Privacy-Preserving Runtime Verification

by

Mahyar Karimi

March, 2026

A thesis submitted to the
Graduate School
of the
Institute of Science and Technology Austria
in partial fulfillment of the requirements
for the degree of
Master of Science

Committee in charge:

Thomas A. Henzinger, Supervisor
Krzysztof Pietrzak

The thesis of Mahyar Karimi, titled *Privacy-Preserving Runtime Verification*, is approved by:

Supervisor: Thomas A. Henzinger, ISTA, Klosterneuburg, Austria

Signature: ______________________________

Committee Member: Krzysztof Pietrzak, ISTA, Klosterneuburg, Austria

Signature: ______________________________

Signed page is on file

ISTA Master's Thesis, ISSN: 2791-4585

Abstract

Runtime verification offers scalable solutions to improve the safety and reliability of systems. However, systems that require verification or monitoring by a third party to ensure compliance with a specification might contain sensitive information, causing privacy concerns when usual runtime verification approaches are used. Privacy is compromised if protected information about the system, or sensitive data that is processed by the system, is revealed. In addition, revealing the specification being monitored may undermine the essence of third-party verification.

In this thesis, we propose a protocol for privacy-preserving runtime verification of systems against formal sequential specifications. We develop the protocol in two steps. In the first step, the monitor verifies whether the system satisfies the specification without learning anything else, though both parties are aware of the specification. In the second step, we extend the protocol to ensure that the system remains oblivious to the monitored specification, while the monitor learns only whether the system satisfies the specification and nothing more. Our protocol adapts and improves existing techniques used in cryptography, and more specifically, multi-party computation.

The sequential specification defines the observation step of the monitor, whose granularity depends on the situation (e.g., banks may be monitored on a daily basis). Our protocol exchanges a single message per observation step, after an initialization phase. This design minimizes communication overhead, enabling relatively lightweight privacy-preserving monitoring. We implement our approach for monitoring specifications described by register automata and evaluate it experimentally.

Acknowledgements

I thank my supervisor, Thomas A. Henzinger, for his guidance and support throughout this project. I thank K. S. Thejaswini for her equal contributions to the research that forms the basis of this thesis; the protocol design, security proofs, and theoretical framework were developed collaboratively. I also thank Krzysztof Pietrzak for serving on my thesis committee and for insightful discussions on the cryptographic aspects of this work.

This work is part of the project VAMOS, which has received funding from the European Research Council (ERC) under grant agreement No. 101020093, and the Austrian Science Fund (FWF) SFB project SpyCoDe F8502.

About the Author

Mahyar Karimi completed his BSc in Computer Engineering at the University of Tehran before joining ISTA as a researcher supervised by Thomas A. Henzinger. His research lies at the intersection of formal verification and cryptography, with a particular focus on runtime verification (monitoring) and secure multi-party computation. His work has been published in a top-tier conference in the fields of computer security and formal methods.

List of Collaborators and Publications

This thesis is based on the following publication:

Thomas A. Henzinger, Mahyar Karimi, and K. S. Thejaswini. 2025. Privacy-Preserving Runtime Verification. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (CCS '25), October 13–17, 2025, Taipei, Taiwan. ACM, New York, NY, USA. `https://doi.org/10.1145/3719027.3765137`

All authors contributed equally to this work. The theoretical framework, protocol design, and security proofs were developed collaboratively. The experimental design, implementation, and evaluation were carried out by Mahyar Karimi.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Ensuring that complex systems behave correctly is a central challenge in computer science. The two classical approaches to exhaustive verification, model checking and theorem proving, offer strong guarantees but share a common prerequisite: they require access to the system's source code or a formal model of it. Model checking must explore the reachable state space of a system model, which grows exponentially in the number of components; theorem proving requires substantial manual effort and domain expertise. Both are inherently *static* methods that analyze the system before deployment, and neither can be applied when the source is unavailable—for instance, when the system is a proprietary third-party service or a legacy component without documentation.

Runtime verification (RV) takes a fundamentally different angle [LS09, BFFR18, HG08]. Rather than reasoning about all possible behaviors from the source, RV monitors the actual behavior of a running system by observing the stream of events it produces—its execution *trace*. A *monitor* checks, step by step, whether the observed trace conforms to a given specification. This removes the dependency on the source code entirely: the monitor needs no access to the system's internals, only to its observable outputs. RV cannot provide exhaustive coverage—it analyzes only the current run—but it gives precise, timely verdicts at computational costs that are modest relative to the system being observed. For large, evolving, or deployed systems where exhaustive verification is out of reach, this is an attractive trade-off. We provide further background on runtime verification, including specification languages, monitoring architectures, and monitor synthesis, in Section 2.1.

However, RV introduces a dependency of its own: the monitor must have access to the system's execution trace. In many practical settings, the monitor is not the system's own operator but an external party—a regulator, an auditor, or a certification body—that must independently verify compliance with a given standard [BFFR18, JLK$^+$16]. This separation between the monitored system and the monitoring authority raises a question that does not arise in self-monitoring: *what information must the system reveal to the monitor, and what should remain hidden?*

1.1 Privacy in third-party monitoring

This question has no easy answer, because monitoring and privacy pull in opposite directions. Monitoring requires inspecting the system's execution trace to check it against a specification, while privacy demands that sensitive details of that trace (trade secrets, personally identifiable data, proprietary system internals) remain undisclosed. The tension is concrete: a bank undergoing a regulatory audit may need to share transaction data to demonstrate compliance; a hospital may need to report operational statistics to meet healthcare standards; a self-driving car or metro system may be monitored for safety by an external certification body. In each case, the system must convince a third-party monitor of its compliance without revealing protected information.

One possible solution is to allow the system to self-monitor by embedding the specification into the system and generating its own verification results. However, this approach lacks credibility in settings where third-party validation is crucial. For instance, if hospitals were entirely self-certified, the certificates would lack the impartiality needed to inspire public confidence. This necessitates a solution that allows third-party monitors to continuously and repeatedly verify whether a system complies with a given specification, all while obscuring the internals of the system and its data.

Hiding the system's data from the monitor is only one dimension of the problem. In some settings, the specification itself must also remain secret. Consider again the domain of healthcare, where runtime verification of clinical traces has already been studied [JLK+16]. In the National Health Services (NHS) in England, hospital funding is tied to performance metrics, optimization of which led many hospitals to restructure their operations [Cra17, Mea14]. Unfortunately, such restructurings resulted in extreme cases like the events at Mid-Staffordshire NHS Foundation Trust, where financial targets led to patient neglect [IF13]. This exemplifies Goodhart's Law: *"When a measure becomes a target, it ceases to be a good measure,"* or, as an ex-NHS manager put it, *"hitting the target but missing the point."* Indeed any measure has the potential to become a target when gamifications are employed to achieve that target. To avoid such distortions, it is crucial to also develop protocols for monitoring that keep both the system/data and specification private, ensuring accurate and unbiased evaluation.

Recent advances have demonstrated that verification can be performed in a privacy-preserving manner spanning a variety of domains, including SAT solving [LJA+22], verifying resolution proofs [LAH+22], matching strings against regular expressions [LWS+24], and model-checking specifications described by CTL formulas [JLAP20]. However, privacy-preserving verification often introduces significant computational overhead. This poses scalability challenges, particularly in real-world applications where the underlying systems involve an enormous number of states. In contrast, runtime verification focuses not on verifying the entire system but rather on monitoring specific outputs produced by the system during execution. This approach inherently involves smaller-scale data exchanges, making it a suitable candidate for privacy-preserving methods.

1.2 Contributions

We provide a protocol that would aid a monitoring setup as shown in Fig. 1.1. Implementing our protocol on the system's side and the monitor's side enables monitoring where only one message is sent per observation round, the specification is kept secret, and the observable outputs of the system are kept secret.

We model the specification as a Boolean circuit that takes two inputs—the current monitor state and the current system observation—and produces two outputs: the next monitor state and a single-bit *flag* indicating whether the observed prefix violates the specification. Our protocol ensures that in each round, the monitor learns exactly one bit—the flag—and nothing else: neither the input, output, nor internal state of the monitored system, nor even the internal state of the specification itself.

We develop the protocol in two steps, addressing two security settings:

(a) **Open specification.** The specification is not a secret and is known to both parties. The protocol in this setting is a modification of the classic MPC protocol known as Yao's garbled circuits [Yao86, GMW87]. We modify the classical protocol to enable repeated computation while maintaining the secrecy of the specification state.

(b) **Hidden specification.** The specification is a secret and known only to the monitor. Inspired by recent advances in private function evaluation (PFE), we extend the protocol to additionally hide the circuit topology using techniques based on the Decisional Diffie-Hellman (DDH) assumption. We build upon the seminal work of Katz and Malka [KM11] and the recent work of Liu, Wang, and Yu [LWY22].

Note that in setting (a), the monitored system knows which of its parts (which system inputs, which system outputs, and which internal system states) are being examined by the specification, while in setting (b) it does not. Hence setting (b) is particularly interesting for large systems being monitored against small specifications.

Our protocol adapts and combines existing cryptographic techniques—Yao's garbled circuits, the DDH-based label structure of Liu, Wang, and Yu [LWY22], and Bellare-Micali oblivious transfer [BM89]—to the reactive monitoring setting. Two-party protocols for reactive functionalities are well-studied [HL10]; the cryptographic building blocks we use are standard. Our main practical design choice is that, after an initial setup phase, each monitoring round requires only a single message from the system to the monitor. Secret-sharing-based MPC requires three or more parties [GMW87] and is therefore inapplicable in our two-party setting, while OT-based two-party alternatives exchange at least three messages per round [CSW20]. FHE-based constructions, such as that of Banno et al. [BMM$^+$22], face a different obstacle: their evaluation cost is linear in the DFA size of the specification, which can be exponentially larger than the circuit representation we use (see Chapter 5). The single-message property is convenient when hardware for bidirectional communication is limited.

We implemented our protocol to analyze the influence of several parameters involved in building such protocols. We use specifications described as register automata [GDPT13], which we convert to Boolean circuits. Our experiments show the feasibility of our protocol when the circuit sizes are on the order of 10^5 for acceptable security parameters. Additionally, in the hidden-specification setting, the time per round is influenced more by the size of the specification than by the size of the monitored system. This allows scalability to large system sizes as long as the specification remains small.

1.3 Thesis outline

This thesis is organized as follows.

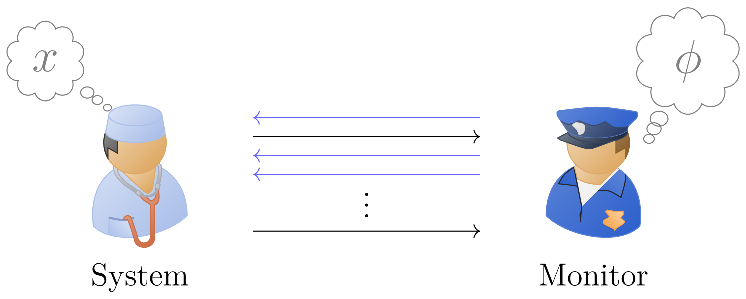


Figure 1.1: The privacy-preserving monitoring setting. The physician represents the System (e.g., a medical center) and the police officer represents the Monitor (e.g., law enforcement).

Chapter 2: Background. We provide the necessary background on runtime verification, multi-party computation, garbled circuits, oblivious transfer, and the Decisional Diffie-Hellman assumption.

Chapter 3: Problem Formulation. We formally define privacy-preserving monitoring, including the ideal settings, correctness, and security definitions based on the simulation paradigm.

Chapter 4: The Protocol. We introduce a general framework for reactive secure monitoring, discuss the design space and motivate our construction choices, and then present the protocol in two steps. First, we describe the open-specification setting where both parties know the specification circuit. Then, we extend the protocol to the hidden-specification setting, where the circuit topology must also be kept secret.

Chapter 5: Experimental Evaluation. We describe our C++ prototype implementation and evaluate the protocol on two monitoring scenarios: an access control system and a lock management system for parallel programs.

Chapter 6: Related Work. We survey related work in privacy-preserving verification and monitoring, and compare our approach with existing methods.

Chapter 7: Conclusion and Future Work. We summarize our contributions and discuss directions for future research.

CHAPTER 2

Background

In this chapter, we provide the necessary background for understanding the protocol presented in this thesis. We begin with a detailed account of runtime verification, covering specification languages, monitoring architectures, and the specification model we use. We then introduce the cryptographic primitives that form the foundation of our privacy-preserving protocol: garbled circuits, oblivious transfer, and the Decisional Diffie-Hellman assumption. We also discuss the simulation-based security model and reactive functionalities.

2.1 Runtime verification and monitoring

Model checking and theorem proving both require a formal representation of the system under analysis. In model checking, the system is described as a finite-state model, typically a Kripke structure or a labeled transition system constructed from the source code or a design artifact, and the verifier exhaustively explores its reachable state space to determine whether a temporal logic formula holds over all possible runs. Theorem proving demands something similar: a formal specification of the program's intended behavior is developed alongside the implementation, and proofs are constructed that relate the two. In both cases, access to the source code is a prerequisite for the method to apply. Verification happens before deployment, on a representation of the system, not on the system itself.

Runtime verification takes a different approach by treating the system as a black box observed through its outputs [LS09, BFFR18]. A *monitor* receives a stream of events produced by the running system and checks, step by step, whether the observed sequence of events conforms to a given specification. No access to the source code is required; the monitor interacts only with what the system produces, not with how it produces it. This makes runtime verification applicable where static analysis is impractical: for proprietary components whose source is unavailable, for deployed systems that cannot be taken offline, or for systems too large to model exhaustively. The cost is that the monitor cannot reason about behaviors it has not seen; it checks the one execution in front of it, not all possible runs. The dependency on the source is replaced by a dependency on the execution trace.

The field of runtime verification, as a self-named community, grew out of the RV workshop established in 2001, which became a full conference in 2010 and has been held annually since then [BFFR18]. Research on monitoring techniques has been around for much longer, present in other communities such as testing, dynamic analysis, and trace analysis that do not always use the same name. RV is now employed in both academia and industry, before deployment for testing and debugging, and after deployment to ensure reliability, safety, and compliance [BFFR18, JLK$^+$16].

Specification languages. The system's behavior is abstracted as a sequence of observable *events*, forming an execution *trace*, and a specification describes the set of acceptable (or unacceptable) traces. The most common specification language in RV is linear temporal logic (LTL) [Pnu77], which expresses properties such as "every request is eventually followed by a response" using operators like *always* and *eventually.* State machines and regular expressions provide more operational alternatives. Since observed traces are necessarily finite prefixes of potentially infinite executions, Bauer, Leucker, and Schallhart [BLS11] developed a three-valued semantics for LTL (true, false, and inconclusive) that accounts for this incompleteness.

Many real systems produce events that carry data: a function call carries its arguments, a network packet carries its payload. Standard propositional LTL cannot express properties over such data; parametric extensions of temporal logics and state machines address this by introducing variables and guards over data domains [BFFR18]. Our own specifications follow this pattern, using register automata [GDPT13] that store and compare data values across events.

Monitoring setup. A typical RV setup consists of three components: the *system under scrutiny*, the *monitor*, and the *instrumentation* that connects them [BFFR18]. The instrumentation determines which aspects of the system's execution are made visible to the monitor and how the two execute in relation to one another. *Offline* monitoring records the execution trace in full and analyzes it after the system terminates; this is less intrusive but delays the detection of violations. *Online* monitoring runs the monitor concurrently with the system and processes events as they arrive, enabling timely detection at the cost of tighter performance constraints.

Within online monitoring, there is a further distinction in how tightly the monitor and system are coupled. In *synchronous* monitoring, the system blocks after each event until the monitor has processed it; this guarantees that every violation is caught before the next step, but the monitoring overhead directly slows down the system. In *asynchronous* monitoring, the system and monitor execute independently, with events buffered and forwarded without blocking; this reduces the performance impact on the system but may delay detection and, if the monitor falls behind, can introduce gaps in the analysis [BFFR18]. The overhead question is particularly important in safety-critical or real-time applications, where the additional latency introduced by a synchronous monitor may itself violate timing constraints. Sampling-based techniques that observe only a subset of events offer one way to control overhead, though they introduce uncertainty into the monitoring verdict.

Monitor synthesis and monitorability. Monitors are rarely written by hand. Instead, they are typically generated by *automated synthesis procedures* that take a formal specification as input and produce an executable monitor as output [BFFR18]. Frameworks such as MOP [CR07] have demonstrated this approach for a variety of specification languages, including temporal logics, regular expressions, and context-free grammars.

Automated synthesis serves two purposes: it eliminates manual implementation errors in the monitor, and it provides a clear formal relationship between the specification and the monitor's behavior, which makes it possible to reason about the monitor's correctness. This is relevant to our setting because the monitor in our protocol operates on a Boolean circuit representation of the specification, which is itself derived from a register automaton through an automated compilation pipeline (see Chapter 5).

Not all properties, however, can be meaningfully monitored at runtime. The question of *monitorability*, whether a given property can be checked by a monitor that processes finite prefixes of a trace, has been studied extensively [PZ06, BLS07]. Safety properties, which stipulate that "nothing bad happens," are naturally suited to runtime monitoring: any violation of a safety property manifests as a finite prefix that cannot be extended to a correct execution [KKL$^+$02]. Our work, like much of the RV literature, focuses on safety specifications for this reason.

Our specification model. In this work, we model specifications as *safety specifications* [KKL$^+$02]. A safety specification can be represented as a state machine: the monitor starts in an initial state and, at each observation round, transitions to a new state based on the current observation. A Boolean function, which we call flag, indicates whether the observed prefix violates the specification. We model the specification as a circuit $\mathcal{C}$ that computes two functions simultaneously: a next-state function $\mathsf{nextstate}$ and the violation flag flag. The circuit takes as input the current monitor state $\mu[r]$ and the current system observation $\sigma[r]$, and produces the next monitor state $\mu[r+1] = \mathsf{nextstate}(\mu[r], \sigma[r])$ and the flag $\tau[r] = \mathsf{flag}(\mu[r], \sigma[r])$. For our implementations, we use specifications described as register automata [GDPT13], which we convert into Boolean circuits using the Yosys synthesis suite.

2.2 Multi-party computation

Multi-party computation (MPC), studied since the 1980s [Yao82], allows two or more parties who do not trust each other to collaboratively compute a function on their private inputs without revealing these inputs to each other. Typically, MPC focuses on a "one-shot" setting, where the parties compute one or more outputs based on their inputs, but do so exactly once. A specific variant of MPC that is relevant to our work is *private function evaluation* (PFE), where one party owns a private function (a "secret circuit") along with a part of the input, while the other party holds the rest of the input. The goal is to design a protocol that allows the function to be evaluated using the inputs of both parties securely: one or both parties can learn the result of the function on the input, but nothing else is revealed. Runtime verification, however, requires not a one-shot but a repeated evaluation process, whose granularity—called the (*observation*) *round*—depends on the application. The performance of banks may be observed daily, hospitals monthly, autopilots and metro systems every second. Such *reactive functionalities* are important for monitoring systems, where internal states of the system (e.g., accumulated data or ongoing logs) and the monitor (for sequential specifications) need to remain hidden even across repeated interactions.

2.3 Yao's garbled circuits

The most fundamental tool used in secure two-party computation is attributed to Yao and was dubbed Yao's garbling by Goldreich, Micali, and Wigderson [GMW87].

2.3.1 Garbling a single gate

We first describe a toy version of the problem posed by Yao. Consider two parties A and B. Can we have a secure protocol where party A has two bits, and B wants to know the output of gate G computing a Boolean function over two bits, where gate G is known to both parties? Yao's garbling produces a simple solution to this as follows.

Party A starts by randomly generating strings of a fixed length: L^0, L^1, R^0, R^1, S^0, and S^1. The strings L^0 and L^1 correspond intuitively to each value 0 and 1 taken by the left input wire of the gate, respectively. Similarly, R^0 and R^1 correspond to the values taken by the right input wire, and S^0 and S^1 to the output wire taking values 0 and 1, respectively.

After the labeling step, Party A *encrypts* the label of the output of G using keys that correspond to the input. So, if party B had keys that correspond to input $(0, 1)$, then it can only open the output that would represent $G(0, 1)$. More formally, party A calls a subroutine $\mathbf{encYao}_G$ that generates the *garbled gate* which consists of four ciphertexts as follows:

$$\mathbf{encYao}_G\left([L^0, L^1], [R^0, R^1], [S^0, S^1]\right) := \left\{\mathrm{Enc}_{L^\alpha, R^\beta}\left(S^{G(\alpha,\beta)}\right)\right\}_{\alpha,\beta\in\{0,1\}} \tag{\$}$$

and sends it to party B, but the encrypted messages are sent in random order. That is, if gate G was an *AND* gate, then the garbled gate would be a random order of the elements $\{\mathrm{Enc}_{L^0,R^0}(S^0), \mathrm{Enc}_{L^1,R^0}(S^0), \mathrm{Enc}_{L^0,R^1}(S^0), \mathrm{Enc}_{L^1,R^1}(S^1)\}$. Party A also sends $\langle S^0, 0\rangle$ and $\langle S^1, 1\rangle$ to indicate to B that S^0 corresponds to bit 0 and S^1 to bit 1. If party A's input for the left and the right gate are $\ell, r \in \{0, 1\}$, then she also sends the random labels L^ℓ and R^r.

Party B then uses L^ℓ and R^r as keys to open the four ciphertexts; with very high probability, only one of the four will open, which corresponds to exactly $S^{G(\ell,r)}$. If $S^{G(\ell,r)} = S^0$, he concludes $G(\ell, r) = 0$, and 1 if $S^{G(\ell,r)} = S^1$.

2.3.2 Extension with oblivious transfer

Now, consider the same situation with a gate G over two bits; however, one bit of input is known to party A and the other bit is known to party B. Can we modify the above protocol to ensure that party B learns $G(\alpha, \beta)$ without learning α, where α is party A's input and β is party B's input?

This is where *oblivious transfer* (`OT`) comes in. Oblivious transfer is a two-party functionality in which a sender holds two strings $x_0, x_1 \in \{0, 1\}^n$ and a receiver holds a choice bit $\beta \in \{0, 1\}$. The functionality delivers x_β to the receiver while hiding β from the sender and $x_{1-\beta}$ from the receiver:

$$\mathtt{OT}\colon \{0,1\}^{2n} \times \{0,1\} \to \{0,1\}^n, \qquad \mathtt{OT}((x_0, x_1), \beta) = x_\beta.$$

We describe a concrete protocol realizing this functionality in Section 2.4.

Using `OT` as a sub-protocol, we can modify the previously described garbling protocol. The garbling protocol proceeds as follows. Party A similarly finds labels L^0, L^1, R^0, R^1, S^0, and S^1 corresponding to 0 or 1 for the wires, and prepares the garbled gates as described in the previous step. After this, party A and party B run a protocol for oblivious transfer where A has the labels for the input wire corresponding to B's input β, say R_0 and R_1, and B has the input bit. At the end of the `OT` protocol, B would receive R_β. Later, party A also sends L_α. This way, out of the four ciphertexts with the keys L_α and R_β, party B can only open the one ciphertext that contains the key $S^{G(\alpha,\beta)}$. He matches this string obtained with S^0 or S^1, both received from A.

2.3.3 Garbling an entire circuit

Consider the same problem; however, instead of just a simple gate G, it is a circuit $\mathcal{C}$ that party B wants to use to evaluate on party A's input. Then for each wire W in the circuit, party A similarly prepares random keys to represent the value W^0 and W^1. Whenever an output wire feeds into an input for a gate, party A uses the same random keys for the output and input wire. For each gate G in the circuit $\mathcal{C}$, party A computes $\textbf{encYao}_G$ using the corresponding gate's inputs and output wire labels. Party A further sends the labels generated for the output wire along with whether they correspond to 0 or 1 to B, and the input labels of the wires which correspond to her input.

Party B can use the keys corresponding to the input of party A and unlock the gates to learn the keys to the next gates in a bottom-up manner and work through the circuit until he obtains the keys corresponding to the output wires. Then party B can match the keys with the corresponding values sent by A.

To summarize, Yao's garbled circuits allow two parties to evaluate a Boolean circuit on their joint inputs without revealing those inputs to each other. Party A (the garbler) encodes the circuit by replacing each wire value with a random label and encrypting each gate's output label under the corresponding input labels. Party B (the evaluator) uses the labels it receives—its own obtained via oblivious transfer, A's sent directly—to decrypt the circuit gate by gate and learn only the final output. This is the core building block of our protocol; the remaining challenge is adapting it to the *reactive* setting, where the circuit must be evaluated repeatedly with hidden carry-over state.

2.4 Oblivious transfer

We introduced the oblivious transfer functionality $\texttt{OT}\colon \{0,1\}^{2n} \times \{0,1\} \to \{0,1\}^n$ in Section 2.3. Secure protocols for `OT` have been known since the 1980s, with the earliest forms proposed by Rabin [Rab81], and later improvements and alternative protocols by Kilian [Kil88], Bellare and Micali [BM89], and several others.

The Bellare-Micali protocol. Our implementation uses the `OT` protocol of Bellare and Micali [BM89], which operates in a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order q. The protocol proceeds in three messages:

1. **Sender setup.** Party A (the sender) chooses a random group element $C \xleftarrow{\$} \mathbb{G}$ and sends C to party B.

2. **Receiver choice.** Party B (the chooser) picks a random exponent $k \xleftarrow{\$} \mathbb{Z}_q$ and computes two public keys: $\mathsf{pk}_\beta = g^k$ and $\mathsf{pk}_{1-\beta} = C/g^k$. Party B sends both pk_0 and pk_1 to party A.

3. **Sender encryption.** Party A checks that $\mathsf{pk}_0 \cdot \mathsf{pk}_1 = C$, then for each $i \in \{0,1\}$, picks a random $r_i \xleftarrow{\$} \mathbb{Z}_q$ and computes the ciphertext $(c_{i,1}, c_{i,2}) = (g^{r_i}, H(\mathsf{pk}_i^{r_i}) \oplus x_i)$ where H is a hash function modeled as a random oracle. Party A sends both ciphertexts to B.

Party B can decrypt only the β-th ciphertext: using its secret key k, it computes $H(c_{\beta,1}^k) \oplus c_{\beta,2} = x_\beta$. The other ciphertext is encrypted under the public key $\mathsf{pk}_{1-\beta} = C/g^k$. Decrypting it would require the discrete logarithm of $\mathsf{pk}_{1-\beta}$, i.e., the value k' such that $g^{k'} = C/g^k$. Since C was chosen uniformly at random by party A, the discrete logarithm of C is unknown to B, and therefore so is k'.

Two-party protocols for reactive functionalities are well-studied [HL10]; standard OT-based constructions require at least three message exchanges per round [CSW20]. A naïve application of `OT` in every round can therefore become a communication bottleneck; in Chapter 4 we show how to amortize this cost to a single message per round.

2.5 The Decisional Diffie-Hellman assumption

Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order q and let $a, b \in \mathbb{Z}_q$. Since exponentiation commutes, $(g^a)^b = (g^b)^a = g^{ab}$, so a party who knows a can compute g^{ab} from g^b, and vice versa. The *Computational* Diffie-Hellman assumption (CDH) asserts that, without knowledge of either exponent, computing g^{ab} from (g^a, g^b) is infeasible. The stronger *Decisional* Diffie-Hellman assumption (DDH) asserts that one cannot even *distinguish* g^{ab} from a uniformly random group element [Bon98].

Assumption 2.1 (Decisional Diffie-Hellman assumption [Bon98]) *Let $\mathbb{G} = \langle g \rangle$ be a cyclic group of prime order $q \in \Theta(2^k)$. The DDH assumption states that no probabilistic polynomial-time algorithm can distinguish, with non-negligible advantage, between the following two distributions:*

- Real DH tuples: $(g^a,\ g^b,\ g^{ab})$, *where* $a, b \xleftarrow{\$} \mathbb{Z}_q$;
- Random tuples: $(g^a,\ g^b,\ g^c)$, *where* $a, b, c \xleftarrow{\$} \mathbb{Z}_q$.

Note that the distinguisher observes g^a and g^b but does not know the exponents a or b.

The requirement that $\mathbb{G}$ have prime order is essential: in groups whose order has small prime divisors, the Legendre symbol leaks information about g^{ab}, making DDH trivially false [Bon98, Section 1.1]. This property is crucial for our hidden-specification setting, where the circuit topology must be obfuscated.

A key result we use in our security proofs is a lemma from Naor and Reingold [NR04], which lifts the DDH assumption from pairs to n-tuples:

Lemma 2.1 ([NR04, Lemma 4.4]) *Assuming that the DDH assumption holds in a cyclic group $\mathbb{G} = \langle g \rangle$ of prime order $q \in \Theta(2^k)$ for $n \in \mathrm{poly}(\kappa)$, given n randomly chosen elements from the group $g_1, g_2, \ldots, g_n \xleftarrow{\$} \mathbb{G}$ and $n+1$ randomly chosen exponents $a, a_1, a_2, \ldots, a_n \xleftarrow{\$} \mathbb{Z}_q$, we have that $(g_1^a, g_2^a, \ldots, g_n^a)$ is computationally indistinguishable from an n-tuple $(g_1^{a_1}, g_2^{a_2}, \ldots, g_n^{a_n})$.*

2.6 Simulation-based security

The standard framework for proving the security of cryptographic protocols is the *real/ideal paradigm* [Lin17]. The idea is to define two worlds: an *ideal world* and a *real world*. In the ideal world, the parties hand their private inputs to a trusted third party, who performs the computation honestly and returns the correct outputs—no messages are exchanged between the parties, so there is nothing to “leak.” In the real world, no trusted third party exists; the parties interact directly using the protocol. A protocol is said to be *secure* if the real-world execution is computationally indistinguishable from the ideal-world execution.

More concretely, security is proven by constructing a *simulator* for each party. A simulator is a probabilistic polynomial-time algorithm that takes as input only what the party is entitled to know—its own private inputs and the outputs it is supposed to receive—and produces a *simulated transcript*. If the simulated transcript is computationally indistinguishable from the party’s *real view* (i.e., the collection of all messages the party received, along with its own random coins), then we conclude that the protocol reveals nothing beyond what the party could have computed from its inputs and outputs alone. The intuition is that if an efficient simulator can fabricate a convincing transcript without access to the other party’s secrets, then the real transcript cannot contain meaningful information about those secrets either.

Constructing such a simulator is often the hardest part of a security proof. The simulator must account for every message the party observes and produce each component of the transcript in a way that is statistically or computationally close to the real distribution, despite not knowing the other party’s private input. In garbled-circuit-based protocols, for example, the simulator for the evaluator must produce garbled gates that “look real” even though the simulator does not know the garbler’s input. The standard strategy, detailed in Lindell’s tutorial [Lin17], is to proceed via a *hybrid argument*: one defines a sequence of intermediate experiments, each differing from the previous one in a single component, and shows that adjacent experiments are indistinguishable under a cryptographic assumption. The first experiment corresponds to the real execution and the last to the simulated one; if the endpoints were distinguishable with non-negligible advantage ε, then by an averaging argument some adjacent pair of hybrids would be distinguishable with advantage at least ε/n, contradicting the underlying cryptographic assumption.

In our setting, we consider *semi-honest* (also called *honest-but-curious*) adversaries. A semi-honest party follows the protocol faithfully—it does not alter, reorder, or inject messages—but may examine the transcript it accumulates to try to learn additional information about the other party’s private data. This is in contrast to *malicious* adversaries, who may deviate from the protocol arbitrarily. The semi-honest model is appropriate for many monitoring scenarios where both parties—say, a regulatory authority and a service provider—have an institutional incentive to follow the protocol, but may be curious about each other’s private data. We adopt this model throughout the thesis; the security proofs in Chapter 4 and Appendix A construct explicit simulators for each party and establish indistinguishability via hybrid arguments.

CHAPTER 3

Problem Formulation

This chapter lays the groundwork for privacy-preserving monitoring. We define the parties involved, describe the ideal setting we want to emulate, and formalize what it means for a protocol to be correct and secure. The security definitions follow the simulation paradigm introduced in Section 2.6.

3.1 Setting

The setup involves two parties: the *System* (Sys) and the *Monitor* (Mon). Think of the System as the entity being watched—it could be a medical device, an access-control system, or any process that produces a stream of data. The Monitor is the party that checks whether that data stream satisfies some rule or policy. Both parties are assumed to be *semi-honest*: they follow the agreed-upon protocol faithfully, but may afterward look at the messages they received and try to infer extra information. This is a weaker adversary model than the *malicious* setting, where parties can deviate from the protocol in arbitrary ways.

At each round the System produces a fixed-length data item. Formally, its output at round r is a bit-string $\sigma[r] \in \{0,1\}^s$, and we write $\sigma = \sigma[1], \dots, \sigma[\ell]$ for the full sequence of length ℓ. In practice, $\sigma[r]$ might encode sensor readings, event logs, or any other observable snapshot of the System at that moment. The Monitor holds a *safety specification* [KKL+02] $\phi \subseteq (\{0,1\}^s)^\omega$ (other classes of monitoring specifications could also be considered [BFFR18]). In plain terms, the specification describes the set of all data streams that are considered acceptable; if the System's data ever leaves this set, the Monitor should raise an alarm. We model the specification as a state machine with initial state $\mu[1]$. Every time a new data item $\sigma[r]$ arrives, the state is updated deterministically: $\mu[r+1] = \mathsf{nextstate}(\sigma[r], \mu[r])$. A function $\mathsf{flag}(\sigma[r], \mu[r])$ outputs 1 ("bad") if the data seen so far violates the specification, and 0 ("ok") otherwise. Both nextstate and flag are encoded together as a single Boolean circuit $\mathcal{C}$. We call the pair of circuit and initial state $(\mathcal{C}, \mu[1])$ an *initialized circuit*—it contains everything needed to begin monitoring from scratch.

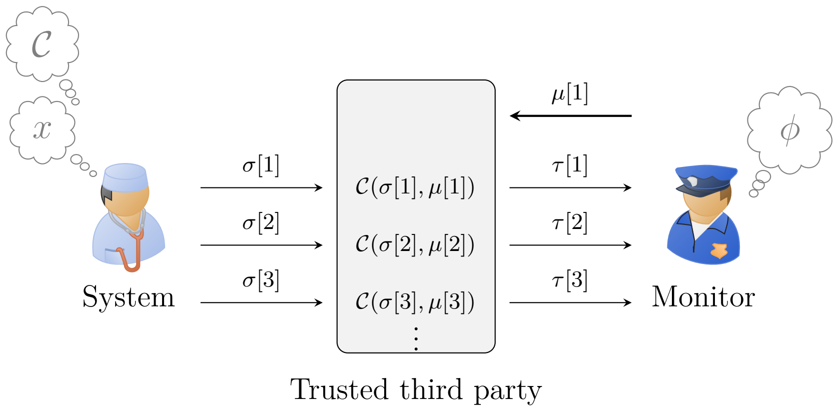


Figure 3.1: Ideal setting with a trusted third party for monitoring where the specification is *not* a secret: both parties know the circuit $\mathcal{C}$.

3.2 Ideal settings

Before we can say what it means for a protocol to be "secure," we need a gold standard to compare against. That gold standard is an *ideal* world in which a perfectly trustworthy middleman—a trusted third party (TTP)—carries out the monitoring on behalf of both parties. Because the TTP is honest by assumption, neither party can learn anything it should not, and any real protocol we design must be at least as private as this ideal world. It is worth stressing, however, that even the ideal world is not a world of zero leakage. The Monitor still learns the verdict at every round, and from the sequence of verdicts it may be able to infer something about the System's data—for example, the mere fact that a violation occurred at round r reveals that the data up to that point fell outside the specification. Similarly, both parties observe how many rounds have elapsed. What the ideal world guarantees is that these inherent leakages—the ones baked into the task of monitoring itself—are the *only* leakages. A secure protocol may not leak less than the ideal world, but it must not leak more.

We consider two variants, depending on whether the specification itself is a secret. In the *open-specification* setting, both the System and the Monitor already know the monitoring circuit $\mathcal{C}$; the Monitor only needs to tell the TTP its starting state $\mu[1]$. This models situations where the rules being checked are public—for example, a regulatory standard that both sides are aware of. In the *hidden-specification* setting, the circuit is private to the Monitor: the System does not know *what* is being checked. Here the Monitor also hands $\mathcal{C} = (\mathsf{nextstate}, \mathsf{flag})$ to the TTP. This models situations where revealing the specification would itself leak sensitive information—for instance, if knowing the exact compliance rule would allow the System to game it.

In both variants the interaction proceeds the same way. At every round r, the System sends its data $\sigma[r]$ to the TTP. The TTP evaluates the specification—computing the verdict $\tau[r] = \mathsf{flag}(\mu[r], \sigma[r])$—and sends the result back to the Monitor, while internally updating its state to $\mu[r+1] = \mathsf{nextstate}(\mu[r], \sigma[r])$. The System never sees the verdict, and the Monitor never sees more than the single yes/no answer. These two settings are illustrated in Figs. 3.1 and 3.2.

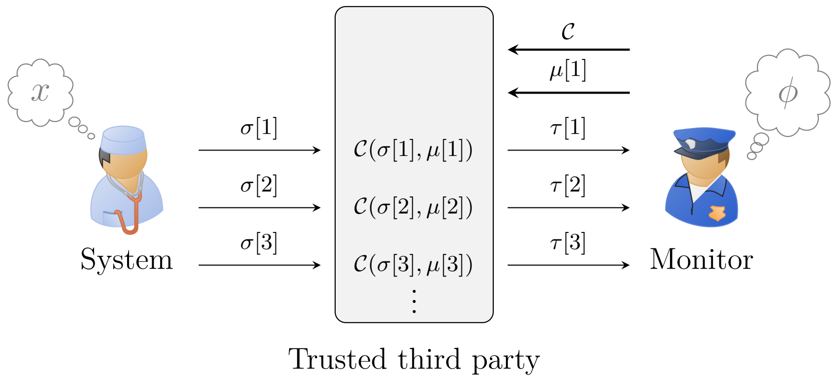


Figure 3.2: Ideal setting with a trusted third party for monitoring where the specification is a secret.

3.3 Monitoring protocol

With the ideal world in place, we now describe what a real-world monitoring protocol looks like. A monitoring protocol is a pair of instructions $\pi = (\pi_{\mathsf{Sys}}, \pi_{\mathsf{Mon}})$, one for each party, telling the System and the Monitor exactly what to compute and what messages to send at each step. The protocol begins with a *setup phase* (round 0), during which the parties exchange any material they need before monitoring begins—for example, cryptographic keys or garbled circuits. After setup, monitoring proceeds in rounds: in every subsequent round r, the System sends a single message to the Monitor, and the Monitor replies with either "proceed" (the data seen so far is acceptable) or "terminate" (a violation has been detected). We restrict each round to a single message because monitoring should remain lightweight and avoid multiple back-and-forth exchanges.

We note that the one-message-per-round restriction is, to some extent, an idealization. In practice, protocols that rely on modular-exponentiation-based primitives can produce messages that are tens of megabytes long for even a modest circuit (see the message-size measurements in Fig. 5.1). When a single "message" already dwarfs most network MTUs by orders of magnitude, the distinction between one logical message and several is overshadowed by the sheer cost of producing and transmitting the data. The restriction is therefore best understood as a convenient abstraction for counting communication rounds, not as a practical engineering constraint.

Correctness. A protocol is correct if, regardless of the specification or the data stream, the Monitor ends up with the same verdict it would have obtained in the ideal world. In other words, removing the TTP and running the real protocol instead should not change the answer.

Definition 3.1 (Correctness) *A protocol $\pi = (\pi_{\mathsf{Sys}}, \pi_{\mathsf{Mon}})$ is a* correct monitoring protocol *if, for every initialized circuit $(\mathcal{C}, \mu[1])$, observable sequence σ, and security parameter n, the Monitor's output in an execution of π agrees with the ideal-setting output on the same inputs with overwhelming probability, i.e., with probability at least $1 - n^{-d}$ for every fixed $d \in \mathbb{N}$.*

The "overwhelming probability" qualifier accounts for the fact that the cryptographic building blocks we use are probabilistic: there is always a vanishingly small chance of failure, but that chance shrinks faster than any polynomial as the security parameter grows.

3.4 Security

Correctness guarantees that the protocol gives the right answer. Security guarantees that it does not give away anything *else.* The core idea is straightforward: if a party could learn something by participating in the real protocol that it could not have learned in the ideal world, then the protocol is insecure.

To make this precise, we use the simulation paradigm (introduced in Section 2.6). For each party, we compare two transcripts:

- The **real transcript**: everything the party actually sees during a run of the protocol—its own input, the random coins it flipped, and every message it received.
- The **simulated transcript**: a fake transcript manufactured by an efficient algorithm (the *simulator*) that has access only to the party's own input and the output it would receive in the ideal world.

If no efficient algorithm can reliably tell the real transcript from the simulated one, then the protocol is secure: whatever a party might try to extract from the real messages, it could equally well have fabricated on its own from the ideal-world output alone. Because the protocols are randomized, we are really comparing *distributions* over transcripts, not individual transcripts. In the hidden-specification setting, this reasoning also protects the Monitor's circuit: the TTP evaluates the specification without ever showing it to the System, and a real protocol is secure if the System's real transcript is indistinguishable from one produced without knowledge of the circuit.

Real view. Formally, the *view of the Monitor*, $\mathtt{view}^{\pi}_{\mathsf{Mon}}(x_M, \sigma, 1^n)$, consists of the Monitor's input x_M, the random bits it used, and the messages $m_1, \ldots, m_\ell$ it received during protocol π. This is the totality of information available to the Monitor.

Ideal view. The *ideal view*, $\mathcal{S}^{\textsc{Ideal}}_{\mathsf{Mon},\pi}(x_M, \sigma, 1^n)$, is a transcript produced by a simulator $\mathcal{S}_{\mathsf{Mon}}$ (a probabilistic polynomial-time machine) that sees only the Monitor's inputs and ideal-setting outputs—crucially, it never sees the System's private data. When the protocol is clear from context we drop π from the subscript. The views $\mathtt{view}^{\pi}_{\mathsf{Sys}}$ and $\mathcal{S}^{\textsc{Ideal}}_{\mathsf{Sys}}$ of the System are defined analogously.

We use the symbol $\overset{c}{\equiv}$ to denote computational indistinguishability [BM82]. The security notion splits naturally into two definitions, one for each setting.

Open specification. When both parties already know the circuit, security means that neither party's real transcript reveals anything beyond what the ideal world would have provided. The simulator for each party must be able to produce a convincing fake transcript using only that party's input and output.

Definition 3.2 (Security without specification hiding) *A protocol* $\pi = (\pi_{\mathit{Sys}}, \pi_{\mathit{Mon}})$ *is a* secure monitoring protocol without specification hiding *for a circuit* $\mathcal{C}$ *if there exist* *probabilistic polynomial-time simulators $\mathcal{S}_{\mathsf{Mon}}$ and $\mathcal{S}_{\mathsf{Sys}}$ such that, for all initial states $\mu[1]$, observable sequences σ, and security parameters $n \in \mathbb{N}$:*

$$\left\{\mathcal{S}^{\textsc{Ideal}}_{\mathsf{Mon}}(\mu[1], \sigma, 1^n)\right\}_{\mu[1],\sigma} \overset{c}{\equiv} \{\boldsymbol{view}^{\pi}_{\mathsf{Mon}}(\mu[1], \sigma, 1^n)\}_{\mu[1],\sigma},$$
$$\left\{\mathcal{S}^{\textsc{Ideal}}_{\mathsf{Sys}}(\mu[1], \sigma, 1^n)\right\}_{\mu[1],\sigma} \overset{c}{\equiv} \left\{\boldsymbol{view}^{\pi}_{\mathsf{Sys}}(\mu[1], \sigma, 1^n)\right\}_{\mu[1],\sigma}.$$

Hidden specification. When the specification is secret, security must additionally guarantee that the System cannot learn anything about the circuit. Note that the definition below quantifies over *all* possible circuits, not just a specific one—the protocol must protect every specification equally well.

Definition 3.3 (Security with specification hiding) *A protocol $\pi = (\pi_{\mathsf{Sys}}, \pi_{\mathsf{Mon}})$ is a* secure monitoring protocol with specification hiding *if there exist probabilistic polynomial-time simulators $\mathcal{S}_{\mathsf{Mon}}$ and $\mathcal{S}_{\mathsf{Sys}}$ such that, for all initialized circuits $(\mathcal{C}, \mu[1])$, observable sequences σ, and security parameters $n \in \mathbb{N}$:*

$$\left\{\mathcal{S}^{\textsc{Ideal}}_{\mathsf{Mon}}\Big((\mathcal{C}, \mu[1]), \sigma, 1^n\Big)\right\}_{(\mathcal{C},\mu[1]),\sigma} \overset{c}{\equiv} \left\{\boldsymbol{view}^{\pi}_{\mathsf{Mon}}\Big((\mathcal{C}, \mu[1]), \sigma, 1^n\Big)\right\}_{(\mathcal{C},\mu[1]),\sigma},$$
$$\left\{\mathcal{S}^{\textsc{Ideal}}_{\mathsf{Sys}}\Big((\mathcal{C}, \mu[1]), \sigma, 1^n\Big)\right\}_{(\mathcal{C},\mu[1]),\sigma} \overset{c}{\equiv} \left\{\boldsymbol{view}^{\pi}_{\mathsf{Sys}}\Big((\mathcal{C}, \mu[1]), \sigma, 1^n\Big)\right\}_{(\mathcal{C},\mu[1]),\sigma}.$$

CHAPTER 4

The Protocol

In this chapter, we present our privacy-preserving monitoring protocol. We begin by restating the monitoring problem in the language of reactive two-party computation (Section 4.1), then briefly discuss the design space (Section 4.2). The protocol itself is developed in two steps: first for the setting where the specification circuit is known to both parties (Section 4.4), and then for the setting where the circuit must also be kept secret from the System (Section 4.5).

4.1 Reactive monitoring as secure computation

Monitoring differs from textbook secure computation in one important way: it is *stateful*. The System and the Monitor do not evaluate a function once; they evaluate it *every round*, and each evaluation depends on state produced by the previous one. A protocol that achieves this is what we call a *reactive secure monitoring protocol*.

Concretely, recall from Chapter 3 that the monitoring functionality is determined by a circuit $\mathcal{C}$ (encoding both the flag function flag and the state-update function $\mathsf{nextstate}$), an initial monitor state $\mu[1]$, and a stream of System inputs $\sigma[1], \sigma[2], \ldots$ At every round r the functionality produces the verdict $\tau[r] = \mathsf{flag}(\sigma[r], \mu[r])$ and internally advances its state to $\mu[r+1] = \mathsf{nextstate}(\sigma[r], \mu[r])$. Neither party ever sees the intermediate states.

A protocol π *securely realizes* this functionality if it satisfies four properties:

1. **Correctness.** In every round the Monitor obtains the same verdict $\tau[r] = \mathsf{flag}(\sigma[r], \mu[r])$ it would have received from the ideal-world TTP, with overwhelming probability (see Definition 3.1).

2. **System privacy.** A simulator $\mathcal{S}_{\mathsf{Mon}}$, given only the Monitor's own inputs $(\mathcal{C}, \mu[1])$ and the verdicts $\tau[1], \ldots, \tau[\ell]$, can produce a transcript indistinguishable from the Monitor's real view. The Monitor learns nothing about $\sigma[r]$ beyond what the verdict implies.

3. **Monitor privacy.** A simulator $\mathcal{S}_{\mathsf{Sys}}$, given only the System's inputs and (in the open-specification setting) the circuit $\mathcal{C}$, can produce a transcript indistinguishable from the System's real view. The System learns nothing about $\mu[r]$ for any r.

4. **Specification privacy (hidden-specification setting only).** The System's simulator needs only the circuit *size* c, not $\mathcal{C}$ itself. This hides the circuit topology—and hence the specification—from the System.

After an initial setup phase, each subsequent round consists of one message from the System to the Monitor, plus a single-bit response (PROCEED or TERMINATE). As noted in Section 3.3, this "single message" is largely a bookkeeping convenience: in the hidden-specification step every wire label is a group element produced by bare modular exponentiation, with no batching or compression, so a single round's payload can run to tens of megabytes (see Fig. 5.1). Many MPC protocols get by with $O(\text{depth}(\mathcal{C}))$ or $O(\log c)$ rounds of interaction, which would be perfectly acceptable here; we impose the single-message constraint because it simplifies the security argument and because, at the scale of our experiments, the bottleneck is computation (the modular exponentiations themselves) rather than the number of network round-trips.

Our protocol provides two instantiations with increasing privacy:

1. **Open-specification step** (Section 4.4). Both parties know $\mathcal{C}$. The key idea is a reactive adaptation of Yao's garbled circuits in which wire labels for the monitor state are reused across rounds to maintain state without revealing it.
2. **Hidden-specification step** (Section 4.5). Only the Monitor knows $\mathcal{C}$; the System knows only the gate count c. The additional tool is DDH-based topology hiding, following [LWY22].

4.2 Design space

Before describing the protocol, we briefly justify the three main design choices.

Garbled circuits over FHE and secret sharing. As discussed in Chapter 6, the FHE-based protocols of Banno et al. [BMM+22] and Waga et al. [WMS+24] operate in a different trust model, and their evaluation cost is linear in the DFA size of the specification—exponentially larger than the circuit representation for specifications with large state spaces (see Table 5.1). Secret-sharing-based MPC requires three or more parties [GMW87] and does not directly apply to our two-party setting. Two-party protocols for reactive functionalities are standard [HL10], but typically require multiple message exchanges per round (at least three for OT-based constructions [CSW20]). Garbled circuits give us a single-message-per-round design after the initial setup, with the Monitor evaluating non-interactively.

Reactive label reuse. A naïve approach would run an independent Yao instance per round. That forces oblivious transfer (OT) in every round (at least three messages) and requires explicitly transferring the monitor state—either in the clear (breaking privacy) or via further OT. Our protocol avoids both problems by *reusing labels*: the wire labels assigned to the monitor-state output wires of round r become the input labels for round $r+1$. The Monitor already holds the correct labels from evaluating the previous round, so no OT is needed after round 1, and it never learns the actual bit values of $\mu[r]$. The security of this reuse relies on a hybrid argument detailed in Appendix A.

DDH-based topology hiding. In the hidden-specification setting, the circuit topology must be concealed. Universal circuits [Val76] would impose a $\Theta(\log^2 c)$ blowup in circuit size. Homomorphic-encryption-based PFE [KM11] requires multi-round interaction during setup. The DDH-based approach of Liu et al. [LWY22] avoids both: the Monitor assigns random group elements as *base labels* to feed-out wires and exponentiates them to produce feed-in labels; by the DDH assumption (Lemma 2.1), the System cannot tell which feed-out wire connects to which feed-in wire. Base labels are set once during setup and reused across all rounds, amortizing the cost. Our extension beyond [LWY22] adds reactive state: exponents from round r carry over to round $r{+}1$, enabling stateful computation while preserving topology hiding.

4.3 Circuit conventions and notation

Every specification is compiled into a Boolean circuit $\mathcal{C}$ before the protocol begins. We fix the following conventions.

- *Size.* The circuit contains c gates, $s{+}m$ input wires (the first m for the Monitor's state, the remaining s for the System's data), and $m{+}1$ output wires (the first m feeding back the updated state, the last carrying the verdict bit $\tau[r]$).
- *Gates.* Every gate is a NAND gate with exactly two *feed-in* wires (left and right) and one *feed-out* wire. We write $G_1, \ldots, G_c$ for the gates; gate G_j has feed-in wires ι_{2j-1} (left) and ι_{2j} (right), and feed-out wire ω_{m+s+j}.
- *Wires.* A *feed-out* wire exits a gate (or serves as a circuit input); a *feed-in* wire enters a gate. Every feed-in wire is connected to exactly one feed-out wire, but a feed-out wire may fan out to several feed-in wires. There are $I = 2c$ feed-in wires and $c{+}s{+}m$ feed-out wires. The first $m{+}s$ feed-out wires $\omega_1, \ldots, \omega_{m+s}$ are the circuit inputs; among these, $\omega_1, \ldots, \omega_m$ carry the Monitor's state and $\omega_{m+1}, \ldots, \omega_{m+s}$ carry the System's data. The $O = c{+}s{-}1$ non-output feed-out wires plus the $m{+}1$ output wires fill out the rest: the output wires are $\omega_{O+1}, \ldots, \omega_{O+m}$ (feedback state) and ω_{O+m+1} (the flag bit).
- *Fan-out restriction.* We assume that output wires of the circuit do not connect back to any feed-in wires.

See the black labels in Fig. 4.1 for a pictorial summary.

4.4 Step 1: Monitoring with an open specification

We begin with the conceptually simpler case: both parties already know the specification circuit $\mathcal{C}$. The internal state computed by the circuit is still kept secret. The protocol is a reactive adaptation of Yao's garbled circuits (see Section 2.3); the central idea is that wire labels for the monitor state are reused from one round to the next, avoiding additional oblivious transfer after the first round.

Setup (round 0). Both parties agree on the circuit $\mathcal{C}$ and a security parameter n. The Monitor holds an m-bit initial state $\mu[1]$. The System and Monitor execute a 1-out-of-2 oblivious transfer for each of the m state bits: the System acts as sender with random label pairs $(w_i^0[1], w_i^1[1])$, and the Monitor acts as chooser with choice bit $\mu_i[1]$. At the end of this step, the Monitor holds the labels $w_i^{\mu_i[1]}[1]$ for $i \in \{1, \ldots, m\}$ without the System learning $\mu[1]$, and without the Monitor learning the other labels.

Round r ($r \geqslant 1$). The System holds the current data item $\sigma[r] \in \{0,1\}^s$. It proceeds as follows.

1. **Generate feed-out labels.** For every feed-out wire ω_i with $i \in \{m{+}1, \ldots, c{+}s{+}m\}$, the System draws two fresh random strings $w_i^0[r]$ and $w_i^1[r]$.

2. **Reuse state labels.** For the Monitor-state input wires $\omega_1, \ldots, \omega_m$:

   - In round 1, the labels $w_i^0[1], w_i^1[1]$ were already generated during setup.
   - In rounds $r > 1$, the System reuses the labels of the feedback wires from the previous round: $w_i^b[r] \leftarrow w_{O+i}^b[r{-}1]$ for $b \in \{0,1\}$ and $i \in \{1, \ldots, m\}$.

3. **Assign feed-in labels.** Each feed-in wire ι_i inherits the labels of the feed-out wire it is connected to: if ω_j connects to ι_i, then $u_i^b[r] = w_j^b[r]$.

4. **Garble and send.** For each gate G_j, the System computes the garbled gate

$$\mathbf{encGG}_j[r] = \mathrm{encYao}_{G_j} \begin{pmatrix} \left[u_{2j-1}^0[r], u_{2j-1}^1[r]\right], \\ \left[u_{2j}^0[r], u_{2j}^1[r]\right], \\ \left[w_{m+s+j}^0[r], w_{m+s+j}^1[r]\right] \end{pmatrix}$$

   and sends to the Monitor: all garbled gates $\mathbf{encGG}_j[r]$, the System's input labels $w_{m+i}^{\sigma_i[r]}[r]$ for $i \in \{1, \ldots, s\}$, and both flag labels $w_{O+m+1}^0[r]$, $w_{O+m+1}^1[r]$.

5. **Evaluate.** The Monitor ungarbles the circuit bottom-up using the monitor-state labels it already holds (from setup or the previous round) together with the System's input labels. After evaluation, the Monitor holds one label per output wire: the state-feedback labels $w_{O+i}^{\mu_i[r+1]}[r]$ (which become the input labels for round $r{+}1$) and the flag label, which it compares against the two flag labels to determine $\tau[r]$.

The following theorem shows that the open-specification step is correct and secure, assuming that the encryption is secure under the chosen plaintext attack (CPA) [Bir11].

Theorem 4.1 *Under CPA-secure encryption and semi-honest oblivious transfer, the open-specification step is a correct and secure monitoring protocol (without specification hiding) for any polynomial number of rounds.*

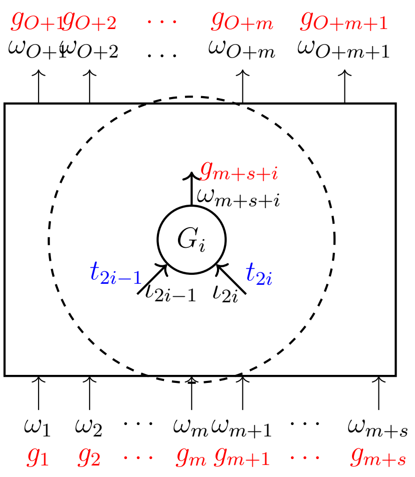

g_{O+1} g_{O+2} $\cdots$ g_{O+m} g_{O+m+1}
ω_{O+1} ω_{O+2} $\ldots$ ω_{O+m} ω_{O+m+1}
g_{m+s+i}
ω_{m+s+i}
G_i
t_{2i-1}
ι_{2i-1}
ι_{2i}
t_{2i}
ω_1 ω_2 $\cdots$ ω_m ω_{m+1} $\cdots$ ω_{m+s}
g_1 g_2 $\cdots$ g_m g_{m+1} $\cdots$ g_{m+s}

Figure 4.1: Base labels: feed-out wires.

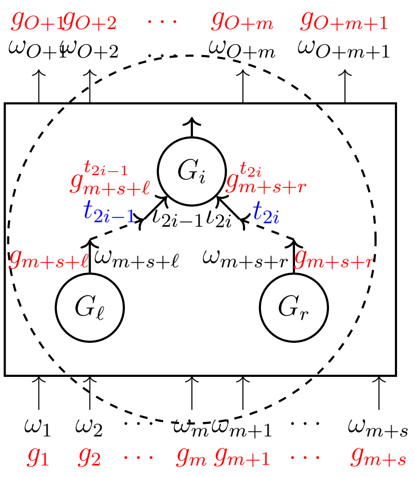

g_{O+1} g_{O+2} $\cdots$ g_{O+m} g_{O+m+1}
ω_{O+1} ω_{O+2} $\ldots$ ω_{O+m} ω_{O+m+1}
G_i
$g_{m+s+\ell}^{t_{2i-1}}$
$g_{m+s+r}^{t_{2i}}$
t_{2i-1}
ι_{2i-1}
ι_{2i}
t_{2i}
$g_{m+s+\ell}$ $\omega_{m+s+\ell}$
ω_{m+s+r} g_{m+s+r}
G_ℓ
G_r
ω_1 ω_2 $\cdots$ ω_m ω_{m+1} $\cdots$ ω_{m+s}
g_1 g_2 $\cdots$ g_m g_{m+1} $\cdots$ g_{m+s}

Figure 4.2: Base labels: feed-in wires.

4.5 Step 2: Extending to hidden specifications

In the open-specification step, the System knows the full circuit and can wire up the labels itself. When the circuit must also be hidden, the System no longer knows which feed-out wire connects to which feed-in wire. The challenge is to let the System prepare wire labels for garbling *without* learning the circuit topology. We solve this by having the Monitor provide *base labels*—random group elements associated with wires—from which the System derives the round-specific labels by exponentiation. The DDH assumption (Assumption 2.1) ensures that the exponentiated labels leak nothing about the wiring.

Since only the gate *function* matters (and every gate is NAND), what must be hidden is the *permutation* mapping feed-out wires to feed-in wires. The Monitor obfuscates this permutation using a cyclic group $\mathbb{G}$ of prime order q in which DDH holds.

Setup (round 0). The Monitor and System agree on the gate count c, a security parameter n, and the group $\mathbb{G}$.

1. **Feed-out base labels** (see Fig. 4.1). The Monitor picks a random group element $g_i \overset{\$}{\leftarrow} \mathbb{G}$ for each non-output feed-out wire ω_i ($i \in \{1, \ldots, O\}$) and sends the list

$[g_1, \ldots, g_O]$ to the System. The m feedback output wires $\omega_{O+1}, \ldots, \omega_{O+m}$ reuse the same base labels $g_1, \ldots, g_m$. The final output wire ω_{O+m+1} (the flag bit) receives no base label during setup.

2. **Feed-in base labels** (see Figs. 4.1 and 4.2). For each feed-in wire ι_i ($i \in \{1, \ldots, I\}$), the Monitor draws a random exponent $t_i \xleftarrow{\$} \mathbb{Z}_q$. Let $\pi(i) = j$ when feed-out wire ω_j connects to feed-in wire ι_i. The Monitor computes $\ell_i = g_{\pi(i)}^{t_i}$ and sends the list $[\ell_1, \ldots, \ell_I]$ to the System.
3. **Oblivious transfer.** Exactly as in the open-specification step: the System sends random label pairs for the m state wires, and the Monitor uses OT to obtain the labels corresponding to $\mu[1]$.

The System now holds three base labels per gate (two feed-in, one feed-out) but cannot tell which feed-out wire each feed-in label was derived from: by the DDH assumption (Lemma 2.1), the tuple $(\ell_1, \ldots, \ell_I)$ is indistinguishable from I uniformly random group elements, regardless of the permutation π.

Round r ($r \geqslant 1$). The System holds $\sigma[r]$ and proceeds as follows.

1. **Choose round exponents.** In round 1 the System picks distinct $\alpha^0[1], \alpha^1[1] \xleftarrow{\$} \mathbb{Z}_q$. In subsequent rounds, it reuses the output exponents from the previous round: $\alpha^b[r] \leftarrow \beta^b[r-1]$.
2. **Derive feed-out labels.** For each non-output feed-out wire ω_j ($j \in \{1, \ldots, O\}$): $w_j^b[r] = g_j^{\alpha^b[r]}$. The System also draws fresh $\beta^0[r], \beta^1[r] \xleftarrow{\$} \mathbb{Z}_q$ and sets the feedback-output labels $w_{O+j}^b[r] = g_j^{\beta^b[r]}$ for $j \in \{1, \ldots, m\}$. For the flag output wire ω_{O+m+1}, it picks $w_{O+m+1}^0[r], w_{O+m+1}^1[r] \xleftarrow{\$} \mathbb{G}$ (random group elements, not derived from a base label).
3. **Derive feed-in labels.** For each feed-in wire ι_i: $u_i^b[r] = \ell_i^{\alpha^b[r]}$.
4. **Garble and send.** Same as in the open-specification step: garble every gate G_j using the feed-in labels $(u_{2j-1}^b[r], u_{2j}^b[r])$ and feed-out label $w_{m+s+j}^b[r]$, and send all garbled gates, the System's input labels, and both flag labels.
5. **Evaluate.** The Monitor ungarbles bottom-up. After obtaining the feed-out label $w_{m+s+j}^{b^*}[r] = g_{m+s+j}^{\alpha^{b^*}[r]}$ from gate G_j, the Monitor raises it to the exponent t_i of the downstream feed-in wire ι_i to recover the feed-in label: $(g_{m+s+j}^{\alpha^{b^*}[r]})^{t_i} = (g_{m+s+j}^{t_i})^{\alpha^{b^*}[r]} = \ell_i^{\alpha^{b^*}[r]} = u_i^{b^*}[r]$. In other words, the Monitor can propagate labels through the circuit by exponentiating, without the System ever revealing the wiring.

Because $\alpha^b[r+1] = \beta^b[r]$ and the feedback base labels coincide with the state-input base labels $(g_1, \ldots, g_m)$, the label-reuse mechanism from the open-specification step carries over: the Monitor's state-output labels from round r serve directly as state-input labels for round $r+1$, preserving reactivity.

Both message sizes and computation time are linear in $c+s+m$ (assuming the security parameter is a constant).

Theorem 4.2 *Under CPA-secure encryption, the DDH assumption on $\mathbb{G}$, and semi-honest oblivious transfer, the hidden-specification step is a correct and secure monitoring protocol with specification hiding for any polynomial number of rounds.*

The proof constructs intermediate indistinguishable transcripts by progressively replacing elements with random group elements. A key ingredient is a corollary of Lemma 2.1 from Naor and Reingold [NR04]; the same lemma is used in the work of Liu, Wang, and Yu [LWY22, Lemma 3]. The full proof appears in Appendix A.

CHAPTER 5

Experimental Evaluation

To test the feasibility of our protocol for monitoring, we developed an experimental C++ prototype[1] and performed experiments to evaluate the following key questions.

1. How do the measured requirements of both steps of the protocol change under varying security parameters ($n \geqslant 1024$ would be industrial standard),
   a) in terms of time taken per round? (see Figs. 5.2 and 5.3)
   b) in terms of message sizes per round? (see Fig. 5.1)
2. When the specification size (c) is fixed, but the size of System observables data (s) is large, how much do these measurements change for the hidden-specification step? (see Figs. 5.5 and 5.6)

5.1 Experiment scenarios

To answer these questions, we consider two experiment scenarios.

5.1.1 Access control system (ACS)

In this scenario, we consider an access control system for an office building, where two types of employees, namely types A and B, enter or exit the building through a set of external doors. The ACS tracks the numbers of entries and exits for each type of employee and for each door. The Monitor keeps two variables $\texttt{cnt}_A$ and $\texttt{cnt}_B$, where $\texttt{cnt}_e$ denotes count of type-e employees currently in the building. At round r, for door i of the building, the Monitor receives input from the ACS, structured as follows: $\texttt{entered}^i_A[r]$, $\texttt{exited}^i_A[r]$, $\texttt{entered}^i_B[r]$, $\texttt{exited}^i_B[r]$. There are N doors, hence N such quadruples. Each $\texttt{entered}^i_e$ denotes the number of type-e employees who have entered through door i of the building *since* the last round; `exited` values have a similar definition.

[1]This prototype is accessible online at `https://github.com/mahykari/ppm`.

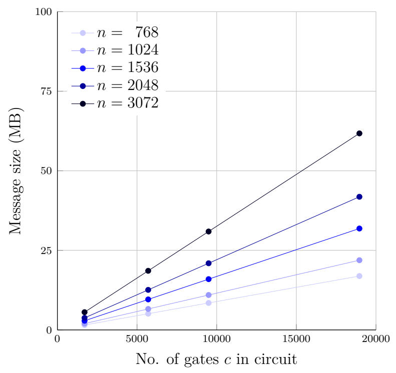


Figure 5.1: Message size vs. gate count for the Locks scenario: hidden-specification step.

Specification. Our specification for this system requires that the number of type A employees currently in the building is never less than type B employees; concretely, the flag function of the register machine activates if and only if $\mathtt{cnt}_A < \mathtt{cnt}_B$. The value of $\mathtt{cnt}_e$ updates with the following rule: $\mathtt{cnt}_e \leftarrow \mathtt{cnt}_e + \sum_i (\mathtt{entered}_e^i[r]) - \sum_i (\mathtt{exited}_e^i[r])$. Each number is an unsigned integer of fixed bit-width W. The monitor only keeps track of employee count per type and needs $2W$ bits for Monitor state, whereas the input of the ACS to each round takes $4NW$ bits.

To answer Question 2, we create another case where the ACS keeps track of the internal doors in the building as well (e.g., individual offices). We use N' to denote the number of internal doors. In this case, each ACS update takes $4(N+N')W$ bits, but the specification is still expressed over only $4NW$ bits.

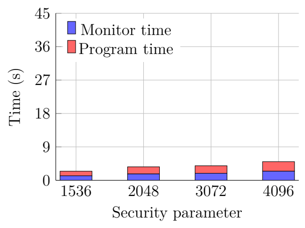
5.1.2 Locks of a parallel program

In this scenario, every lock has at most one "user" at any given time. Each lock provides a `lock()` and `unlock()` interface, and all the locks are initially "unlocked." The Monitor keeps track of all lock states $\mathtt{lock}_1, \ldots, \mathtt{lock}_N$, where N denotes the total number of locks in the system. Each $\mathtt{lock}_i$ can have value `LOCK` or `UNLOCK`. The Monitor, at round t, receives input from the lock system, structured as follows: $\mathtt{request}_1[t], \ldots, \mathtt{request}_N[t]$, where N denotes the number of locks, and each `request` can have value `LOCK`, `UNLOCK`, or `SKIP`.

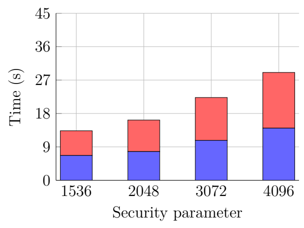
Specification. The specification for this scenario requires that `lock()` or `unlock()` is never called twice in a row for any of the locks in the system. Concretely, at round r, we have: $\mathsf{flag} \iff \bigvee_i (\mathtt{request}_i[r] = \mathtt{lock}_i)$. To update $\mathtt{lock}_i$, we simply replace its value with $\mathtt{request}_i[r]$. As the Monitor keeps track of all locks, it needs N bits for its state. Each `request` takes 2 bits to represent, and hence each update to the Monitor needs $2N$ bits.

5.2 Experiment results

Execution time. To answer Question 1a, we plot the breakdown of execution times for the open-specification step in Fig. 5.2 (scenario ACS), and in Figs. 5.3 and 5.4 (scenarios ACS and locks) for the hidden-specification step. For these cases, ACS only updates on external doors (i.e., $N' = 0$), and we select values for N among $\{10, 30\}$ and W

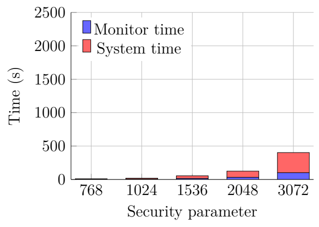


(a) $s = 640, c = 7127$

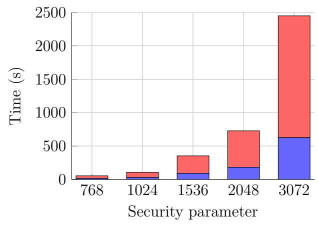


(b) $s = 3840, c = 42145$

Figure 5.2: Timings for the ACS scenario: open-specification step.

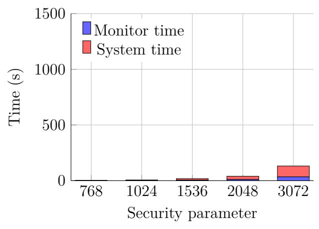


(a) $s = 640, c = 7127$

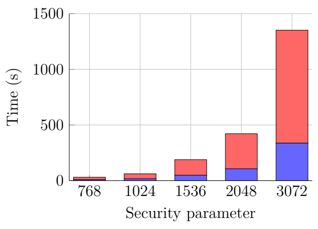


(b) $s = 3840, c = 42145$

Figure 5.3: Timings for the ACS scenario: hidden-specification step.



(a) $s = 200, c = 1705$

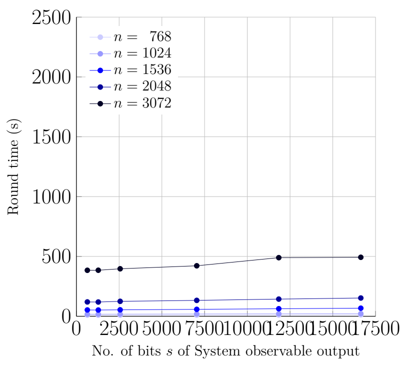


(b) $s = 2000, c = 18959$

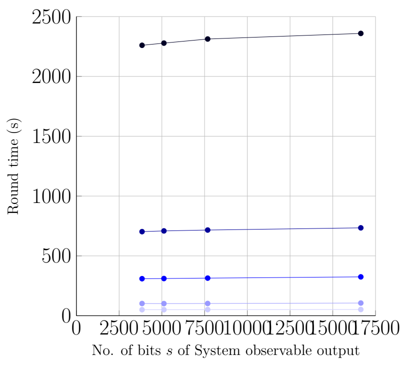
Figure 5.4: Timings for the Locks scenario: hidden-specification step.

among $\{16, 32\}$ to give us 4 different instances. For the Locks scenario, we consider the values $\{100, 300, 500, 1000\}$ for the parameter N. We show the smallest and largest instances; the intermediate parameter settings exhibit the same patterns in all cases. The open-specification step relies only on random string generation instead of group operations and has symmetric garbling and ungarbling phases, and the round times are lower and more uniformly distributed between System and Monitor. In the hidden-specification step, time is mostly spent on the System side, since the System performs more group operations than the Monitor in the garbling process; as the security parameter increases, this difference becomes more visible. The superlinear growth of round times also conforms with the complexity of group operations.

Message size. For Question 1b, we plot the breakdown of message sizes for the hidden-specification step in Fig. 5.1, using the Locks scenario with $N \in \{100, 300, 500, 1000\}$. The open-specification step exhibits the same linear correlation between message size and gate count; we omit its plot since it is qualitatively identical. As expected, message sizes grow linearly with the number of gates in both steps of the protocol. In the open-specification step, message sizes are proportional to the label length, which depends on the security parameter n.

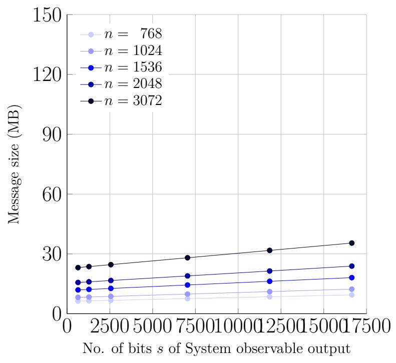


(a) $c = 7127$

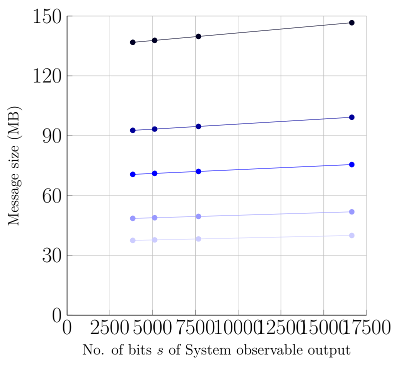


(b) $c = 42145$

Figure 5.5: Timings for the ACS scenario, hidden-specification step: fixed specifications, increasing System observable output size.



(a) $c = 7127$



(b) $c = 42145$

Figure 5.6: Message size for the ACS scenario: fixed specifications, increasing System observable output size.

Benchmark	**Banno et al.**	
	DFA Size	**Time**
MOD ($m = 500$)	500	~0.002 s
BGM (ψ_2), REVERSE	2885376	~24 s
BGM (ψ_2), BLOCK	11126	0.182 s
BGM (ψ_4), REVERSE	N/A	time-out
BGM (ψ_4), BLOCK	7026	0.049 s
ACS (Fig. 5.3a)	$\geqslant 2^{32}$	10 hours (estimated)
LOCKS (Fig. 5.4a)	$\geqslant 2^{300}$	10^6 years (estimated)

Table 5.1: Monitoring latency of a single event in the protocol of Banno et al. [BMM+22].

Scaling with System input. Finally, for Question 2, we plot Figs. 5.5 and 5.6, which show how round time increases with increasing sizes of the System's observable data, while keeping "relevant" System input size fixed and again consider parameters of N and W from $\{10, 30\}$ and $\{16, 32\}$, respectively. However, to inflate System input size, we use increasing values for the parameter N' (number of internal doors). Observe that the circuit size remains the same even while the specification considers an increasing number of external doors. With the parameters we have used in our experiments, as System input increases, the time taken per round increases only marginally, as a result of the garbling phase's dominance in execution time (Fig. 5.5). The increase in message size is more observable, as one key is sent per each bit of the input (Fig. 5.6). The omitted intermediate circuit sizes ($c = 14542$ and $c = 20618$) show the same trends. However, even this growth is less steep than the growth of the message size when both circuit size and System observables are scaled proportionally as seen in Fig. 5.1.

We remark that all the monitoring latencies reported are for a single event rather than a trace.

5.3 Comparison with related work

As discussed in Chapter 6, the setting considered by Banno et al. [BMM+22] (and therefore also Waga et al. [WMS+24]) is orthogonal to ours—each protocol is tailored to its specific context and not directly applicable to the other. However, some of their specifications they describe might be relevant to our setting, and vice versa. Therefore, we evaluated our protocol on several of their specifications and, conversely, estimated the performance of their protocol on the ACS and locks scenarios from our work.

Banno et al. consider two scenarios: a DFA that counts the number of 1s in its input modulus m (MOD) and Blood Glucose Monitoring (BGM). They also have two protocols REVERSE and BLOCK. We implemented the specifications for each scenario with the *highest* reported monitoring latencies in their work for either REVERSE or BLOCK, with the same security parameter as Banno et al. ($n = 1024$). For these values, our protocol takes time that is in the order of magnitude of 100 milliseconds. We remark that their experiments were run on Intel Xeon Silver (32 cores and 64 threads), a superior hardware to ours. These times are summarized in Table 5.1.

To test our specifications against their protocol, we also estimated the time taken by their protocol on the ACS and Locks scenarios we designed. However, our scenarios cannot be

Benchmark	**Hidden-Specification Step**	
	Circuit Size	**Time for $n = 1024$**
MOD ($m = 500$)	146	0.30 s
BGM (ψ_2)	118	0.24 s
BGM (ψ_4)	89	0.23 s
ACS (Fig. 5.3a)	7127	18.94 s
LOCKS (Fig. 5.4a)	5700	19.96 s

Table 5.2: Monitoring latency of a single event in the hidden-specification step.

directly specified as LTL expressions over the observable output alone to be used directly as an input into their protocol, since our description language is more expressive than LTL specification. Since their protocol converts LTL specifications into DFAs, we considered the size of the smallest possible DFAs accepting the ACS and Locks scenarios to estimate the running time. We then extrapolated the monitoring latency on ACS and the locks scenario from the fact that their protocol is linear in the size of the DFA, and scaled from the other DFA instances provided in their work. These times are summarized in Table 5.2.

For both Tables 5.1 and 5.2, we use the same notation as their paper to refer to the LTL specifications (ψ_2 and ψ_4) in their work, obtained originally from related work on runtime verification for artificial pancreas [CFMS15].

The sizes of the formulas considered in the experiments conducted by Waga et al. [WMS+24] are similar in terms of DFA sizes to those considered by Banno et al., and we therefore only restrict our comparison to the work of Banno et al. It is reasonable to expect that the monitoring latency of both Waga et al.'s protocols on these specifications would also be in the same order of magnitude.

Remark. Unlike the hidden-specification step, which requires circuits with a single type of binary gate, the open-specification step supports multiple gate types, enabling the use of optimized garbling techniques [ZRE15, App16]. Further, even the hidden-specification step can be made more efficient if the number of gates of each type is known to all, while still ensuring the circuit topology is not known. This opens potential avenues for optimization. Additionally, our experiments showed a 50% average reduction in circuit size when ABC utilized all basic gates instead of only NAND gates, which could contribute to further speed-up.

5.4 Implementation details

We implemented our protocol as a C++ prototype, available at `https://github.com/mahykari/ppm`. This section describes the software architecture, the circuit synthesis pipeline, the cryptographic implementation choices, and the engineering decisions that shaped the system.

5.4.1 Software architecture

The implementation models the System and Monitor as separate operating-system processes that communicate via message passing. Internally, each party is implemented as a *communicating transition system*—a state machine whose states can perform read, write, send, and receive operations. This architecture is implemented through an abstract `State` class with two key methods: `isSend()` and `isRecv()`, which indicate whether the current state requires sending or receiving a message, and `next()`, which computes the successor state.

Each protocol variant is encapsulated in its own C++ namespace: `Y` for the open-specification step (Yao's protocol) and `LWY` for the hidden-specification step (the DDH-based extension). Within each namespace, the protocol logic is expressed as a sequence of concrete state classes. For example, the hidden-specification System proceeds through states such as `InitSystem`, `RecvLabels`, `GenerateGarbledGates`, `SendGarbledGates`, `SendSystemInputLabels`, `SendFlagBitLabels`, and (in the first round only) `System-ObliviousTransfer`, followed by `RecvFlagBit` and `UpdateSystem`; subsequent rounds cycle back to the garbling phase.

A key feature of this design is that it supports *protocol composition*: the oblivious transfer sub-protocol (Bellare-Micali) is itself implemented as a separate state machine in the `BM` namespace, with its own sender and chooser state sequences. During the OT phase, the outer protocol state (`SystemObliviousTransfer` or `MonitorObliviousTransfer`) delegates to the inner OT state machine, forwarding send/receive operations through the same message handler. This is invoked once per Monitor-state bit in the first round, totalling m OT instances during setup.

An `Interface` class for each party (e.g., `LWY::SystemInterface`) drives the state machine: at each step, it synchronizes communication (sending or receiving as dictated by the current state), then transitions to the next state. The main loop runs until the state becomes null (indicating protocol completion or termination).

Memory management. Each party maintains a `Memory` structure that persists across state transitions. For the System in the hidden-specification step, this includes the driver base labels (as `BigInt` values from GMP), the feed-in wire labels, the garbled gates, the current garbling exponents (α^0, α^1), and the next-round exponents (β^0, β^1). For the Monitor, the memory stores the circuit representation, the shuffled circuit (for topology hiding), the evaluated driver labels from the previous round, and the flag-bit labels.

All big-integer arithmetic is handled through the GMP library's C++ wrapper (`mpz_class`), aliased as `BigInt`. Group elements and exponents are represented uniformly as `BigInt` values. The `QuadraticResidueGroup` class provides the group operations (multiplication, exponentiation, inversion) as modular arithmetic operations on these integers.

Message passing. Communication between the System and Monitor processes uses ZeroMQ, a high-performance asynchronous messaging library. We use the request-reply (REQ-REP) socket pattern: each party creates one REQ socket (for sending) and one REP socket (for receiving), bound to fixed TCP ports on `localhost`. The `MessageHandler` class encapsulates this setup, providing simple `send()` and `recv()` methods that serialize messages as strings.

Messages are serialized as whitespace-delimited hexadecimal strings. Big integers are converted to base-16 string representations via GMP's `get_str()` method. Garbled gates are serialized as four consecutive ciphertext strings. This text-based serialization is straightforward and aids debugging, though a binary format would reduce message sizes by roughly 50%.

5.4.2 Circuit synthesis pipeline

The end-to-end pipeline for converting a monitoring specification into an executable protocol involves four stages: specification authoring, circuit synthesis, BLIF parsing, and circuit construction.

Step 1: Specification authoring. Each specification is written as a synthesizable Verilog module named `Spec`, with three ports: `monitor` (input, the current monitor state), `system` (input, the current system observation), and `out` (output, the concatenation of the next monitor state and the flag bit). The Verilog module uses combinational logic (`always @(*)`) to compute the next state and flag from the current inputs.

For example, the ACS specification declares parameters `NDOORS` and `WORDLEN`, splits the system input into per-door quadruples of entered/exited counts, accumulates them into counters `cntA` and `cntB`, and sets the fault flag if `cntA < cntB`. The Locks specification similarly processes per-lock commands and flags double-locking or double-unlocking.

Step 2: Circuit synthesis with Yosys and ABC. The shell script `YosysParser.sh` invokes the Yosys open synthesis suite with a synthesis script (`synth.ys`). The script performs the following operations:

1. **Read and elaborate.** Yosys reads the Verilog file with parameterized defines (e.g., `-D NDOORS=10 -D WORDLEN=16`) and elaborates the design hierarchy.
2. **Technology mapping.** The `techmap` pass converts high-level Verilog constructs (adders, comparators, multiplexers) into a gate-level netlist.
3. **Gate mapping with ABC.** The `abc -g NAND` pass uses the Berkeley ABC synthesis tool [BM10] to map the netlist to NAND gates only (required for the hidden-specification step, where all gates must be of a single type to avoid leaking topology information through gate types). ABC also performs logic optimization, which typically reduces circuit size by 40–50% compared to a naïve decomposition.
4. **Equivalence checking.** The synthesis script creates a miter circuit between the original and synthesized modules and uses SAT solving to formally verify their equivalence, ensuring correctness of the synthesis.
5. **Output.** Yosys writes the result in two formats: synthesized Verilog (`synth.v`) and BLIF (`synth.blif`). The BLIF file is consumed by the C++ code.

Step 3: BLIF parsing. The `BlifParser` class in the C++ code reads the BLIF file produced by Yosys. It parses the `.inputs`, `.outputs`, and `.names` (gate) directives, building an in-memory circuit graph. Each BLIF gate specifies its input wires and a truth table; since all gates are NAND gates after ABC synthesis, each truth table maps to a single NAND operation.

Step 4: Circuit construction. The parsed BLIF graph is converted into a `Circuit` object, which stores the gates as `Driver` objects (an abstraction for anything that can drive a wire: either an input wire or a gate output). Each `Gate` records its left and right input driver IDs, and each `Driver` records which gates it feeds into. The circuit supports evaluation (`evaluate()`) for testing correctness, and shuffling (`shuffle()`) for the hidden-specification step, where the Monitor randomly permutes the gate ordering to define the topology-hiding permutation π.

5.4.3 Cryptographic implementation

Group choice and parameters. The DDH assumption is required to hold in the group used by the hidden-specification step. We use the group of *quadratic residues* QR_p modulo a safe prime p. A prime p is *safe* if $p = 2q + 1$ where q is also prime (a Sophie Germain prime). The group QR_p has order q, and the DDH assumption is believed to hold in QR_p when p is a safe prime [Bon98].

We use the specific primes defined in RFC 2409 [CH98] and RFC 3526 [KK03]: MODP groups of sizes 768, 1024, 1536, 2048, 3072, and 4096 bits. These are well-known, widely deployed primes whose safety (in the safe-prime sense) has been independently verified. The security parameter n in our experiments corresponds to the bit-length of the prime p; for example, $n = 2048$ uses the 2048-bit MODP prime, providing approximately 112 bits of security against discrete-logarithm attacks.

The base generator of QR_p is 4 (since $4 = 2^2$ is always a quadratic residue). Random generators are produced by exponentiating 4 to a random exponent modulo p. Random exponents are drawn uniformly from $\{1, \ldots, q-1\}$ using GMP's `mpz_urandomm`.

Symmetric encryption (garbling). The garbled-gate encryption scheme $\mathbf{Enc}_{L,R}(S)$ from Eq. ($) is instantiated using the SHAKE-256 extendable-output function (XOF) from the SHA-3 family. Specifically, for keys L and R (represented as hexadecimal strings) and secret S:

$$\mathrm{Enc}_{L,R}(S) = \text{SHAKE-256}(L \parallel R, |S| + 100) \oplus (S \parallel 1^{100})$$

where $|S|$ denotes the length of S in hexadecimal digits and 1^{100} is a 100-hex-digit constant (400 bits) appended for decryption verification. During decryption, the recipient XORs the hash output with each of the four ciphertexts; the correct decryption is identified by checking that the last 100 hex digits equal $\mathtt{f}^{100}$.

We chose SHAKE-256 over a fixed-output hash (such as SHA-512, which is also implemented in the codebase as an alternative garbler) because group elements in our protocol can be arbitrarily large—a 3072-bit prime yields group elements of 768 hex digits, well beyond the 128-hex-digit output of SHA-512. The XOF property of SHAKE-256 allows us to produce hash outputs of any desired length. SHAKE-256 provides a fixed security level of 256 bits, which does not scale with the group's security parameter; however, 256 bits of hash security exceeds the security level of all group sizes we use (the 4096-bit group provides $\sim$140 bits of security), so this does not introduce a practical weakness.

Oblivious transfer. We implement the Bellare-Micali OT protocol [BM89], which proceeds in three messages:

1. The Sender generates a random group element C and sends it to the Chooser.
2. The Chooser generates a random exponent k, computes a public key $pk_\sigma = g^k$ (where g is a generator) and $pk_{1-\sigma} = C/g^k$, and sends pk_0 to the Sender. (The Sender can compute $pk_1 = C/pk_0$.)
3. The Sender encrypts each message m_b under the corresponding public key using ElGamal-style encryption and sends both ciphertexts. The Chooser can only decrypt m_σ because it knows the discrete log of pk_σ.

In the `BM` namespace, this is implemented as a six-state sender machine (`InitSender` → `GenerateConstant` → `SendConstant` → `RecvPublicKey` → `EncryptMessages` → `SendEncryptedMessages`) and a corresponding six-state chooser machine.

Randomness. Cryptographic randomness is sourced from `/dev/urandom`, the Linux kernel's non-blocking pseudo-random number generator. To minimize the overhead of repeated system calls, random bytes are buffered: a 1 MB buffer is allocated at program startup and refilled from `/dev/urandom` whenever it is exhausted. Random hexadecimal strings for wire labels in the open-specification step are generated directly from this buffer. For the hidden-specification step, random group exponents are generated via GMP's `mpz_urandomm`, seeded with a time-based seed (note: this uses Mersenne Twister internally, which is not cryptographically secure; the implementation includes a TODO to replace this with a CSPRNG for production use).

5.4.4 Engineering decisions

Choice of C++. We chose C++ for its combination of high performance and access to mature cryptographic libraries. The performance-critical operations—modular exponentiation and hashing—are delegated to GMP and OpenSSL, both of which are highly optimized C libraries with decades of development. C++ provides zero-cost abstractions (templates, virtual dispatch for the state machine pattern) without sacrificing the ability to call into C libraries directly. Alternatives such as Rust would offer stronger memory safety guarantees but lack the mature ecosystem of cryptographic big-integer libraries; Python, while offering rapid prototyping, would be too slow for the tight inner loops of garbling and exponentiation.

Choice of ZeroMQ. ZeroMQ provides a simple, high-level API for inter-process communication while handling low-level concerns such as message framing, buffering, and reconnection. We use the REQ-REP pattern, which enforces strict alternation between send and receive operations—matching the turn-taking structure of our protocol. Alternatives such as raw TCP sockets would require manual message framing; gRPC would add unnecessary complexity (protocol buffers, code generation) for our simple message format.

Localhost binding. In all experiments, both the System and Monitor processes run on the same machine and communicate over `localhost`. This means the reported round times reflect computation time but not network latency. For a real deployment over a network, the additional latency per round would equal one network round-trip time (typically 1–100 ms for a WAN), which is negligible compared to the computational cost of garbling for circuits with $c > 1000$ gates.

Timeout. We use a timeout of one hour per protocol round (excluding the first round's initialization). This generous timeout accommodates the largest circuit sizes ($c \approx 42000$ gates) at the highest security parameters ($n = 4096$), where a single round can take over 30 minutes. In a production deployment, the timeout would be tuned to the expected circuit size and security parameter.

Experimental hardware. All experiments were run on a personal computer with an Intel Core i5-1235U processor, 16 GB of memory, running Linux Mint 21.3.

CHAPTER 6

Related Work

In this chapter, we survey existing work related to privacy-preserving verification and monitoring, and position our contributions relative to the literature.

6.1 Privacy-preserving static verification

Recent work on privacy-preserving verification has largely focused on static settings. A wide range of verification paradigms have been shown to be amenable to cryptographic techniques: for example, verifying resolution proofs in zero knowledge [LAH+22], solving SAT formulas [LJA+22], matching strings against regular expressions [LWS+24], checking string and regular expression equivalence [KAAP25], and model-checking CTL specifications [JLAP20]. In contrast, our work targets runtime verification, where the system's data evolves dynamically and must be processed incrementally at each step. This fundamental difference means that static privacy-preserving verification techniques cannot be directly applied to the monitoring setting we consider.

6.2 Privacy-preserving runtime monitoring

Banno et al. [BMM+22] and Waga et al. [WMS+24] use fully homomorphic encryption (FHE) for oblivious online monitoring of LTL [Pnu77] and STL [MN04] specifications, respectively. Their work is the closest in spirit to ours, but the two lines of work differ in trust model, specification language, and the underlying cryptographic machinery.

Trust model. In their setting, an evaluating server (or the system itself) performs the monitoring computation on FHE-encrypted data; because only the key holder can decrypt the result, the server learns neither the system's data nor the verdict. The system—as key holder—is the party that ultimately learns whether the specification is satisfied. In our setting the roles are reversed: the *Monitor* must learn the verdict, while the System's data stays hidden. Adapting FHE to our trust model is conceivable—for example, the System could encrypt its data under the Monitor's public key—but doing so naïvely would let the Monitor decrypt the raw data, defeating the purpose. A more careful adaptation using re-encryption or proxy schemes might work, but would add further overhead to a primitive that is already expensive.

Specification representation and scalability. Banno et al. and Waga et al. represent specifications as temporal-logic formulas that are converted into deterministic finite automata (DFAs). Their FHE-based evaluation is linear in the DFA size: each monitoring round homomorphically multiplies an encrypted state vector by a transition matrix, which requires $O(|Q|^2)$ ciphertext multiplications per round (where $|Q|$ is the number of DFA states). For small specifications this is fast—around 2 ms for a 500-state DFA [BMM+22]. However, specifications arising in practice can produce DFAs that are astronomically large: the access-control specification in our experiments has at least 2^{32} states, and the lock-ordering specification has at least 2^{300} states (see Table 5.1). At those sizes, the FHE-based approach becomes infeasible regardless of implementation quality, because the bottleneck—homomorphic ciphertext multiplication, which is orders of magnitude slower than plaintext arithmetic—is applied $O(|Q|^2)$ times per round.

Our protocol avoids the DFA blowup entirely by working directly on Boolean circuits synthesized from the specification. The circuit representation is exponentially more compact for specifications with large state spaces (our lock-ordering circuit has roughly 6000 gates), and garbled-circuit evaluation replaces the expensive homomorphic multiplications with symmetric-key operations (or, in the hidden-specification step, modular exponentiations in a DDH group). The price we pay is a weaker adversary model: our protocol assumes semi-honest parties, whereas Banno et al. and Waga et al. handle malicious adversaries.

Scope of contribution. We do not claim to introduce new cryptographic primitives or techniques. Two-party computation for reactive functionalities is textbook material—Hazay and Lindell [HL10] give a standard treatment—and our protocol combines well-established building blocks (Yao's garbled circuits, the DDH-based topology hiding of Liu et al. [LWY22], and Bellare-Micali oblivious transfer [BM89]) in a straightforward way. The label-reuse mechanism that avoids repeated oblivious transfer is our main design choice, but the underlying idea is not fundamentally different from standard techniques for stateful two-party computation. Our contribution is applying these tools to the monitoring problem and providing concrete implementations; whether a more careful application of FHE or other mainstream MPC techniques could achieve comparable performance remains an open question, and our experiments in Chapter 5 supply baselines for such a comparison.

6.3 Two-party computation and private function evaluation

Our protocols draw inspiration from two-party computation [GMW87, Yao82], and more specifically private function evaluation (PFE), where one party owns a secret circuit and part of the input, while the other party holds the remaining input. The seminal PFE protocol of Katz and Malka [KM11] uses homomorphic encryption to obliviously evaluate circuits whose topology is hidden. Mohassel and Sadeghian [MS13] and Bingöl et al. [BBKL19] achieved efficiency improvements through reduced overhead and better use of oblivious transfer extensions, respectively.

Most recently, Liu, Wang, and Yu [LWY22] provided a way to repeatedly compute the same function several times under standard IND-CPA and DDH assumptions against *covert adversaries*. Our hidden-specification step draws specifically on their work. However,

	Ours	**Banno+**	**Waga+**	**LWY**
Reactive state	✓	✓	✓	—
Hidden spec	✓	—	—	✓
Msgs/round	1	$O(1)$	$O(1)$	$O(1)$
Adversary model	semi-honest	malicious	malicious	covert
Spec language	circuits	LTL/DFA	STL	circuits

Table 6.1: Comparison of our protocol with the most closely related works. "Banno+" refers to Banno et al. [BMM$^+$22] and "Waga+" to Waga et al. [WMS$^+$24].

their protocol does not support *reactive functionalities*: it cannot maintain an internal state that persists across evaluations and is kept secret from all parties. This is essential for runtime monitoring where the specification maintains state across rounds.

More broadly, two-party protocols for reactive functionalities are well-understood; Hazay and Lindell [HL10] give a textbook treatment. Secret-sharing-based constructions (e.g., Shamir or arithmetic sharing) require three or more parties and are therefore inapplicable to our two-party setting, but OT-based two-party alternatives handle reactive functionalities with standard techniques—at the cost of multiple message exchanges per round [CSW20]. Our protocol avoids this multi-round interaction by reusing garbled-circuit labels across rounds, achieving a single message per round from System to Monitor. This is a practical convenience, not a fundamental barrier: the trade-off is restricting to the semi-honest model.

Positioning summary. Table 6.1 summarizes the key differences between our work and the most closely related approaches.

CHAPTER 7

Conclusion and Future Work

7.1 Summary

In this thesis, we presented a protocol for privacy-preserving runtime monitoring, developed in two steps. In the first step (Section 4.4), we constructed a protocol for the open-specification setting, where the specification circuit is known to both the Monitor and the System. In the second step (Section 4.5), we extended this to the hidden-specification setting, where the circuit describing the specification must also remain secret from the System.

Both steps of the protocol are correct and secure under standard cryptographic assumptions (CPA-secure encryption, secure oblivious transfer, and, for the hidden-specification step, the Decisional Diffie-Hellman assumption). The protocol sends only a single message per round from the System to the Monitor, reducing communication overhead compared to approaches based on secret sharing.

Our experimental evaluation (Chapter 5) demonstrates the feasibility of the protocol for circuits with up to 10^5 gates at industrially relevant security parameters. In particular, the hidden-specification step exhibits a time complexity dominated by the size of the specification circuit rather than the size of the System's observable output, enabling scalability to large system sizes as long as the specification remains small.

7.2 Future work

While the levels of privacy provided by our protocol are sufficient for many real-world applications, they fall short of the requirements in highly privacy-sensitive settings. Several directions for future work remain.

Covert and malicious adversaries. We only assume semi-honest parties in the current work. Extending the protocol to protect against actively malicious parties—those who intentionally deviate from the protocol—may be computationally expensive. A potential compromise is to consider *covert systems*, as proposed by Aumann and Lindell [AL10], where adversaries that deviate from the protocol are caught with a positive probability. For monitoring applications, repeated interactions increase the likelihood of catching cheating agents across multiple rounds, ultimately approaching probabilities close to 1.

Circuit optimization. Even for the setting of semi-honest parties, we believe that our protocol could be enhanced by optimizing the number of gates that represent the specification (see Fig. 5.5). Heuristics can be employed to reduce circuit size, but an alternative approach is to relax the specifications—either in terms of soundness or completeness—depending on the specific requirements, to enable encoding with smaller circuits.

Parallelization. Another direction to improve the performance of our protocol is to parallelize the computation. The process of garbling gates is inherently parallelizable for both parties, particularly for the System. Similarly, the Monitor's task of ungarbling can also be parallelized, with the primary bottleneck being the depth of the circuit. Consequently, finding circuits with lower depth, even if it means increasing the number of gates, could enable faster parallelized algorithms.

Integration with monitoring tools. Our protocol works for specifications described by register automata, or using Yosys for circuit synthesis. It would be future work to integrate it with state-of-the-art monitoring tools such as BeepBeep [BKH18], DejaVu [HPU18], or MonPoly [BKZ17].

Distributed monitoring. Lastly, our current protocol assumes that the Monitor and the monitored System are single entities, and that the Monitor relies on a linear order of observations. Developing privacy-preserving monitoring protocols for scenarios where the System and Monitor are distributed is an interesting research challenge.

Bibliography

[AL10] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptol.*, 23(2):281–343, 2010.

[App16] Benny Applebaum. Garbling xor gates “for free” in the standard model. *Journal of Cryptology*, 29(3):552–576, Jul 2016.

[BBKL19] Muhammed Ali Bingöl, Osman Biçer, Mehmet Sabir Kiraz, and Albert Levi. An efficient 2-party private function evaluation protocol based on half gates. *The Computer Journal*, 62(4):598–613, 2019.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. *Introduction to Runtime Verification*, pages 1–33. Springer International Publishing, Cham, 2018.

[Bir11] Alex Biryukov. *Chosen Plaintext Attack*, pages 205–206. Springer US, Boston, MA, 2011.

[BKH18] Mohamed Recem Boussaha, Raphaël Khoury, and Sylvain Hallé. Monitoring of security properties using beepbeep. In Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquin Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 160–169, Cham, 2018. Springer International Publishing.

[BKZ17] David A. Basin, Felix Klaedtke, and Eugen Zalinescu. The monpoly monitoring tool. In *RV-CuBES*, 2017.

[BLS07] Andreas Bauer, Martin Leucker, and Christian Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In Oleg Sokolsky and Serdar Taşıran, editors, *Runtime Verification (RV 2007)*, volume 4839 of *LNCS*, pages 126–138. Springer, 2007.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology*, 20(4):14:1–14:64, 2011.

[BM82] Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudo random bits. In *23rd Annual Symposium on Foundations of Computer Science FOCS*, pages 112–117. IEEE Computer Society, 1982.

[BM89] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO ’89, 9th Annual International Cryptology Conference*, volume 435 of *Lecture Notes in Computer Science*, pages 547–557. Springer, 1989.

[BM10] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *Computer Aided Verification*, pages 24–40, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[BMM$^+$22] Ryotaro Banno, Kotaro Matsuoka, Naoki Matsumoto, Song Bian, Masaki Waga, and Kohei Suenaga. Oblivious online monitoring for safety LTL specification via Fully Homomorphic Encryption. In *Computer Aided Verification - CAV*, pages 447–468. Springer International Publishing, 2022.

[Bon98] Dan Boneh. The decision diffie-hellman problem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 48–63, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[CFMS15] Fraser Cameron, Georgios Fainekos, David M. Maahs, and Sriram Sankaranarayanan. Towards a verified artificial pancreas: Challenges and solutions for runtime verification. In *Runtime Verification*, pages 3–17. Springer International Publishing, 2015.

[CH98] David Carrel and Dan Harkins. The Internet Key Exchange (IKE). RFC 2409, November 1998.

[CR07] Feng Chen and Grigore Roşu. MOP: An efficient and generic runtime verification framework. In *Proceedings of OOPSLA 2007*, pages 569–588. ACM, 2007.

[Cra17] S Michael Crawford. Goodhart's law: when waiting times became a target, they stopped being a good measure. *BMJ*, 359, 2017.

[CSW20] Ran Canetti, Pratik Sarkar, and Xiao Wang. Blazing fast ot for three-round uc ot extension. In *Public-Key Cryptography – PKC*, pages 299–327. Springer International Publishing, 2020.

[GDPT13] Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of ETAPS*, volume 7795 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2013.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, page 218–229, New York, NY, USA, 1987. Association for Computing Machinery.

[HG08] Klaus Havelund and Allen Goldberg. Verify your runs. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments (VSTTE 2005)*, volume 4171 of *LNCS*, pages 374–383. Springer, 2008.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient Secure Two-Party Protocols: Techniques and Constructions.* Springer, 2010.

[HPU18] Klaus Havelund, Doron Peled, and Dogan Ulus. Dejavu: A monitoring tool for first-order temporal logic. In *2018 IEEE Workshop on Monitoring and Testing of Cyber-Physical Systems (MT-CPS)*, pages 12–13, 2018.

[IF13] Mid Staffordshire NHS Foundation Trust Public Inquiry and R. Francis. *Report of the Mid Staffordshire NHS Foundation Trust Public Inquiry: Executive Summary.* HC (Series) (Great Britain. Parliament. House of Commons). Stationery Office, 2013.

[JLAP20] Samuel Judson, Ning Luo, Timos Antonopoulos, and Ruzica Piskac. Privacy preserving CTL model checking through oblivious graph algorithms. In Jay Ligatti, Xinming Ou, Wouter Lueks, and Paul Syverson, editors, *WPES'20: Proceedings of the 19th Workshop on Privacy in the Electronic Society, Virtual Event, USA, November 9, 2020*, pages 101–115. ACM, 2020.

[JLK$^+$16] Yu Jiang, Han Liu, Hui Kong, Rui Wang, Mohammad Hosseini, Jiaguang Sun, and Lui Sha. Use runtime verification to improve the quality of medical care practice. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, page 112–121, New York, NY, USA, 2016. Association for Computing Machinery.

[KAAP25] John Kolesar, Shan Ali, Timos Antonopoulos, and Ruzica Piskac. Coinductive proofs of regular expression equivalence in zero knowledge, 2025.

[Kil88] Joe Kilian. Founding crytpography on oblivious transfer. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, page 20–31, New York, NY, USA, 1988. Association for Computing Machinery.

[KK03] Mika Kojo and Tero Kivinen. More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE). RFC 3526, May 2003.

[KKL$^+$02] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring: Fundamentals of java-mac1. *Electronic Notes in Theoretical Computer Science*, 70(4):80–94, 2002. RV'02, Runtime Verification 2002 (FLoC Satellite Event).

[KM11] Jonathan Katz and Lior Malka. Constant-round private function evaluation with linear complexity. In *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security*, volume 7073 of *Lecture Notes in Computer Science*, pages 556–571. Springer, 2011.

[LAH$^+$22] Ning Luo, Timos Antonopoulos, William R. Harris, Ruzica Piskac, Eran Tromer, and Xiao Wang. Proving UNSAT in zero knowledge. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 2203–2217. ACM, 2022.

[Lin17] Yehuda Lindell. How to simulate it – A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. Springer, 2017.

[LJA$^+$22] Ning Luo, Samuel Judson, Timos Antonopoulos, Ruzica Piskac, and Xiao Wang. ppsat: Towards two-party private SAT solving. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2983–3000. USENIX Association, 2022.

[LS09] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[LWS$^+$24] Ning Luo, Chenkai Weng, Jaspal Singh, Gefei Tan, Mariana Raykova, and Ruzica Piskac. Privacy-preserving regular expression matching using TNFA. In *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security*, volume 14983 of *Lecture Notes in Computer Science*, pages 225–246. Springer, 2024.

[LWY22] Yi Liu, Qi Wang, and Siu-Ming Yiu. Making private function evaluation safer, faster, and simpler. In Goichiro Hanaoka, Junji Shikata, and Yohei Watanabe, editors, *Public-Key Cryptography – PKC 2022*, pages 349–378, Cham, 2022. Springer International Publishing.

[Mea14] Alex Mears. Gaming and targets in the English NHS. *Universal Journal of Management*, 2:293–301, 09 2014.

[MN04] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, pages 152–166. Springer, 2004.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, 2013*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.

[NR04] Moni Naor and Omer Reingold. Number-theoretic constructions of efficient pseudo-random functions. *J. ACM*, 51(2):231–262, mar 2004.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[PZ06] Amir Pnueli and Alon Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *LNCS*, pages 573–586. Springer, 2006.

[Rab81] M. Rabin. How to exchange secrets by oblivious transfer. Technical report, Technical Report Tech. Memo TR-81, Aiken Computation Laboratory, 1981.

[Val76] Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing (STOC '76)*, pages 196–203. ACM, 1976.

[WMS+24] Masaki Waga, Kotaro Matsuoka, Takashi Suwa, Naoki Matsumoto, Ryotaro Banno, Song Bian, and Kohei Suenaga. Oblivious monitoring for discrete-time STL via fully homomorphic encryption. In *Runtime Verification - 24th International Conference, RV*, volume 15191 of *Lecture Notes in Computer Science*, pages 59–69. Springer, 2024.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (SFCS 1986)*, pages 162–167, 1986.

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Advances in Cryptology - EUROCRYPT 2015*, pages 220–250, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

AI Assistance Declaration

Portions of this thesis were developed with the assistance of AI-based tools, including GitHub Copilot (using generative models such as Anthropic Claude Opus 4.6 and Sonnet 4.6). These tools were used for drafting support, language refinement, and code-related suggestions. All content has been reviewed, edited, and validated by the author, who takes full responsibility for the final text and results presented.

APPENDIX A

Security Proofs

In this appendix, we provide detailed proofs for the main theorems stated in Chapter 4. Each proof proceeds by defining explicit hybrid experiments, constructing simulators, and reducing indistinguishability to standard cryptographic assumptions.

Throughout this appendix, we use ℓ to denote the number of monitoring rounds, which is assumed to be a fixed polynomial in the security parameter n.

A.1 Correctness and security of the open-specification step

Theorem 4.1 *Under CPA-secure encryption and semi-honest oblivious transfer, the open-specification step is a correct and secure monitoring protocol (without specification hiding) for any polynomial number of rounds.*

Proof. We prove correctness, Monitor security, and System security separately.

Correctness. The correctness argument follows the structure of standard Yao garbled circuit evaluation, extended to the reactive setting.

At each round r, the System generates fresh random labels $w_i^0[r], w_i^1[r]$ for each feed-out wire ω_i (except the Monitor's state input wires for $r > 1$, which reuse the output labels from round $r - 1$). The System garbles each gate G_j by encrypting the output label under the two input labels using the CPA-secure encryption scheme Enc. The Monitor receives:

- the garbled gates $\mathbf{encGG}_j[r]$ for all $j \in \{1, \dots, c\}$;
- the System's input labels $w_{m+i}^{\sigma_i[r]}[r]$ for $i \in \{1, \dots, s\}$;
- the Monitor's state labels (via OT in round 1, or carried over from round $r - 1$);
- both labels $w_{O+m+1}^0[r]$ and $w_{O+m+1}^1[r]$ for the flag output wire.

For each gate G_j in the circuit (processed bottom-up), the Monitor holds exactly one label for each input wire: the label corresponding to the actual bit value on that wire. Using these as decryption keys, exactly one of the four ciphertexts in $\mathbf{encGG}_j[r]$ decrypts successfully (with probability $1 - 2^{-100}$ per gate, due to the 100-bit verification padding). The successfully decrypted value is the output label corresponding to the correct bit value $G_j(\alpha, \beta)$ where α, β are the actual bit values on the left and right input wires.

Propagating from input to output, the Monitor obtains the label $w_{O+m+1}^{\tau[r]}[r]$ for the flag bit, which it matches against the two flag labels to determine $\tau[r]$. By construction, $\tau[r] = \mathsf{flag}(\sigma[r], \mu[r])$, which is the correct output.

For the reactive component: the Monitor also obtains labels $w_{O+i}^{\mu_i[r+1]}[r]$ for $i \in \{1, \ldots, m\}$, corresponding to the next monitor state. By the label-reuse step of the protocol (see Section 4.4), $w_i^b[r+1] = w_{O+i}^b[r]$, so these labels are exactly the input labels the Monitor needs for round $r+1$. The union bound over all gates and all ℓ rounds gives a total error probability of at most $c \cdot \ell \cdot 2^{-100}$, which is negligible for any polynomial ℓ.

Security for the Monitor (System's view). We construct a simulator $\mathcal{S}_{\mathsf{Sys}}$ for the System's view. In the real protocol, the System's view consists of:

- its own input sequence $\sigma[1], \ldots, \sigma[\ell]$;
- the circuit $\mathcal{C}$ (known to both parties);
- the messages received during the OT phase (in round 1);
- the flag bits $\tau[1], \ldots, \tau[\ell]$ (the "proceed" / "terminate" responses from the Monitor).

The simulator $\mathcal{S}_{\mathsf{Sys}}$ operates as follows. It knows $\mathcal{C}$, $\sigma[1], \ldots, \sigma[\ell]$, and $\tau[1], \ldots, \tau[\ell]$ (the System's inputs and the outputs it is entitled to receive). It generates random internal coins and simulates the OT messages using the OT simulator guaranteed by the OT security assumption. Since the System acts as the OT sender, the OT simulator produces messages indistinguishable from real ones without knowledge of the Monitor's choice bits $\mu_i[1]$.

This is straightforward because the System generates all garbled gates and labels itself—it does not receive any messages that depend on the Monitor's private state $\mu[r]$. The only message the System receives from the Monitor is the flag bit, which the simulator knows. Therefore, $\mathcal{S}_{\mathsf{Sys}}$ simply outputs the System's own random coins, the simulated OT transcript, and the flag bits. The indistinguishability follows from the OT security assumption.

Security for the System (Monitor's view). This is the main part of the proof. We construct a simulator $\mathcal{S}_{\mathsf{Mon}}$ and show indistinguishability through a hybrid argument.

The Monitor's real view. In round r, the Monitor's view consists of:

- its own inputs $(\mathcal{C}, \mu[1])$ and random coins;
- in round 1: the OT transcript, through which it receives $w_i^{\mu_i[1]}[1]$ for each $i \in \{1, \ldots, m\}$;

- for each round r: the garbled gates $\mathbf{encGG}_j[r]$ for $j \in \{1, \ldots, c\}$, the System input labels $w_{m+i}^{\sigma_i[r]}[r]$ for $i \in \{1, \ldots, s\}$, and the flag labels $w_{O+m+1}^0[r]$, $w_{O+m+1}^1[r]$.

Hybrid experiments. We define a sequence of hybrid experiments $\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_c$ for each round r. In hybrid $\mathcal{H}_0$, the transcript is generated exactly as in the real protocol. In hybrid $\mathcal{H}_c$, the transcript is generated by the simulator. The intermediate hybrids interpolate by replacing garbled gates with simulated ones, proceeding from the output gates to the input gates.

For a single round r, let $G_{j_1}, G_{j_2}, \ldots, G_{j_c}$ be a topological ordering of the gates from outputs to inputs (i.e., G_{j_1} is an output gate).

- **Hybrid $\mathcal{H}_0$ (real):** All garbled gates are computed honestly. The System input labels and Monitor state labels correspond to the actual input values.
- **Hybrid $\mathcal{H}_k$ (for $1 \leqslant k \leqslant c$):** Gates $G_{j_1}, \ldots, G_{j_k}$ are *simulated*: for each such gate, the four ciphertexts are encryptions of the label $w_{\text{out}}^{b^*}[r]$ where b^* is the correct output bit, using all four combinations of input labels. Equivalently, three of the four ciphertexts are replaced with encryptions of the same (correct) output label instead of the labels corresponding to incorrect outputs. Gates $G_{j_{k+1}}, \ldots, G_{j_c}$ are still garbled honestly.
- **Hybrid $\mathcal{H}_c$ (simulated):** All gates are simulated. The Monitor's view consists of garbled gates that encrypt only the correct output label (four times each), plus the labels corresponding to the actual wire values. This is exactly what the simulator $\mathcal{S}_{\mathsf{Mon}}$ produces, since it can compute the correct wire values given $(\mathcal{C}, \mu[1], \tau[1], \ldots, \tau[\ell])$.

Indistinguishability of adjacent hybrids. We show that $\mathcal{H}_{k-1}$ and $\mathcal{H}_k$ are computationally indistinguishable for each k. Consider gate G_{j_k} with input wires carrying values $a, b \in \{0, 1\}$ in the honest evaluation. In $\mathcal{H}_{k-1}$, the four ciphertexts of G_{j_k} are:

$$\text{Enc}_{w_L^\alpha, w_R^\beta}\left(w_{\text{out}}^{G_{j_k}(\alpha,\beta)}\right) \quad \text{for } (\alpha, \beta) \in \{0, 1\}^2.$$

In $\mathcal{H}_k$, the ciphertext for $(\alpha, \beta) \neq (a, b)$ is replaced with:

$$\text{Enc}_{w_L^\alpha, w_R^\beta}\left(w_{\text{out}}^{b^*}\right) \quad \text{where } b^* = G_{j_k}(a, b).$$

The Monitor never obtains the keys w_L^α for $\alpha \neq a$ or w_R^β for $\beta \neq b$ (since gates $G_{j_1}, \ldots, G_{j_{k-1}}$ are already simulated and only provide the correct label). Therefore, distinguishing $\mathcal{H}_{k-1}$ from $\mathcal{H}_k$ requires breaking the CPA security of Enc: a distinguisher could be used to construct an adversary against the CPA game by embedding the challenge ciphertext in one of the three modified positions.

Handling the reactive component. The key subtlety in the reactive setting is that the labels for the Monitor's state output wires in round r become the input labels for round $r + 1$. We must show that this reuse does not break the hybrid argument.

Consider two consecutive rounds r and $r + 1$. The Monitor's state labels $w_{O+i}^b[r]$ (for $b \in \{0, 1\}$, $i \in \{1, \ldots, m\}$) are reused as $w_i^b[r + 1]$. In each round, the hybrid argument replaces garbled gates one at a time. The critical observation is:

1. The Monitor knows only the label $w_{O+i}^{\mu_i[r+1]}[r]$ for the actual state value $\mu_i[r+1]$, not both labels. So the Monitor does not gain additional information about the "other" label $w_{O+i}^{1-\mu_i[r+1]}[r]$ from evaluating the circuit in round r.

2. In round $r+1$, the Monitor uses $w_i^{\mu_i[r+1]}[r+1] = w_{O+i}^{\mu_i[r+1]}[r]$ as an input label. The garbled gates of round $r+1$ use fresh random labels for all other wires, so the hybrid argument for round $r+1$ is independent of the specific values of $w_i^b[r+1]$.

3. The CPA security of Enc holds even when the keys (labels) are reused across different ciphertexts, as long as the adversary does not know both keys for any gate. Since the Monitor only holds one label per wire, this condition is maintained.

Therefore, the hybrid argument can be applied independently to each round, yielding a total distinguishing advantage of at most $c \cdot \ell \cdot \epsilon_{\text{CPA}}$, where ϵ_{CPA} is the CPA advantage, which is negligible for polynomial ℓ.

Simulator construction. The simulator $\mathcal{S}_{\mathsf{Mon}}$ for the Monitor operates as follows:

1. It receives $(\mathcal{C}, \mu[1])$ and $\tau[1], \ldots, \tau[\ell]$.

2. For each round r, it chooses an arbitrary monitor-state string $\tilde{\mu}[r+1] \in \{0,1\}^m$ (e.g., 0^m). Note that the simulator does *not* know the System's inputs $\sigma[r]$ and therefore cannot compute the true next state $\mu[r+1] = f_s(\mu[r], \sigma[r])$. This is not needed: in the fully simulated transcript (hybrid $\mathcal{H}_c$), every garbled gate encrypts a single output label under all four input-label pairs, so the gate tables do not implement any real Boolean function. The specific bit values assigned to internal and state wires are therefore irrelevant to the Monitor's view—only the flag output $\tau[r]$ must be correct.

3. For each round r:

   - It generates one random label $\tilde{w}_i[r]$ per wire, corresponding to the nominal bit value assigned to that wire: the true $\mu_i[1]$ for initial-state wires, $\tilde{\mu}_i[r+1]$ for state-output wires, and arbitrary values for all other internal wires.
   - For each gate G_j, it produces simulated garbled gates: four ciphertexts, all encrypting the correct output label $\tilde{w}_{\text{out}}[r]$ under all four input-label combinations (using random labels for the "wrong" input values).
   - It sets the flag labels so that the label corresponding to $\tau[r]$ is the one the Monitor obtains.

4. It simulates the OT transcript for round 1 using the OT simulator.

The simulator's output is indistinguishable from the real view by the hybrid argument above, completing the proof. □

A.2 Correctness and security of the hidden-specification step

Theorem 4.2 *Under CPA-secure encryption, the DDH assumption on $\mathbb{G}$, and semi-honest oblivious transfer, the hidden-specification step is a correct and secure monitoring protocol with specification hiding for any polynomial number of rounds.*

Proof. We prove correctness, Monitor security, and System security separately. The System security proof is the most involved, as it requires showing that the DDH assumption hides the circuit topology.

Correctness. The correctness of the hidden-specification step follows from the algebraic structure of the labeling scheme.

During setup, the Monitor assigns base labels $g_j \in \mathbb{G}$ to each feed-out wire ω_j (for $j \in \{1, \ldots, O\}$) and exponents $t_i \in \mathbb{Z}_q$ to each feed-in wire ι_i (for $i \in \{1, \ldots, I\}$). The Monitor computes feed-in base labels $\ell_i = g_{\pi(i)}^{t_i}$, where $\pi(i) = j$ if feed-out wire ω_j connects to feed-in wire ι_i.

At round r, the System chooses exponents $\alpha^0[r], \alpha^1[r] \in \mathbb{Z}_q$ and assigns:

$$\begin{aligned} w_j^b[r] &= g_j^{\alpha^b[r]} && \text{for feed-out wires } j \in \{1, \ldots, O\}, \\ u_i^b[r] &= \ell_i^{\alpha^b[r]} = g_{\pi(i)}^{t_i \cdot \alpha^b[r]} && \text{for feed-in wires } i \in \{1, \ldots, I\}. \end{aligned}$$

For a gate G_k with feed-in wires ι_{2k-1} and ι_{2k} connected to feed-out wires of gates G_ℓ and G_r respectively, the System garbles using input labels $u_{2k-1}^b[r] = g_{m+s+\ell}^{t_{2k-1} \cdot \alpha^b[r]}$ and $u_{2k}^b[r] = g_{m+s+r}^{t_{2k} \cdot \alpha^b[r]}$, and output labels $w_{m+s+k}^b[r] = g_{m+s+k}^{\alpha^b[r]}$.

After ungarbling gate G_k, the Monitor obtains $w_{m+s+k}^{b^*}[r] = g_{m+s+k}^{\alpha^{b^*}[r]}$ for the correct output bit b^*. To compute the input label for a downstream gate $G_{k'}$ whose feed-in wire $\iota_{2k'-1}$ is connected to ω_{m+s+k}, the Monitor computes:

$$\left(g_{m+s+k}^{\alpha^{b^*}[r]} \right)^{t_{2k'-1}} = g_{m+s+k}^{t_{2k'-1} \cdot \alpha^{b^*}[r]} = u_{2k'-1}^{b^*}[r],$$

which is exactly the correct feed-in label. This confirms that the Monitor can propagate labels through the circuit correctly.

For the output wires, the System also chooses exponents $\beta^0[r], \beta^1[r]$ and sets $w_{O+i}^b[r] = g_i^{\beta^b[r]}$ for $i \in \{1, \ldots, m\}$ (the Monitor state outputs). In round $r+1$, the System sets $\alpha^b[r+1] = \beta^b[r]$, so $w_j^b[r+1] = g_j^{\beta^b[r]} = w_{O+j}^b[r]$ for $j \in \{1, \ldots, m\}$, maintaining label continuity across rounds.

Security for the Monitor (System's view). We must show that the System learns neither the Monitor's specification state $\mu[r]$ nor the circuit topology (encoded by π).

The System's view across all rounds consists of:

- the base labels $g_1, \ldots, g_O \in \mathbb{G}$ (received during setup);

- the feed-in base labels $\ell_1, \ldots, \ell_I \in \mathbb{G}$ (received during setup);
- the OT transcript (round 1 only);
- the flag bits $\tau[1], \ldots, \tau[\ell]$.

We proceed through a sequence of hybrid experiments.

Hybrid $\mathcal{H}_0$ (real): The experiment is exactly the real protocol execution.

Hybrid $\mathcal{H}_1$ (random feed-in labels): Replace the feed-in base labels $\ell_i = g_{\pi(i)}^{t_i}$ with uniformly random group elements $\ell_i' \xleftarrow{\$} \mathbb{G}$, for all $i \in \{1, \ldots, I\}$.

Claim: $\mathcal{H}_0 \overset{c}{\equiv} \mathcal{H}_1$ under the DDH assumption.

Proof of claim: The System sees O base labels $(g_1, \ldots, g_O)$ and I feed-in labels $(\ell_1, \ldots, \ell_I)$ where $\ell_i = g_{\pi(i)}^{t_i}$. Since each t_i is independently random and π maps each ℓ_i to some g_j, this is a collection of I independent instances of exponentiated group elements.

By Lemma 2.1 (the Naor-Reingold multi-instance DDH lemma), given O random group elements $g_1, \ldots, g_O$, the tuple $(g_{\pi(1)}^{t_1}, g_{\pi(2)}^{t_2}, \ldots, g_{\pi(I)}^{t_I})$ is computationally indistinguishable from a tuple of I uniformly random group elements, regardless of the function π. This is because even if we condition on π, each t_i is independently random, and the DDH assumption guarantees that exponentiating a random group element by a random exponent produces a distribution indistinguishable from uniform.

More precisely, we can reduce to DDH by considering one feed-in wire at a time. For feed-in wire ι_i connected to feed-out wire ω_j (so $\pi(i) = j$), the System sees g_j and $g_j^{t_i}$. Given a DDH challenge (g, g^a, g^b, Z) where Z is either g^{ab} or g^c for random c, we can embed $g_j = g^a$ and $\ell_i = Z^{b'}$ for a fresh random b', so that $Z = g^{ab}$ yields $\ell_i = g_j^{ab'}$ (a valid exponentiated label) while $Z = g^c$ yields $\ell_i = g^{cb'}$ (a random group element). Since there are I feed-in wires, a standard hybrid argument over the I instances gives a reduction with advantage loss I, which is polynomial.

After this replacement, the feed-in base labels are independent of π, and hence the circuit topology is hidden from the System.

Hybrid $\mathcal{H}_2$ (simulated OT): Replace the OT transcript with a simulated transcript produced by the OT simulator.

Claim: $\mathcal{H}_1 \overset{c}{\equiv} \mathcal{H}_2$ by the security of the OT protocol against semi-honest adversaries.

Proof of claim: In the OT phase, the System acts as the sender with messages $(w_i^0[1], w_i^1[1])$ for each Monitor state bit i, and the Monitor acts as the chooser with choice bit $\mu_i[1]$. The OT security guarantee provides a simulator that, given the sender's messages but not the choice bit, produces a transcript indistinguishable from the real one. Since $\mathcal{H}_1$ already replaces the feed-in labels with random values, the OT messages $(w_i^0[1], w_i^1[1])$ are functions only of the (still honest) base labels and exponents, and the OT simulator's output remains indistinguishable.

Hybrid $\mathcal{H}_2$ constitutes the simulator's output. The simulator $\mathcal{S}_{\mathsf{Sys}}$ operates as follows:

1. Generate O random group elements as base labels.
2. Generate I random group elements as feed-in base labels (independent of any permutation π).
3. Simulate the OT transcript using the OT simulator.
4. Output the flag bits $\tau[1], \ldots, \tau[\ell]$.

This simulator uses only the System's inputs ($\sigma[1], \ldots, \sigma[\ell]$), the circuit size c (from which O and I are derived), and the flag bits—it does not require knowledge of $\mathcal{C}$ or $\mu[1]$. The indistinguishability of its output from the real System view follows from $\mathcal{H}_0 \overset{c}{\equiv} \mathcal{H}_1 \overset{c}{\equiv} \mathcal{H}_2$.

Security for the System (Monitor's view). The Monitor's real view across all rounds consists of:

- its own inputs $(\mathcal{C}, \mu[1])$ and random coins;
- the OT transcript (round 1), through which it receives $w_i^{\mu_i[1]}[1]$ for each i;
- for each round r: the garbled gates $\mathbf{encGG}_j[r]$ for all gates j, the System input labels, and the flag labels.

The proof proceeds through a hybrid argument analogous to the open-specification case, but working with group-element labels instead of random strings.

Hybrid experiments for each round r. As in Appendix A.1, we replace garbled gates from output to input, one gate at a time.

- **Hybrid $\mathcal{H}'_0$:** The garbled gates are computed honestly with labels $w_j^b[r] = g_j^{\alpha^b[r]}$.
- **Hybrid $\mathcal{H}'_k$:** Gates $G_{j_1}, \ldots, G_{j_k}$ (ordered output-to-input) are simulated: all four ciphertexts encrypt the correct output label.
- **Hybrid $\mathcal{H}'_c$:** All gates are simulated.

The indistinguishability of adjacent hybrids $\mathcal{H}'_{k-1}$ and $\mathcal{H}'_k$ follows from the CPA security of Enc, by the same argument as in Appendix A.1. The Monitor holds only one label per wire (the one corresponding to the correct bit value), so it cannot distinguish a ciphertext encrypting the wrong output label from one encrypting the correct label, because it lacks the keys to decrypt the modified ciphertexts.

Handling reactivity with exponent reuse. The hidden-specification step introduces an additional subtlety: the exponents $\alpha^0[r+1] = \beta^0[r]$ and $\alpha^1[r+1] = \beta^1[r]$ are reused across rounds. We must verify this does not compromise security.

In each round r, the System chooses fresh $\beta^0[r], \beta^1[r] \overset{\$}{\leftarrow} \mathbb{Z}_q$ for the output wires. These become $\alpha^0[r+1], \alpha^1[r+1]$ in round $r+1$. The Monitor's view of the output labels in round r includes only $w_{O+i}^{\mu_i[r+1]}[r] = g_i^{\beta^{\mu_i[r+1]}[r]}$ for each i—it does *not* see $\beta^0[r]$ or $\beta^1[r]$ directly, only the group elements g_i raised to one of these exponents.

In round $r+1$, the Monitor sees new garbled gates computed with $\alpha^b[r+1] = \beta^b[r]$. The hybrid argument for round $r+1$ only requires that the Monitor holds one label per wire, which is maintained by the label reuse mechanism. The exponent values $\alpha^0[r+1]$ and $\alpha^1[r+1]$ are not revealed by the group elements the Monitor observes (extracting them would require computing discrete logarithms). Therefore, the hybrid argument applies independently to each round.

Simulator construction. The simulator $\mathcal{S}_{\mathsf{Mon}}$ for the Monitor operates as follows:

1. It receives $(\mathcal{C}, \mu[1])$ and $\tau[1], \ldots, \tau[\ell]$.
2. For each round r, it chooses an arbitrary monitor-state string $\tilde{\mu}[r+1] \in \{0,1\}^m$ (e.g., 0^m). As in Appendix A.1, the simulator does *not* know $\sigma[r]$ and cannot compute the true next state; nor does it need to, because in the fully simulated transcript every garbled gate encrypts a single output label under all four input-label pairs.
3. For each round r, it generates one random group element per wire (labelled according to the nominal bit value: $\mu_i[1]$ for initial-state wires, $\tilde{\mu}_i[r+1]$ for state-output wires, and arbitrary values elsewhere) and produces simulated garbled gates (all four ciphertexts encrypting the correct output label).
4. It simulates the OT transcript for round 1 using the OT simulator, providing the Monitor's state label corresponding to $\mu_i[1]$.
5. It sets the flag labels so that the Monitor determines $\tau[r]$ correctly.

Note that the simulator does *not* need the System's inputs $\sigma[r]$—it only needs $\tau[r]$—because the simulated garbled gates encrypt only one label per gate, and the specific bit values on internal and state wires are immaterial to the Monitor's view.

The total distinguishing advantage is at most $c \cdot \ell \cdot \epsilon_{\mathrm{CPA}} + I \cdot \epsilon_{\mathrm{DDH}} + \epsilon_{\mathrm{OT}}$, where ϵ_{CPA}, ϵ_{DDH}, and ϵ_{OT} are the advantages of the respective assumptions, all of which are negligible. This completes the proof. $\square$