
7     && eval_atomic_prop APBaysFull σ then
8     assert (None? (find_color' Empty 1 bays))
9   else if eval_atomic_prop APHasStoreRequest σ
10     && ¬(eval_atomic_prop APBaysFull σ) then
11    assert (Some? (find_color' Empty 1 bays))
12  else if eval_atomic_prop APHasRetrieveRequest σ
13    && eval_atomic_prop APRequestRed σ
14    && eval_atomic_prop APContainsRed σ then
15    assert (mem APContainsRed (label_fn σ))
16  else if eval_atomic_prop APHasRetrieveRequest σ
17    && eval_atomic_prop APRequestRed σ
18    && ¬(eval_atomic_prop APContainsRed σ) then
19    assert (¬(mem APContainsRed (label_fn σ)))
20  else if eval_atomic_prop APHasRetrieveRequest σ
21    && eval_atomic_prop APRequestWhite σ
22    && eval_atomic_prop APContainsWhite σ then
23    assert (mem APContainsWhite (label_fn σ))
24  else if eval_atomic_prop APHasRetrieveRequest σ
25    && eval_atomic_prop APRequestWhite σ
26    && ¬(eval_atomic_prop APContainsWhite σ) then
27    assert (¬(mem APContainsWhite (label_fn σ)))
28  else if eval_atomic_prop APHasRetrieveRequest σ
29    && eval_atomic_prop APRequestBlue σ
30    && eval_atomic_prop APContainsBlue σ then
31    assert (mem APContainsBlue (label_fn σ))
32  else if eval_atomic_prop APHasRetrieveRequest σ
33    && eval_atomic_prop APRequestBlue σ
34    && ¬(eval_atomic_prop APContainsBlue σ) then
35    assert (¬(mem APContainsBlue (label_fn σ)))
36  else ()
37  end
38  | _ → ()
```

A.3. Decider

A decider procedure is implemented as part of the dynamic attestation checking. The decider checks that traces are generated by *spec*

```
1  let rec is_trace
2    (t : IntCom) (tr : list ExtCom) (st : state)
3    : Tot bool (decreases (length tr)) =
4    match tr with
5    | [] → true
6    | com :: tr' →
7      let cmd', term', state' =
8        eval_until_next_com st t in
9      match cmd' with
10     | None → false
11     | Some com' →
12       if None? term' then
13         com = com' && is_trace spec tr' state'
14       else
15         com = com' &&
16         is_trace (unwrap_some term') tr' state'
```

A.4. Example of Valid Traces

This example shows how a trace can be validated by the dynamic attestation checker, given the current state and HMI inputs.

```
1      RPCs:Notfull – Store red – Hascolor –
2      Retrieve blue – Hascolor – Retrieve white –
3      Notfull – Store red
4
5      At state
6  ⟨ { bay1 : white; bay2 : blue; bay3 : empty;
7      bay4 : white; bay5 : red; bay6 : red;
8      bay7 : empty; bay8 : empty; bay9 : empty }
9      StoreRequest red ⟩.
10
11     With the following HMI inputs:
12 ⟨ { bay1 : white; bay2 : blue; bay3 : empty;
13     bay4 : white; bay5 : red; bay6 : red;
14     bay7 : empty; bay8 : empty; bay9 : empty }
15     StoreRequest red ⟩.
16 RPCs:Notfull – Store red
17
18 ⟨ { bay1 : white; bay2 : blue; bay3 : red;
19     bay4 : white; bay5 : red; bay6 : red;
20     bay7 : empty; bay8 : empty; bay9 : empty }
21     RetrieveRequest blue ⟩.
22 RPCs:Hascolor – Retrieve blue
23
24 ⟨ { bay1 : white; bay2 : empty; bay3 : red;
25     bay4 : white; bay5 : red; bay6 : red;
26     bay7 : empty; bay8 : empty; bay9 : empty }
27     RetrieveRequest white ⟩.
28 RPCs:Hascolor – Retrieve white
29
30 ⟨ { bay1 : empty; bay2 : empty; bay3 : red;
31     bay4 : white; bay5 : red; bay6 : red;
32     bay7 : empty; bay8 : empty; bay9 : empty }
33     StoreRequest red ⟩.
34 RPCs:Notfull – Store red
35
36 ⟨ { bay1 : red; bay2 : empty; bay3 : red;
37     bay4 : white; bay5 : red; bay6 : red;
38     bay7 : empty; bay8 : empty; bay9 : empty }
39     StoreRequest red ⟩.
```

A.5. Examples of Invalid Traces

The examples listed below break the safety properties in §2.1.3. The examples show possible $State \times ExtCom$ pairs that would trigger a fail-safe.

```
 1      1- Store wrong color
 2  ⟨ { bay1 : white; bay2 : blue; bay3 : empty;
 3      bay4 : white; bay5 : red; bay6 : red;
 4      bay7 : empty; bay8 : empty; bay9 : empty }
 5      StoreRequest red ⟩.
 6  RPCs: Notfull - Store blue
 7
 8      2- Store with full
 9  ⟨ { bay1 : white; bay2 : blue; bay3 : white;
10      bay4 : white; bay5 : red; bay6 : red;
11      bay7 : red; bay8 : blue; bay9 : blue }
12      StoreRequest red ⟩.
13  RPCs: Notfull - Store red
14
15      3- Command mismatch
16  ⟨ { bay1 : white; bay2 : blue; bay3 : empty;
17      bay4 : white; bay5 : red; bay6 : red;
18      bay7 : empty; bay8 : empty; bay9 : empty }
19      StoreRequest red ⟩.
20  RPCs: Hascolor - Retrieve red
21
22      4- Response Mismatch
23  ⟨ { bay1 : white; bay2 : blue; bay3 : empty;
24      bay4 : white; bay5 : red; bay6 : red;
25      bay7 : empty; bay8 : empty; bay9 : empty }
26      RetrieveRequest red ⟩.
27  RPCs: Notfull - Retrieve red
28
29      5- Retrieve with no color
30  ⟨ { bay1 : white; bay2 : blue; bay3 : empty;
31      bay4 : white; bay5 : blue; bay6 : blue;
32      bay7 : empty; bay8 : empty; bay9 : empty }
33      RetrieveRequest red ⟩.
34  RPCs: Hascolor - Retrieve red
```