UNIVERSITY OF CALIFORNIA,
IRVINE

Towards End-to-End Data Privacy: from Generation to Consumption

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Seoyeon Hwang

2024

Contents

List of Figures

List of Tables

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

My Ph.D. journey was like navigating a lengthy tunnel. Initially excited, I envisioned myself as a researcher, unaware of the hurdles ahead. Having thrilled years of learning how to do research, I often felt lost in the pitch-black tunnel and tried to keep moving forward, groping for the wall. Upon reaching my Ph.D. candidacy, I realized that the tunnel was U-shaped. I could see the light from the exit, but it was still too far. After gradually stacking my progress every year, now I'm almost reaching the end. Although uncertain about what lies beyond, I'm confident I can confront challenges and persevere until the end. Looking back, one thing remains clear: I could not have finished this journey without the invaluable support of the amazing people listed below.

First, I sincerely appreciate my Ph.D. advisor, Gene Tsudik, for allowing me to start this journey and waiting for me to grow with empowering support. He never hesitated to share his insights and encouraged me with caring advice as my academic and personal mentor. I'd also like to thank Stanislaw Jarecki for sparing his time for fun crypto-related conversation and Athina Markopoulou for generously supporting our WiCyS student chapter at UCI. I am honored to have them on my defense committee. I'm also thankful to the ICS Steckler Family Endowed Fellowship for their generous support during my last year of Ph.D.

I cannot miss my early mentors in Korea, Hyang-Sook Lee, Sang-Ho Lee, and Jung Hee Cheon. I would not have dreamed or started this journey without their support.

I am grateful to all my co-authors and collaborators who shared so many days and nights and gave me constructive feedback: Norrathep Rattanavipanon, Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Ercan Ozturk, Xavier Carpent, Yoshimichi Nakatsuka, Devriş İşler, Elina Van Kempen, Zane Karl, Stanislaw Jarecki, Karim Eldefrawy, Rafail Ostrovsky, Moti Yung, and, of course, Gene Tsudik.

I was fortunate to have amazing friends in the SPROUT lab at UCI: Norrathep Rattanavipanon, Ivan De Oliveira Nunes, Ercan Ozturk, Yoshimichi Nakatsuka, Sashidhar Jakkamsetti, Andrew Searles, Youngil Kim, Renascence Tarafder Prapty, Elina Van Kempen, and Benjamin Terner. Special thanks to Gene for running a diverse collaborative lab and planning fun events. Thank you all, and I will never forget all the fun moments!

I also thank all my friends I met in the U.S. and old friends in Korea for their encouragement and understanding, allowing me to take a break from my career and rejuvenate.

Finally, I extend my heartfelt gratitude to my family for their unconditional love and encouragement throughout my journey. My parents, Young Joong and Hee Jeong, have always believed in me and waited for me without imposing undue pressure. My big sis, Joo Yeon, and little brother, Joon Hyeong, have always been proud of me. All of them had been my bedrock while working abroad. My husband, Dayeol, has been there for me and has given me strong support at every step of this journey. As I always tell you, I would not have come this far without you. Thank you, and I love you.

VITA

Seoyeon Hwang

EDUCATION

Doctor of Philosophy in Computer Science	**2024**
University of California, Irvine	*Irvine, California*
Master of Science in Computational Sciences	**2021**
University of California, Irvine	*Irvine, California*
Master of Science in Mathematics	**2016**
Ewha Womans University	*Seoul, South Korea*
Bachelor of Science in Mathematics & Information Security	**2014**
Ewha Womans University	*Seoul, South Korea*

PROFESSIONAL EXPERIENCE

Research Scientist	**2023–Present**
Stealth Software Technologies, Inc.	*Los Angeles, California*
Graduate Research Assistant	**2020–2023**
University of California, Irvine	*Irvine, California*
Applied Scientist Intern in Cryptography Team	**2021**
Amazon.com	*(Remote) Seattle, Washington*
Applied Scientist Intern in Cryptography Team	**2020**
Amazon.com	*(Remote) Seattle, Washington*
Security Intern in Computer Science Laboratory	**2019**
Stanford Research Institute International	*Menlo Park, California*
Junior Engineering Staff	**2016–2017**
Telecommunications Technology Association	*Gyeonggi-do, South Korea*
NSR Cryptographic Skill Training Course	**2015–2016**
National Security Research Institute (NSR)	*Daejeon, South Korea*
Intern	**2013**
Penta Security Systems Inc.	*Seoul, South Korea*
Undergraduate Internship in Cryptography Lab	**2013**
Ewha Womans University	*Seoul, South Korea*
Undergraduate Internships	**2011,2012**
Institute of Mathematical Science in Ewha Womans University	*Seoul, South Korea*

TEACHING EXPERIENCE

Teaching Assistant for Computer & Network Security (ICS 134) **2020**
University of California, Irvine *Irvine, California*

Reader for Computer & Network Security (ICS 134) **2019**
University of California, Irvine *Irvine, California*

Reader for Computer Security (CS 201P) **2019**
University of California, Irvine *Irvine, California*

Reader for Boolean Algebra and Logic (ICS 6B) **2018**
University of California, Irvine *Irvine, California*

REFEREED CONFERENCE PUBLICATIONS

Element Distinctness and Bounded Input Size in Private Set Intersection and Related Protocols **2024**
22nd Conference on Applied Cryptography and Network Security (ACNS)

PARseL: Towards a Verified Root-of-Trust over seL4 **2023**
IEEE/ACM International Conference on Computer-Aided Design (ICCAD)

Privacy-from-Birth: Protecting Sensed Data from Malicious Sensors with VERSA **2022**
IEEE Security and Privacy (S&P)

Communication-Efficient (Proactive) Secure Computation for Dynamic General Adversary Structures and Dynamic Groups **2020**
Security and Cryptography for Networks (SCN)

REFEREED JOURNAL PUBLICATIONS

Balancing Security and Privacy in Genomic Range Queries **2022**
ACM Transactions on Privacy and Security (TOPS)

PAPERS IN SUBMISSION OR UNDER REVIEW

Publicly Verifiable Watermarking Protocol **2024**

Privacy-Preserving Identity Verification Methods for Accountless Users via Private List Intersection and Its Variants	**2024**
Proof of Participation in Federated Learning	**2024**

ABSTRACT OF THE DISSERTATION

Towards End-to-End Data Privacy: from Generation to Consumption

By

Seoyeon Hwang

Doctor of Philosophy in Computer Science

University of California, Irvine, 2024

Professor Gene Tsudik, Chair

Preserving data privacy is a formidable challenge in today's interconnected and data-centric world. Individuals are surrounded by "smart" devices that collect and generate massive amounts of sensitive data. Moreover, organizations collect personalized data, including private information, to provide more functionalities and better quality for their data-driven services. Therefore, ensuring data privacy throughout its lifecycle, i.e., from generation to consumption, is paramount.

To this end, this dissertation tackles several challenges to attain such *end-to-end data privacy.* We first investigate lower-end devices to preserve data privacy *from its generation*, and propose two secure architectures: one for mid-range devices with memory management unit and the other for low-end devices with no security features. Then, we revisit cryptographic computing, a promising privacy-enhancing technology for *data in use*, focusing on input correctness, generalized adversary models, and challenges in real-world applications.

Chapter 1

Introduction

Private data generation has significantly increased due to the growing adoption of data-driven applications. Individuals continuously produce substantial volumes of data across a spectrum of applications, from the Internet of Things (IoT) and wearable devices to social media platforms and e-commerce websites. Moreover, technological advancements such as artificial intelligence, machine learning, and big data analytics propel data collection for various purposes, including the targeted advertisement, personalized healthcare, and predictive analytics. These applications enhance their functionalities and efficacy by harnessing personalized data, predominantly composed of private information.

Such private data is being generated in an increasing number of interconnected devices, emphasizing the critical need to maintain data privacy from its generation. As more devices become interconnected, data breaches on one device can rapidly spread to others, resulting in serious privacy invasions. For example, a smart home with a smart camera, doorbell, and thermostat installed was hacked [270] through their Wi-Fi and smart camera, and the hacker increased the temperature to 90 degrees and played vulgar music through the camera. Such attacks are prevalent on IoT devices, including smart TVs, speakers, and even light bulbs [296, 10, 12, 298]. These incidents highlight the importance of protecting data on such end devices to guarantee data privacy from its generation.

Furthermore, given the impact of private data volume on service functionalities and quality, many organizations are leveraging more data from their services and seeking collaborations. For instance, a 2018 report [268] shows Google's extensive data collection on user information, aiming at providing advertisers with more tailored insights [212]. This trend has led to substantial privacy concerns arising from the over-collected user data across many organizations [132, 201, 177], resulting in bad brand reputations as well as hefty fines [265, 264, 278] mandated by privacy regulations such as the General Data Protection Regulation [244] and the California Consumer Privacy Act [206]. Consequently, it is an inevitable challenge to ensure data privacy while utilizing private data in computation.

Given these motivations, this thesis tackles several challenges to achieve *data privacy from its generation to consumption*; i.e., *end-to-end data privacy*. The first half focuses on preserving data privacy *from generation*, especially on the lower-end smart devices with simple microcontroller units (MCU), and the second half enhances the privacy of data *in use* via improving existing cryptographic computing techniques for real-world applications.

This thesis first delves into mid-range MCUs between the low-end and high-end systems, which are used in various IoT/embedded devices with multiple functionalities. Due to their compact sizes and low prices, these MCUs usually contain minimal security features, such as memory management units (MMU) or memory protection units (MPU), which provide virtual memory support and access control over specific memory regions. Although these mid-range MCUs are more capable than low-end MCUs with a single core, they still lack computing power and resources compared to high-end systems with microprocessors or system-on-chips.

For such resource-constrained devices, many researchers have proposed various architectures for remote attestation ($\mathcal{R}\mathsf{A}$), e.g., [252, 195, 238, 108, 158], to verify the software state of a remote device, often incorporating robust computer-aided formal verification [108, 158]. However, these mostly focus on low-end MCUs and require some hardware modifications. On the other end of the spectrum, isolated execution systems, e.g., [180, 93], implement $\mathcal{R}\mathsf{A}$ for user-level processes on high-end systems, such as desktops, laptops, and cloud servers, with substantial dedicated hardware support. However, this support is unsuitable for mid-range MCUs and complicates the formal verification of their design and implementation. To fill this gap, we propose a secure $\mathcal{R}\mathsf{A}$ architecture for the mid-range MCUs with MMU over a formally verified microkernel seL4 and formally verify its security guarantees and implementation.

Then, this thesis addresses low-end MCUs with no security features. These devices typically run a single software directly on bare metal, i.e., without any microkernel or operating systems. Since no isolations or fine-grained access control can be provided, ensuring data privacy on such devices is crucial to prevent compromised software from leaking (private) data. To provide these guarantees, we claim that the generated data must be protected from its “birth", i.e., from the moment it is digitalized until it leaves the device. We formalize this concept as *Privacy-from-Birth (PfB)* and propose a secure architecture for the low-end MCUs that realizes *PfB* with formal verification. Note that while $\mathcal{R}\mathsf{A}$ techniques provide a passive root of trust by detecting violations or compromises after they occur, our approach introduces an active root of trust, preventing unauthorized access to the sensed data on the low-end devices.

For preserving the privacy of data *in use*, the second half of this thesis focuses on improving *cryptographic computing* techniques. Cryptographic computing is one of the leading privacy-enhancing technologies for data in use[1]. This approach converts input data into a cryptographically protected form, ensuring that no information can be learned from the

[1]Others include confidential computing and differential privacy. The former relies on hardware-based isolation for computation, and the latter adds controlled noise to the dataset for individual privacy.

cryptographically protected data. It is considered the ultimate solution for remote data protection because of its strong guarantees based on mathematical hard problems. However, its application in real-world scenarios is challenging, which is not only because of its high computational cost. This thesis identifies other reasons and proposes how to improve existing techniques to resolve/mitigate them.

First, we revisit secure multiparty computation (MPC), focusing on its security guarantees. We focus on the fact that MPC does not consider *input correctness* while guaranteeing *input privacy* and *computation correctness*. That is, it is guaranteed that the execution of a secure MPC protocol does not leak any information about input data, i.e., input privacy, and outputs the correct computation result over the input data, i.e., computation correctness. However, it is not considered whether the input data is "correct" or "valid", i.e., the correctness of the input is typically not considered. This is critical, especially when a specific input condition exists for computation. We investigate this issue using a special form of MPC, private set intersection (PSI), with input size limits, i.e., a specific size within a range is required to obtain the intersection result.

Next, we propose considering dynamic settings with advanced adversary models to apply MPC in today's complicated large distributed systems. Such large systems are managed by specialists, focusing on security and reliability, and one of the commonly used management strategies is *moving target defense (MTD)*. It adds system changes to increase uncertainty and complexity for attackers. Moreover, the specialists can manage each system entity so that it can be rebooted to a clean state, reducing the chance of compromising everyone in the systems. On the other hand, most MPC protocols characterize the adversary's corruption capability with a threshold, i.e., an adversary can corrupt up to a threshold number of parties. However, considering complex modern distributed systems, more generalized corruption scenarios need to be assumed rather than the number of corruptible parties. We enhance existing MPC protocols to apply the features above.

Lastly, we explore the application of cryptographic techniques to a specific use case, genomic tests, and claim to consider both security and privacy. Due to the nature of genomic data, privacy has gained a lot of attention, and many researchers suggest applying various cryptographic computing techniques for genomic privacy. On the other hand, although equally important, genomic security, such as authenticity and integrity, is often assumed, while lack of those may lead to wrong test results. Furthermore, considering the massive size of genomic data, efficiency is essential for real-world applications. To this end, we propose to balance security, privacy, and efficiency in range query-based genomic tests.

1.1 Summary of Research Contributions

To summarize, this dissertation makes the following contributions:

- For a verified RoT in mid-range MCUs, Chapter 2 provides $\mathcal{P}$ARseL, a provable attestation RoT over a formally verified microkernel, seL4. We verify its security guarantees via computer-aided formal verification and provide formally verified implementation;
- To maintain data privacy in low-end MCUs, Chapter 3 formalizes the notion of *Privacy-from-Birth* (*PfB*) and presents VERSA, a verified remote sensing authorization architecture for low-end MCUs. We also formally verify that VERSA realizes *PfB* goals;
- For input correctness in PSI with input size limits, Chapter 4 constructs protocols to avoid two types of malicious inputs, duplicated and fake elements, to bypass the lower-bound requirement, and merges them with existing protocols for the upper-bounds;
- For advanced system and adversary models, Chapter 5 adds several protocols to existing MPC schemes so that they can adapt to the MTD with dynamically changing participants and corruption scenarios; and

- Chapter 6 suggests ways to balance security and privacy in genomic range query testings using cryptographic computing techniques and extends them to private substring matching-based genomic testings;

1.2 Acknowledgement of Collaborative Work

All work presented in this thesis results from collaborative efforts and contributions. The detailed contributions of the collaborators for each chapter follow.

Chapter 2 is based on the publication "$\mathcal{P}$ARseL: Towards a Verified Root-of-Trust over seL4 " [111]. While everyone was involved in designing the architecture, Sashidhar Jakkamsetti mainly implemented the boot-time component and ran the evaluation over SabreLite [123]. Norrathep Rattanavipanon and Ivan De Oliveira Nunes provided valuable guidance on formal verification, and Norrathep Rattanavipanon additionally shared his knowledge on seL4 and HYDRA [135]. Gene Tsudik initiated the main idea of this project and provided insightful guidance and feedback throughout the project.

Chapter 3 is based on the publication "*Privacy-from-Birth*: Protecting Sensed Data from Malicious Sensors with **VERSA**" [112]. While everyone was involved in designing and formally verifying the security, Ivan De Oliveira Nunes led the project and provided valuable guidance on formal verification and low-end IoT devices throughout the project. Sashidhar Jakkamsetti additionally contributed to the synthesis of the RTL description of Hardware-Monitor and its deployment on the FPGA board. Gene Tsudik provided the initial idea of this project and insightful feedback throughout the project.

Chapter 4 is based on the publication "Element Distinctness and Bounded Input Size in Private Set Intersection and Related Protocols" [78]. While everyone was involved in protocol constructions, Xavier Carpent contributed the PoED protocol, and Gene Tsudik provided the main idea of the AD-APSI protocol. Xavier Carpent and Gene Tsudik gave constructive feedback throughout the project.

Chapter 5 is based on the publication “Communication-Efficient (Proactive) Secure Computation for Dynamic General Adversary Structures and Dynamic Groups" [133]. This project is part of the internship at SRI International, which was mainly led by Karim Eldefrawy. Throughout the duration of the project, he gave valuable feedback on protocol contructions. Rafail Ostrovsky and Moti Yung provided constructive feedback on an extension to MSP-based MPC and real-world motivation.

Chapter 6 is based on the publication “Balancing Security and Privacy in Genomic Range Queries" [176]. This is an extended work from [124] to apply its main idea to the size- and position-hiding private substring matching and sparse integer problems. Ercan Ozturk provided valuable guidance on implementation and paper writing, and also contributed on performance optimization. Gene Tsudik initiated the main idea to extend this project and provided insightful feedback throughout the duration of the project.

Chapter 2

$\mathcal{P}$ARseL: Towards a Verified Root-of-Trust over seL4

We propose to build a provable root-of-trust for attesting remote IoT/embedded devices to ensure their software integrity and guarantee secure data generation. This chapter first targets the mid-range devices, between the low-end and high-end devices, which usually run multiple processes over an OS or a microkernel with memory management hardware. We suggest a provable design over a formally verified microkernel, seL4, and verify its security properties via computer-aided formal verification.

2.1 Introduction

Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) devices have become ubiquitous in modern life, including households, workplaces, factories, agriculture, vehicles, and public spaces. They often collect sensitive information and perform safety-critical tasks, such as monitoring vital signs in medical devices or controlling traffic lights. Given their importance and popularity, these devices are attractive targets for attacks, such as the Colonial Pipeline attack in the American energy grid [138] and Ukraine power grid hack [305].

Attacks are generally conducted via software exploits and malware infestations that result in device compromise. Remote Attestation ($\mathcal{R}\mathsf{A}$) is a security service for detecting compromises on remote embedded devices. It allows a trusted entity (Vrf) to assess the software integrity of an untrusted remote embedded device (Prv). $\mathcal{R}\mathsf{A}$ serves as an important building block for other security services, such as proof of execution [110, 120], control-flow and data-flow attestation [26, 121, 115, 304, 284, 149], and secure software updates [39, 113].

Many prior $\mathcal{R}\mathsf{A}$ techniques (e.g., [108, 195, 67, 238]) focused on low-end devices, that run one simple application atop "bare metal". For example, SANCUS [238] is a pure hardware-based $\mathcal{R}\mathsf{A}$ architecture for low-end devices. Whereas, *VRASED* [108] is a hybrid (hardware/software) $\mathcal{R}\mathsf{A}$ architecture, while PISTIS [158] is a software-only one. All these architectures are unsuited for higher-end devices that execute multiple user space processes in virtual memory.

At the other end of the spectrum, enclaved execution systems [180, 93] implement $\mathcal{R}\mathsf{A}$ for user-level sub-processes (called enclaves) on high-end systems, e.g., desktops, laptops and cloud servers. However, they require substantial dedicated hardware support, thus making this approach unsuitable for the comparatively resource-constrained mid-range devices that we target in this work.

HYDRA [135] is an $\mathcal{R}\mathsf{A}$ architecture aimed at mid-range devices. It does not require additional hardware support other than an (often present) memory management unit (MMU) and a secure boot facility. HYDRA relies on a formally verified microkernel, seL4 [193], to provide strong inter-process memory isolation. However, neither HYDRA's implementation nor its integration with seL4, is formally verified. Also, HYDRA implements both attestation and untrusted application-defined functionalities in the same runtime process (see Sections 2.2.2 and 2.4.1). Thus, HYDRA's trusted computing base (TCB) implementation is application-dependent and, whenever an application changes, errors can be introduced within the TCB. As a consequence, even if the $\mathcal{R}$A component in HYDRA were verified, application bugs could still undermine its security due to the lack of guaranteed isolation. Unfortunately, moving away from this model also introduces non-trivial architectural challenges (see Section 2.4.2), requiring a clean-slate trust model.

Motivated by the above, this work re-visits HYDRA trust model and proposes $\mathcal{P}$ARseL: $\underline{\text{P}}$rovable $\underline{\text{A}}$ttestation $\underline{\text{R}}$oot-of-Trust over $\underline{\text{seL}}$4 Microkernel – a design that separates user-dependent components from the $\mathcal{R}$A TCB. This new model addresses the aforementioned challenges, leading to proper isolation, and facilitates formal verification. Specifically, we use formal verification to prove security properties for the (now isolated) root-of-trust in $\mathcal{P}$ARseL. Proven properties include memory safety, functional correctness, and secret independence. We then deploy and evaluate $\mathcal{P}$ARseL verified C implementation (atop seL4) on a commodity prototyping board, SabreLite [123]. $\mathcal{P}$ARseL implementation is publicly available at [33].

Organization: Section 2.2 overviews background, followed by our goals and assumptions in Section 2.3. $\mathcal{P}$ARseL design is presented in Section 3.6 and its implementation details are in Section 2.5, along with formal verification. $\mathcal{P}$ARseL security analysis follows in Section 2.6 and limitations are discussed in Section 2.7. Finally, this chapter concludes with the related work overview in Section 2.8.

2.2 Background

This section provides background information on seL4, $\mathcal{R}$A, and formal verification tools. Given familiarity with these topics, it can be skipped with no loss of continuity.

2.2.1 `seL4` Microkernel [193]

`seL4` is a member of the L4 family of microkernels. Functional correctness of its implementation, including the `C` code translation [275], is formally verified, i.e., the behavior of `seL4` `C` implementation strictly adheres to its specification. To provide provable memory isolation between processes, `seL4` implements a *capability*-based access control model. A capability is an unforgeable token that represents a set of permissions that define what operations can be performed on the associated object at which privilege level. This enables fine-grained access control by granting or revoking specific permissions to individual components or threads. Also, user-space applications cannot directly access or modify their own capabilities, because each capability is stored in *Capability Space* (CSpace) which is managed by `seL4`. User applications interact with `seL4` through system calls and operate on their capabilities indirectly. Since `seL4` enforces strict access control and authorization checks for system calls, `seL4` retains the ultimate authority over capabilities and their allocation, revocation, and manipulation.

As a micro-kernel, `seL4` provides minimal functionality to user-space applications. For example, inter-processes' data sharing requires the establishment of *inter-process communication* (IPC) by invoking *endpoint* objects, that act as general communication ports. Each *endpoint* is given a capability by assigning it a unique identifier, called a "*badge*", which identifies the sender process during communication. Each process is represented in `seL4` by its *Thread Control Block* object which includes its associated CSpace and *Virtual-address Space* (VSpace) and (optionally) an *IPC buffer*. CSpace contains the capabilities owned by the process. VSpace represents the virtual memory space of the process, defining the mappings between virtual addresses (used by the process) and physical memory. IPC buffer is a fixed region of memory reserved for IPC. To send or receive messages, a process places them in its *message registers* which are put in the IPC buffer. Then, it invokes capabilities within its CSpace via system calls, such as `seL4_Send` and `seL4_Recv` or combinations/variants, such as `seL4_Call`, `seL4_Reply`, or `seL4_ReplyRecv`.

2.2.2 $\mathcal{R}$A & HYDRA

As mentioned earlier, the goal of $\mathcal{R}$A is for a trusted Vrf to securely assess the software integrity of an untrusted remote device, *Prv*. To do so, Vrf issues a unique challenge to *Prv*. Upon its receipt, *Prv* computes an authenticated measurement of its software state. This measurement also includes Vrf's challenge and it is computed using either a *Prv*-Vrf shared secret or a *Prv*-unique private key for which Vrf knows the corresponding public key. *Prv* returns the measurement to Vrf which authenticates it and decides on *Prv*'s state (i.e., compromised or not).

To the best of our knowledge, the only relevant prior result that attempted to fuse $\mathcal{R}$A with `seL4` is HYDRA [135]. It operates in three phases: *Boot*, `seL4` *Setup*, and *Attestation*. In Boot phase, *Prv* executes a ROM-resident secure boot procedure that verifies `seL4` binary. Upon verification, *Prv* loads all executables into RAM and passes control to the kernel. In `seL4` Setup phase, the kernel sets up the user space and initializes the first process, *attestation process* (`AP`). The kernel then hands control to `AP` after assigning all capabilities for all available memory locations to `AP` and verifying `AP`'s binary. `AP` is then responsible for spawning all user processes with lower scheduling priorities and user-defined capabilities, initializing the network interface, and waiting for subsequent attestation requests. Finally, in the Attestation phase (which comprises the rest of the runtime), upon receiving a Vrf-issued attestation request for a particular user-space process, `AP` computes an HMAC [53] of the memory region of that process, using a symmetric key pre-shared with Vrf, and returns the result to Vrf.

HYDRA `AP` implements several system functions that are unrelated to $\mathcal{R}$A functionality. While this approach simplifies *Boot* and `seL4` *Setup* phases, it also makes HYDRA verification challenging. We further discuss this in Section 2.4.1.

2.2.3 *F**, *Low**, and *KaRaMeL*

*F** [285] is a general-purpose functional programming language with an effect system facilitating program verification. Developers can write a program and its specifications in *F**, representing that program's computational and side effects, and then formally verify that it adheres to those specifications using automated theorem-proving techniques. The type system of *F** includes dependent types, monad effects, refinement types, and the weakest precondition calculus, which together allow describing precise and compact specifications for programs using Hoare logic [171]. For example, Figure 2.1 shows two simple functions in *F**. While both take an integer as input and output its absolute value, `abs_pos` "requires" the input integer to be positive as **pre-condition** and "ensures" that the result equals the absolute value of `x` as **post-condition**. The pre-condition of `abs_pos` can be instead written with refinement type input: `(x :  int {x > 0})`. Both have the `Pure` effect, meaning that they are stateless functions, guaranteeing deterministic results and no side effects. `Tot` is a special type of `Pure` with no pre-condition, i.e., it is defined for all possible values of input so that it terminates and returns an output.

```
let abs (x : int) : Tot int
  = if x >= 0 then x else -x

let abs_pos (x : int) : Pure int
(requires x > 0) (ensures fun y -> y = abs x) = x
```

Figure 2.1: Example Functions in F^*

To support stateful programs, F^* provides `ST` effect with the form:

```
ST (a:Type) (pre:s→Type) (post:s→a→s→Type)
```

This means: for a given initial memory "`h0:s`" that satisfies pre-condition "`(pre h0)` is true", a computation "`e`" of type "`ST a (requires pre) (ensures post)`" outputs a result "`r:a`" and updates existing memory to final memory "`h1:s`", which satisfies the post-condition "`(post h0 r h1)` is true".

One notable feature of F^* is *machine integers* and arithmetic operations on them. Machine integers model (un)signed integers with a fixed number of bits, e.g., `uint32, int64`, and `FStar.Int.Cast` module offers conversions between these types. Using machine integers ensures that input and computation result values fit in the given integer bit-width. In addition, one can express their secrecy level, denoted by '`PUB`' or ' `SEC`'. The former are considered public and can be safely shared, while the latter are considered secret, i.e., F^* guarantees no leaks for them. Specifically, it prevents information leakage from timing side-channels and clears all memory that contains `SEC`-level integers when they are no longer needed.

Low* [258] is a subset of F^*, targeting a carefully curated subset of `C` features, such as the `C` memory model with stack- and heap-allocated arrays, machine integers, `C` string literals, and a few system-level functions from the `C` standard library. To support these features, *Low** refines the memory model in F^* by adding a distinguished set of regions modeling `C` call stack – so-called *hyper-stack* memory model. For modeling `C` stack-based memory management mechanism, *Low** introduces a region called `tip` to represent the currently active stack frame and relevant operations, such as `push` and `pop`. *Low** also introduces the `Stack` effect with the form below, to ensure that the stack tip remains unchanged after any pushed frame is popped and the final memory is the same as the initial memory:

```
Stack a pre post = ST a (requires pre) (ensures

     (λ h0 r h1 → post h0 r h1 ∧

     (tip h0 = tip h1) ∧ (∀ x.  x ∈ h1 ⇔ x ∈ h0)))
```

Programmers writing code in *Low** can utilize the entire F^* for proofs and specifications. This is because proofs are erased at compile-time and only low-level *Low** code is left and compiled to C code. Verified *Low** programs can be efficiently extracted to readable and idiomatic C code using the *KaRaMeL* [5] compiler tool (previously known as *KreMLin*). *KaRaMeL* implements a translation scheme from a formal model of *Low**, λow*, to CompCert Clight [61]: a subset of C. This translation preserves trace equivalence with respect to the original F^* semantics. Thus, it preserves the functional behavior of the program without side channels due to memory access patterns that could be introduced by the compiler. The resulting C programs can be compiled with CompCert or other C compilers (e.g., GCC, Clang).

2.2.4 HACL* Cryptographic Library [312]

HACL* [312] is a formally verified cryptographic library written in *Low** and compiled to readable C using *KaRaMeL*. Each cryptographic algorithm specification is derived from the published standard and covers a range of properties, including:

- *Memory safety*: verified software never violates memory abstractions so that it is free from common vulnerabilities due to reads/writes from/to invalid memory addresses, e.g., buffer overflow, null-pointer dereferences, and use-after-free.
- *Type safety*: software is well-typed and type-related operations are enforced, i.e., HACL* code respects interface, and all the operations are performed on the correct types of data.
- *Functional correctness*: input/output of the software for each primitive conform to simple specifications derived from published standards.
- *Secret independence*: observations of the low-level behavior, such as execution time or accessed memory addresses, are independent of secrets used in computation, i.e., the implementation is free of timing side-channels.

2.3 Goals & Assumptions

2.3.1 System Model

We consider Prv to be a mid-range embedded device equipped with an MMU and a secure boot facility[1]. Devices in this class include I.MX6 Sabre Lite [123] and HiFive Unleashed [279] (on which seL4 is fully formally verified [271]). Following seL4 verification axioms, Prv is limited to one active CPU core, i.e., it schedules multiple user-space processes, though only one process is active at a time. We assume that secure boot is correctly enabled prior to device deployment.

$\mathcal{P}$ARseL TCB consists of seL4 micro-kernel, the first process loaded by the micro-kernel in user-space, called *Root Process* (RP), and *Signing Process* (SP), also in user-space (details in Section 3.6). Vrf wants to use $\mathcal{R}$A to establish a secure channel with a particular attested user-space process. To facilitate this, $\mathcal{P}$ARseL attestation response can also include a unique public key associated with the attested process. Vrf can then use the secure channel to communicate sensitive data with the attested process, after verifying its integrity via $\mathcal{R}$A.

$\mathcal{P}$ARseL provides a *static* root of trust for measurement of user-space process, i.e., the binary of processes are measured at their loading time. This is plausible because $\mathcal{P}$ARseL, by design, enforces that no new user process is spawned during runtime and no modifications on code occur without rebooting the device. On the other hand, $\mathcal{P}$ARseL design allows the user-process updates without modifying $\mathcal{P}$ARseL TCB. However, any updates require the device to reboot to re-measure the updated programs, which limits the scalability. We further discuss this limitation and possible alternatives in Section 2.7.

$\mathcal{P}$ARseL design is agnostic to the choice of cryptographic primitives. In fact, $\mathcal{P}$ARseL can

[1]Although common in mid-range embedded devices, secure boot requirement can be relaxed with weaker adversary model where $\mathcal{A}$dv does not have physical access to Prv and the initial deployment of seL4 and $\mathcal{P}$ARseL TCB on Prv is trusted.

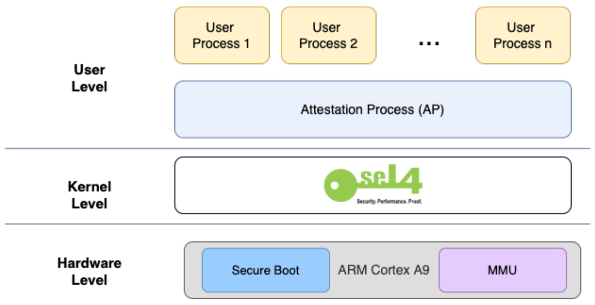
support both (1) symmetric-key cryptography where Prv and Vrf share a master secret from which a subsequent symmetric key can be derived, or (2) public-key cryptography where Prv has a private signing key whose public counterpart is securely provisioned to Vrf. In both cases, the required keys can be hard-coded as part of the $\mathcal{P}$ARseL TCB prior to Prv deployment.

2.3.2 Adversary Model

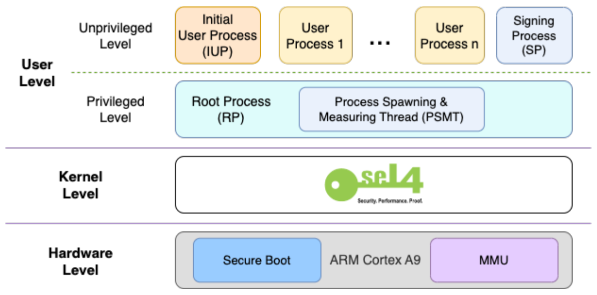
Based on the $\mathcal{R}$A taxonomy in [27], four main types of $\mathcal{A}$dv are:

1. *Remote*: exploits vulnerabilities in Prv software and injects malware over the network;
2. *Local*: controls Prv's local communication channels; may attempt to learn secrets leveraging timing side-channels;
3. *Physical non-intrusive*: has physical access to Prv and attempts to overwrite its software through legal programming interfaces (e.g., via J-TAG/replacing an SD card).
4. *Physical intrusive*: performs invasive physical attacks, physical memory extraction, firmware tampering, and invasive probing, e.g., via various physical side-channels.

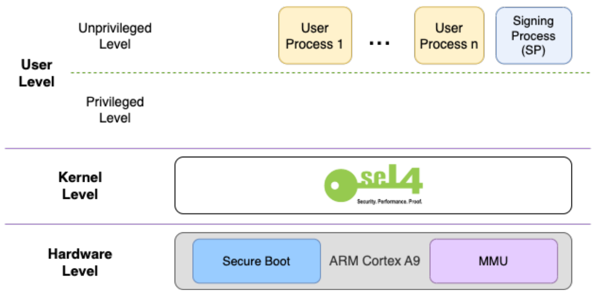
We consider all except (4), protection against which can be obtained via standard physical security measures [260]. This assumption is in line with related work on trusted hardware architectures for embedded systems [252, 67, 108, 195]. In terms of capabilities, if $\mathcal{A}$dv compromises a user-space process in Prv, it takes full control of that user-space process, i.e., it can freely read and write its memory and diverge its control flow. We assume user-space processes as untrusted and therefore compromisable, except for $\mathcal{P}$ARseL TCB. Finally, we assume that $\mathcal{A}$dv can trigger interrupts at any time.



(a) HYDRA Execution Levels



(b) $\mathcal{P}$ARseL on Boot



(c) $\mathcal{P}$ARseL at Runtime

Figure 2.2: Comparison of HYDRA (top) and $\mathcal{P}$ARseL Execution Levels on Boot (bottom-left) and at Runtime (bottom-right)

2.4 Verified Root-of-Trust over seL4 ($\mathcal{P}$ARseL)

This section starts out by describing HYDRA and identifying its shortcomings. It then justifies our approach and discusses how $\mathcal{P}$ARseL realizes it.

2.4.1 HYDRA & Its Limitations

As mentioned above, HYDRA is composed of *Boot*, seL4 *Setup*, and *Attestation* phases. AP is the very first user-space process to run after seL4 *Setup*. As such, AP possesses all capabilities for all available memory and system resources. It is responsible for creating and managing all other processes, ensuring proper configuration of capabilities, and performing $\mathcal{R}$A.

We argue that this design results in an excessive and application-dependent TCB. First, formally verifying the implementation of AP is extremely challenging since it requires a giant manual proof effort that might not be achievable in practice. However, without formal verification, there is no guarantee that AP is vulnerability-free and correct. Since AP has all user-space capabilities, its compromise would lead to a breach of all seL4 isolation guarantees provided. Even assuming the feasibility of AP formal verification, process-spawning component of AP strictly depends on the specific user application configuration. This is so that AP can properly assign custom (user-defined) access control configurations to each application process. Thus, whenever an application changes, AP implementation needs to be adjusted accordingly. Doing so modifies the AP's previously verified TCB. It is clearly infeasible to re-verify AP implementation for all possible application-dependent configurations.

2.4.2 Design Rationale

To enable verifiability, the TCB size at runtime must be reduced, by identifying and removing unnecessary functionalities from the privileged AP process. HYDRA AP functionalities are:

① Spawning all user processes with memory/capability settings;

② Communication with Vrf over the network interface for $\mathcal{R}$A;

③ Attestation of all user processes;

First, we observe that including ② in the TCB yields no benefit since the security of $\mathcal{R}$A does not depend on the availability/integrity of the communication interface. Thus, we move this functionality out of the TCB and handle $Prv \leftrightarrow Vrf$ communication in a separate user-space process. Second, ① performing initialization tasks that are not needed at runtime (i.e., post-boot). Third, further sub-dividing ③:

③-(a) Measuring (reading) the code binary for each user process;

③-(b) Signing the measurement with a private key and a challenge from Vrf;

③-(a) can be also done once, assuming that the code does not change post-boot (as mentioned in Section 2.3.1). Thus, these components can be terminated after completion, at boot time, which effectively limits these components' exploitable time window to boot time.

Also, ① can be sub-divided into:

①-(a) Storing access control capabilities for all processes to be spawned;
①-(b) Spawning the user processes based on given access control capabilities;

To separate all user-dependent components from the TCB, a separate user process can perform ①-(a) and communicate with AP for ①-(b). Or it can be even just a configuration file that AP can read from. Finally, ③-(b) must be active at runtime to process $\mathcal{R}$A requests from Vrf, which represents the only potential remaining entry point for $\mathcal{A}$dv. To close this gap, this operation can be assigned to a tiny dedicated process, called *Signing Process* (SP). Due to its small size and independence of user-defined components, verifying SP is now relatively easier.

2.4.3 $\mathcal{P}$ARseL Design

Combining all the above, Figure 2.2 shows $\mathcal{P}$ARseL execution levels at boot- and at runtime, as compared to HYDRA. $\mathcal{P}$ARseL subdivides seL4 user-space into two execution levels: *Privileged* and *Unprivileged*. We refer to the privileged initial user process as *Root Process* (RP) which has a thread (for the roles of ①-(b) and ③-(a)), called *Process Spawning & Measuring Thread* (PSMT). In contrast, the processes at the unprivileged level have restricted capabilities assigned by RP. Unprivileged processes include *Initial User Process* (IUP) (for ①-(a)), SP, and user-defined processes (UP-s). Capabilities of any process at the unprivileged level do not allow access to any memory not explicitly assigned to that process. RP (including

PSMT) and IUP are terminated at the end of boot phase, and only UP-s and SP remain during run-time, as shown in Figure 2.2c.

2.4.4 $\mathcal{P}$ARseL Execution Phases

$\mathcal{P}$ARseL has seven execution phases in total: three on boot and four at runtime. The three phases in the boot-time are as follows:

(Secure) Boot: The boot-loader verifies, loads, and passes control to, seL4. Thereafter, seL4 verifies the integrity of $\mathcal{P}$ARseL TCB, i.e., the software that runs in RP, and passes control to RP, once verification succeeds.

Process Spawn: RP spawns PSMT as a thread. PSMT spawns IUP as an unprivileged process and establishes an IPC channel with it. Once spawned, IUP sends the configuration of user processes and their process ID-s ($\mathsf{P_{ID}}$-s) to PSMT via IPC. Upon receiving a request, PSMT spawns a new process according to received capabilities. It also ensures that these capabilities are valid, not containing the write capability for its own code segment. Finally, it spawns and sets up an IPC channel with SP. Once all processes are spawned, PSMT sets up an IPC *endpoint* for each user process, assigns a unique *badge* for each endpoint, and associates this unique badge with $\mathsf{P_{ID}}$.

Measurement: While spawning each user process, PSMT also measures (via hashing) its code segment, and stores the results in *measurement map* (mmap) with the $\mathsf{P_{ID}}$ as the lookup key. Once all measurements are complete, PSMT sends the entire mmap to SP through IPC, and RP (including PSMT) is terminated.

Then at runtime, once *Prv* is booted and in a steady state, it repeatedly executes the remaining four phases:

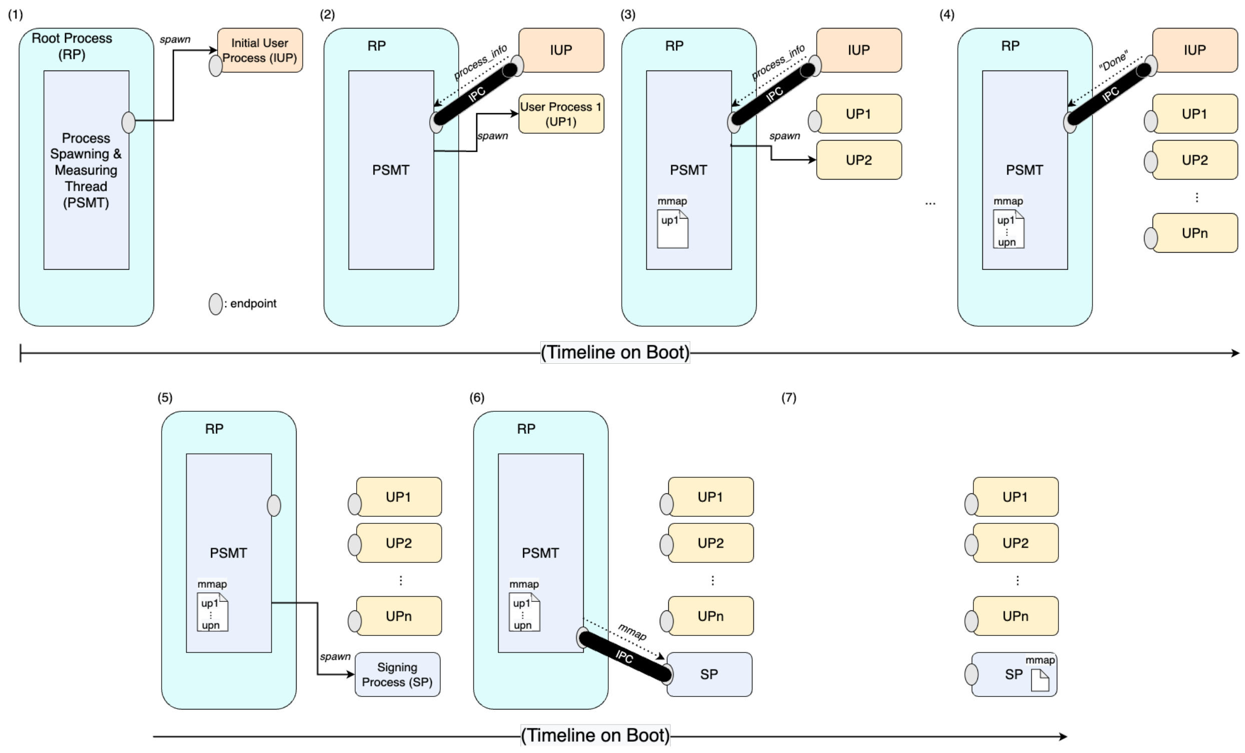


Figure 2.3: Sequence of $\mathcal{P}$ARseL Execution Phases on Boot (After Secure Boot Checks)

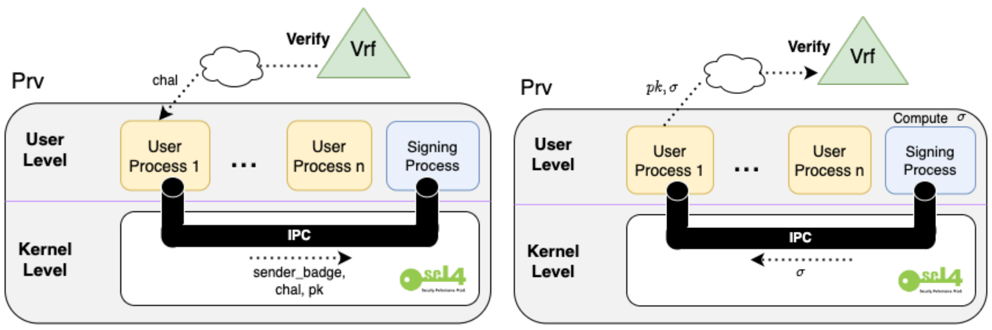


Figure 2.4: Sequence of $\mathcal{P}$ARseL Execution Phases at Runtime

Listen: SP listens to receive messages from user processes through the endpoint set up in the boot phase.

Request: Once a user process, UP, receives an attestation request from Vrf with a fresh challenge, $\mathcal{C}$hal, UP transmits the request to SP through IPC system calls. The request message includes $\mathcal{C}$hal and the public key of UP, pk.

Sign: Upon receiving a request, SP identifies the sender process, UP, from the activated endpoint badge and derives P_{ID}[2]. It then retrieves UP's measurement m_{UP} from mmap using P_{ID} and signs m_{UP} along with the request message using its secret key, $\mathcal{K}$. i.e., The signature is computed as:

$$\sigma := Sign(\mathcal{K}, Hash(\mathcal{C}\mathsf{hal}||\mathsf{pk}||\mathsf{m}_{\mathsf{UP}})) \tag{2.1}$$

Response: SP responds σ to UP via IPC. UP forwards σ and pk to Vrf. Finally, after successful σ verification, Vrf establishes a secure channel with UP using received pk.

Figures 2.3 and 2.4 show the aforementioned $\mathcal{P}$ARseL execution phases on boot and at run-time, respectively.

[2]Note that seL4 guarantees that UP cannot forge its own endpoint badge. Therefore, the attested UP is the same process that provides pk to SP.

2.5 $\mathcal{P}$ARseL Implementation & Verification

2.5.1 Implementation Details

Implementation of Root Process (RP)

Once seL4 passes control to RP, RP initializes user space by creating necessary boot-time objects, such as CSpace, VSpace, and a memory allocator. Then, it initializes PSMT by creating a new thread control block object, a memory frame for its IPC buffer, a new page table, and a new endpoint object. Next, RP maps the page table and IPC buffer frame into the VSpace and configures a badge for the endpoint and thread control block priority. RP then sets up the thread-local storage (for its own storage area) and spawns PSMT. Finally, it waits for PSMT to complete and send ACK.

Implementation of Process Spawning Thread (PSMT)

Once spawned, PSMT creates SP by assigning it a new set of virtual memory, configuring it with two endpoints, and associating a unique badge for each endpoint. SP uses one endpoint for IPC with SP and the other for UP-s. PSMT similarly creates IUP, establishes an IPC between itself and IUP, and spawns IUP. Then, PSMT waits for a request from IUP.

A request includes all the specifications of UP to be spawned, such as P_{ID}, binary location, and capabilities to system resources. Once receiving the request, PSMT first ensures that the requested capabilities do not contain the write capability to UP's binary and then initializes UP accordingly. Next, PSMT computes its measurement, using a hash algorithm (e.g., SHA2-256 [280]) in HACL*, and stores it in mmap in order. PSMT uses a counter to make sure the number of spawned processes does not exceed the size of mmap and assigns a badge based on the counter to make it unique per UP endpoint. Finally, PSMT spawns UP and waits for the next request. Once receiving the “Done” signal from `IUP`, `PSMT` sends the entire `mmap` to `SP` via IPC, waits for `IUP` to finish its tasks (if any), and sends an ACK to `RP`.

Implementation of Initial Process (`IUP`)

In $\mathcal{P}$ARseL, all the user process information is consolidated into a configuration file at compile-time. `IUP` first parses this file and loads its information to a local object. Then, for each `UP`, `IUP` sends a spawn request to `PSMT` with its P_{ID}, and waits for an acknowledgment. After all the `UP`-s are spawned, `IUP` sends the “Done” signal to `PSMT` and finishes its remaining tasks (if any), before terminating itself. Note that if `IUP` contains no tasks other than requesting to spawn `UP`-s, then `PSMT` can directly read the configuration file and spawn user processes, instead of having a separated `IUP`.

Implementation of Signing Process (`SP`)

`SP` has two roles: (1) collecting all the `UP`-s measurements from `PSMT` at boot-time, and (2) repeatedly processing $\mathcal{R}$A requests at runtime. Once `SP` is spawned by `PSMT` during boot-time, `SP` uses `seL4` system calls to receive the entire `mmap` via IPC in the following way:

1. Using `seL4_Recv()`, `SP` listens for measurement message (P_{ID}, m) from `PSMT`’s badge.
2. `SP` uses `seL4_GetMR()` to unmarshal the message and copies (P_{ID}, m) to `mmap`.
3. Using `seL4_Reply()`, `SP` sends ’0’ (as a ACK).

This process is repeated until all the measurements are received from `PSMT`. In the following section, we describe the verified implementation of `SP`’s runtime phase.

2.5.2 Formally Verification of $\mathcal{P}$ARseL Runtime Implementation

We describe the implementation of runtime $\mathcal{P}$ARseL TCB in *Low**, with verified properties, and how to convert it to C code, preserving the verified properties, using *KaRaMeL*.

Verifying Properties

Recall that SP runs the infinite loop of (*Listen, Request, Sign, Response*) phases (see Section 2.4.4). To verify SP, we prove the following invariant properties for this infinite loop: **functional correctness, memory safety**, and **secret independence**.

Functional correctness ensures that each loop iteration performs all the functionalities as intended. In this context, it means each iteration of SP correctly computes the signature according to Equation (2.1) for the given input and returns the computed result without modifying SP internal states. *Memory safety* and *secret independence* guarantee that no additional information beyond the signature result is leaked from SP. This applies to both memory-based leakages as well as timing side channels. In Section 2.6, we show that these three properties are sufficient to provide secure $\mathcal{R}$A in $\mathcal{P}$ARseL.

Runtime SP Implementation in *Low** and C

To prove these properties, we first specify all seL4 APIs used by SP in *Low**. Then, we implement the *Low** code for all SP execution phases and integrate it with the *Low**-specified seL4 APIs and HACL* verified cryptographic functions. Next, we formally verify the combined implementation via *Low** memory model, intermediate assertions, and post-condition of the SP execution. Finally, we convert the final *Low** code to C using the verified *KaRaMeL* compiler.

[Specifying seL4 APIs in *Low**] While SP is implemented in *Low**, the functional correctness of seL4 implementation (including system calls) is verified with a different formal specification language called Isabelle/HOL [239]. Hence, we represent them as axioms, using the construct '`assume val`' in F^*. F^* type checker accepts the given assumption without attempting to verify it, and these axioms are converted to '`extern`' in the generated C code. We specify the input/output of each seL4 system call with required type definitions.

For example, Figure 2.5 shows in order, the original C code for a system call, `seL4_GetMR` from seL4 APIs, corresponding *Low** implementation as an axiom, and the generated C code using *KaRaMeL*. `seL4_GetMR` has an integer input `i` and simply outputs the `i`-th element of `msg` array in `seL4_IPCBuffer` with type `seL4_Word`. Including the new type `seL4_Word` for `uint64`[3], all the definitions or structs in seL4 (lines 1-12 of original C code) are properly converted into *Low** (lines 1-17 of *Low** code). Note that since there is no concept of the *global variable* in functional programming, all global variables or structs used in SP are represented in `state` type (lines 5-7 of *Low** code), initialized in `st_var` (lines 8-15) and defined in function `st` (lines 16-17). Once the *Low** axiom is compiled with *KaRaMeL*, generated C code only contains one line of declaration (line 12 of generated C code) without implementation. The rest of seL4 system calls used in SP, `seL4_Recv`, `seL4_Reply`, and `seL4_SetMR`, are similarly written as axioms.

[Writing SP in *Low**, combining HACL* library] The *Sign* phase is implemented using cryptographic operations in HACL* which is also implemented in *Low** and formally verified according to their specification. Thus, three HACL* functions for concatenation, hash, and sign, are integrated into one signing function for Equation (2.1). We use HMAC [53] for the symmetric signing algorithm with SHA2-256 [280] hash function and EdDSA [57] for the asymmetric one. Runtime SP with the four execution phases is implemented by combining

[3]It is defined either `uint32` or `uint64` depending on the underlying architecture, and the example code is shown with `uint64 seL4_Word`.

```c
#define _seL4_int64_type    long long int
typedef unsigned _seL4_int64_type   seL4_Uint64;
typedef seL4_Uint64 seL4_Word;
typedef struct seL4_IPCBuffer_ {
    seL4_Word msg[seL4_MsgMaxLength]; // seL4_MsgMaxLength = 120
} seL4_IPCBuffer __attribute__((__aligned__(sizeof(struct seL4_IPCBuffer_))));
extern __thread seL4_IPCBuffer *__sel4_ipc_buffer;
__thread __attribute__((weak)) seL4_IPCBuffer *__sel4_ipc_buffer;
LIBSEL4_INLINE_FUNC seL4_IPCBuffer *seL4_GetIPCBuffer(void)
{
    return __sel4_ipc_buffer;
}
LIBSEL4_INLINE_FUNC seL4_Word seL4_GetMR(int i)
{
   return seL4_GetIPCBuffer()->msg[i];
}
```

```
type seL4_Word = uint64
noeq type seL4_IPCBuffer = {
  msg : mbuffer seL4_Word 120;
}
noeq type state = {
  ipc_buffer: ipc:seL4_IPCBuffer;
}
let st_var: state =
  let msg = B.gcmalloc HS.root (I.u64 0) 120ul in
  let ipc_buffer = {
    msg = msg;
  } in
  {
    ipc_buffer = ipc_buffer;
  }
val st (_:unit):state
let st _ = st_var
assume val seL4_GetMR
( i : size_t )
: Stack seL4_Word
( requires fun h0 -> (size_v i < 120) /\ (size_v i >= 0) )
( ensures fun h0 a h1 -> B.(modifies loc_none h0 h1) /\ a == B.get h1 (st ()).ipc_buffer.msg (v i))
```

```c
typedef uint64_t seL4_Word;
typedef uint64_t *seL4_IPCBuffer;
typedef struct state_s
{
  uint64_t *ipc_buffer;
} state;
state st_var;
state st()
{
  return st_var;
}
extern uint64_t seL4_GetMR(uint32_t i);
```

Figure 2.5: Simplified example seL4 API in original seL4 library (top), axiom in F^* (middle), and generated header file in C (bottom)

this signing function and the `seL4` axioms.

First, to receive/send a message through the IPC buffer or store intermediate computation results, we need some local `C` arrays in *Low**. For representing `C` arrays, *Low** provides the `Buffer` module. In *Low**, a buffer is a reference to a sequence of memory with a starting `index` and a `length`. We use `alloca` (or `create` from HACL*) for stack allocation, and retrieve/update the buffer contents using `index`/`upd` with the proper indices.

Then, since the *Sign* phase is in between two `seL4` system calls for *Request* and *Response* phases, proper type conversions are required. Specifically, `seL4` system calls use the type `seL4_Word` and HACL* functions require the `uint8` input type. To safely convert back and forth between `uint8` buffer and `seL4_Word` buffer (with big-endian), we use `uints_to_bytes_be` and `uints_from_bytes_be` of the `Lib.ByteBuffer` module in HACL*.

[Formal Verification] To verify the *functional correctness* of runtime `SP`, we first specify necessary pre-/post-conditions for each `seL4` axiom. For example, the *Low** code in Figure 2.5 shows that the function `seL4_GetMR` correctly returns with the `i`-th element of `msg` array in `seL4_IPCBuffer` (line 22). Also, some properties are needed to be specified to verify that `SP` internal states are not modified. In Figure 2.5, the post-condition `B.(modifies loc_none h0 h1)` indicates that no locations are modified from `seL4_GetMR` function call (line 22).

Next, we insert an assertion detailed in Figure 2.6 after the *Sign* phase to ensure the functional correctness of the signing function, i.e., it correctly computes the signature according to Equation (2.1).

```
// h0 is the initial memory state and h1 is the state right after the signing function call, using ST.get ()
assert ( B.as_seq h1 sign_result_u8 ==
            Spec.Ed25519.sign (B.as_seq h0 s.sign_key)
                (Spec.Agile.Hash.hash alg
                    (Lib.Sequence.concat #uint8 #64 #32
                        (Lib.Sequence.concat #uint8 #32 #32
                        (B.as_seq h chal) (B.as_seq h pk))
                    (B.as_seq h measurement_process))
                )
        );
```

Figure 2.6: Assertion for Functional Correctness of *Sign*, equation (2.1)

Finally, we check the invariance of $\mathcal{K}$ and `mmap` throughout the `SP` execution via intermediate assertions and the post-condition of the runtime `SP` function. Similar to the assertion above, it compares the $\mathcal{K}$ and `mmap` contents in the memory (`h`) after executing each function call with the ones in the initial memory (`h0`), specified in Figure 2.7. This invariance along with the post-conditions of `seL4` APIs and the assertion in Figure 2.6 implies the functional correctness of runtime `SP`.

```
assert (B.as_seq h0 s.mmap == B.as_seq h s.mmap);
assert (B.as_seq h0 s.sign_key == B.as_seq h s.sign_key);
```

Figure 2.7: Assertion for $\mathcal{K}$ and `mmap` invariance

For *memory safety*, we first implement all `SP` components with `Stack` effect, which prevents any memory leakage due to deallocated heap regions. We also check the "liveness" and "disjointness" of all buffers before they are referenced (via `live` and `disjoint` clauses), which prevents stack-based memory corruption. The former guarantees that a buffer must be properly initialized and not de-allocated (so "live") before it is used, whereas the latter ensures that all buffers used in `SP` are located in separate memory regions without any overlap. Lastly, we specify a post-condition for every function in `SP` to ensure that it modifies only the intended memory region. This can be done through the `modifies` clause with the form of `modifies s h0 h1`, which ensures that the memory `h1` after the function call may differ from the initial memory `h0` (before the function call) *at most* regions in `s`, i.e., no regions outside of `s` are modified by the function execution. For example, in Figure 2.5, `seL4_GetMR` function ensures not to modify any memory location (with '`loc_none`') in its post-condition (line 22).

Finally, for the *secret independence*, we use the same technique employed by HACL*. We use the secret machine integers for private values (i.e., $\mathcal{K}$), including all intermediate values, and do not use any branch on those secret integers. This ensures that the execution time or the accessing memory addresses are independent of the secret values so that the implementation is timing side-channel resistant.

[Generating `C` code using *KaRaMeL*] Finally, we carefully write a build system and generate readable `C` code from our verified *Low** code using *KaRaMeL*. It takes an F^* program, erases all the proofs, and rewrites the program from an expression language to a statement language, performing optimizations. If the resulting code contains only *Low** code with no closures, recursive data types, or implicit allocations, then *KaRaMeL* proceeds with a translation to `C`.

KaRaMeL generates a readable `C` library, preserving names so that one not familiar with F^* can review the generated code before integrating it into a larger codebase. For example, the refinement type (`b: B.buffer uint32 B.length b = n`) in *Low** is compiled to a `C` declaration (`uint32_t b[n]`), while referred to via (`uint32_t *`) as `C` pointer.

2.5.3 Secure Boot of seL4 and $\mathcal{P}$ARseL TCB

Similar to HYDRA, $\mathcal{P}$ARseL relies on a secure boot feature to protect against a physical $\mathcal{A}$dv attempting to re-program seL4 and $\mathcal{P}$ARseL TCB when *Prv* is offline. In HYDRA, this feature works by having a ROM boot-loader validate seL4 authenticity before loading it. Once seL4 is running, it authenticates the user-space TCB by comparing it to a benign hash value, hard-coded within the seL4 binary. Since HYDRA TCB is user-dependent, updating a user application implies a software update not only to the TCB but also to the seL4 binary that stores the TCB referenced hash value, which can be inconvenient in practice. Conversely, $\mathcal{P}$ARseL TCB is user-independent, allowing user applications to be updated directly without the need to modify $\mathcal{P}$ARseL TCB or seL4 binary.

2.5.4 Evaluation

Our source code including verification proofs is available at [33].

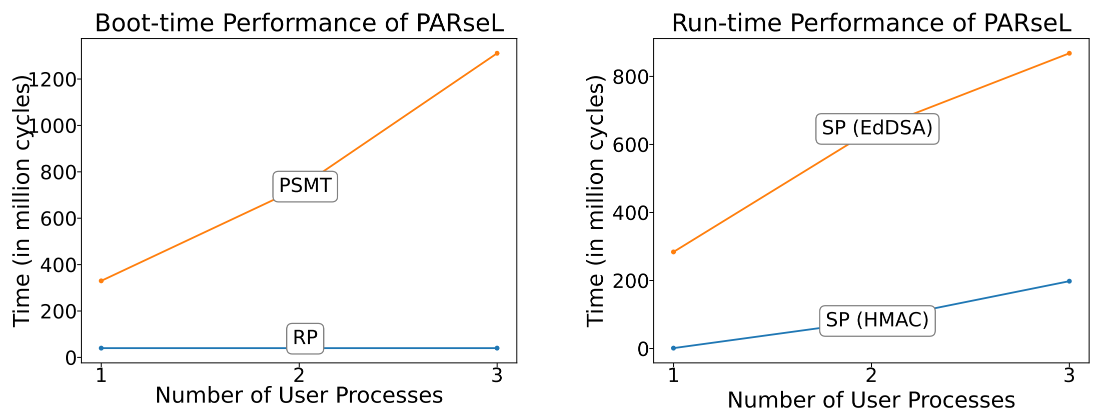


Figure 2.8: $\mathcal{P}$ARseL Performance while varying the number of spawned user processes (excluding SP)

Evaluation Setup

To demonstrate the practicality of $\mathcal{P}$ARseL, we developed our prototype on a commercially available hardware platform: SabreLite [123] – on which seL4 is fully verified [271] including all proofs for functional correctness, integrity, and information flow. SabreLite features an ARM Cortex-A9 CPU core (running at 1 GHz), with RAM of size 1 GB, and a microSD card slot (which we use to boot and load $\mathcal{P}$ARseL image). $\mathcal{P}$ARseL is implemented on seL4 version 12.0.1 (latest at the time of writing). Besides seL4 IPC kernel APIs, RP uses seL4 Runtime, seL4 Utils, and seL4 Bench user-space libraries (offered by seL4 Foundation) to implement PSMT process spawning procedure.

$\mathcal{P}$ARseL Performance

Figure 2.8 depicts the performance of $\mathcal{P}$ARseL on SabreLite. The left sub-figure shows the boot-time performance of RP and PSMT, and the right one shows the run-time performance of SP (using either HMAC or EdDSA). Reported results are averaged over 50 iterations. The size of each spawned process is ≈ 0.4 MB.

RP takes constant 40 ms (40 million cycles @ 1 GHz), as it initiates the device and spawns PSMT, independent of the number of UP-s spawned. The time taken for PSMT increases linearly to the number of UP-s, as expected because PSMT loads, measures, and spawns each UP sequentially. Spawning each 0.4 MB UP takes $\approx$ 150 ms. Concretely, when there are 3 UP-s, the boot-time of $\mathcal{P}$ARseL is 1.3s.

Using HMAC requires significantly fewer cycles than using EdDSA, due to its relatively expensive operations in the latter. SP time to attest using EdDSA is 282 ms while using HMAC is 1.2 ms (when there is one UP running on the device). As the number of UP-s increase, the time taken for SP also increases. This is due to frequent kernel context switching, as seL4 (fully verified implementation) uses only one core.

$\mathcal{P}$ARseL **TCB size**

$\mathcal{P}$ARseL TCB contains 3.9K lines of C code, including 0.6K lines for RP + PSMT (excluding the seL4 user-space libraries), and 3.3K lines for SP. Out of 3.3K lines of SP, 3.2K lines are verified, including 3K lines from HACL* EdDSA and 0.2K lines from SP run-time attestation function. $\mathcal{P}$ARseL TCB compiled binary has 1.5 MB.

2.6 $\mathcal{P}$ARseL Security Analysis

To argue $\mathcal{P}$ARseL security with respect to the adversary model in Section 3.5.2, we start by formulating $\mathcal{P}$ARseL security goal.

Security Definition: *Let $\mathcal{B}$ be an arbitrary software binary selected by Vrf. In the context of a static root of trust for measurement of user-level processes, an $\mathcal{R}$A scheme is considered secure if and only if Vrf is able to use the $\mathcal{R}$A scheme to establish a secure channel with program*

$\mathcal{P}$, where:

** $\mathcal{P}$ is an isolated user-level process running on the correct Prv;*

** At boot time, $\mathcal{P}$ was loaded with the Vrf-selected binary $\mathcal{B}$;*

Security Argument: Assuming that Vrf uses pk, included in σ (recall Equation 2.1), to establish the secure channel, $\mathcal{A}\mathsf{dv}$ can attempt to circumvent $\mathcal{P}\mathsf{ARseL}$ security by:

(1) **Loading the Right Software on the Wrong Device.** $\mathcal{A}\mathsf{dv}$ can load process $P_{\mathcal{A}\mathsf{dv}}$ with the expected binary $\mathcal{B}$ on a different device ($Prv_{\mathcal{A}\mathsf{dv}}$), also equipped with an instance of $\mathcal{P}\mathsf{ARseL}$. Then, $\mathcal{A}\mathsf{dv}$ forwards Vrf's request (intended to the original Prv) to $Prv_{\mathcal{A}\mathsf{dv}}$. $Prv_{\mathcal{A}\mathsf{dv}}$ inadvertently issues a $\mathcal{P}\mathsf{ARseL}$ attestation response that matches software $\mathcal{B}$ (loaded on $P_{\mathcal{A}\mathsf{dv}}$). However, as the secret key $\mathcal{K}$ is unique to each Prv, Vrf would not accept the received σ, thereby refusing to establish the secure channel.

(2) **Loading the Wrong Software on the Right Device.** $\mathcal{A}\mathsf{dv}$ can load a user-space process on the correct Prv but with an incorrect/malicious binary $\mathcal{B}_{\mathcal{A}\mathsf{dv}}$. This can be accomplished with physical access to Prv or by exploiting a vulnerability on a user-space process to perform persistent code injection, re-booting Prv thereafter. In either case, σ would be signed with the expected secret key $\mathcal{K}$. However, $\mathtt{mmap}$ would be updated at boot to reflect $\mathcal{B}_{\mathcal{A}\mathsf{dv}}$, i.e., the hash result $\mathsf{m}_{\mathsf{UP}_{\mathcal{A}\mathsf{dv}}}$. Consequently, Vrf would refuse to establish a secure channel with a process on Prv loaded with $\mathcal{B}_{\mathcal{A}\mathsf{dv}} \neq \mathcal{B}$.

(3) **Loading the Wrong Software on the Wrong Device.** It follows from both arguments above that this option is infeasible to $\mathcal{A}\mathsf{dv}$ due to the mismatches on both secret key $\mathcal{K}$ and measurement $\mathsf{m}_{\mathsf{UP}_{\mathcal{A}\mathsf{dv}}}$.

Therefore, $\mathcal{P}\mathsf{ARseL}$ satisfies the security definition above. $\square$

This argument assumes confidentiality of $\mathcal{K}$. In $\mathcal{P}\mathsf{ARseL}$, this is supported through formal verification of SP functional correctness, secret independence, and memory safety. It also assumes that each process is appropriately measured at boot. In $\mathcal{P}$ARseL, this is implemented by PSMT when computing mmap. The association of pk with the correct $\mathsf{m_{UP}}$ is guaranteed by seL4 badge assignments. Finally, the scheme relies on inter-process isolation for SP and any attested process $\mathcal{P}$, once the secure channel is established. The latter is inherited from seL4 provable isolation.

2.7 Discussion

Limitations: Only $\mathcal{P}$ARseL runtime TCB is verified. The integrity of $\mathcal{P}$ARseL boot time TCB is ensured via secure boot, while the correct implementation of secure boot/boot TCB are assumed. Furthermore, $\mathcal{P}$ARseL measures processes at boot time. Thus, RP configures a write-xor-execution memory permission to prevent a user process from modifying its own code. By default, although seL4 guarantees strong inter-process isolation, it gives each process full control of its own code/data segments. Due to this write-restriction, $\mathcal{P}$ARseL does not support run-time updates to user-level processes. Currently, benign updates must be done physically and require rebooting the device (in order to measure the updated program on boot). However, we believe that any software update framework compatible with seL4 (e.g. [39]) can be used to alongside $\mathcal{P}$ARseL for remote updates. The only requirement then would be to reboot the device after the update, so that $\mathcal{P}$ARseL re-measures all UP-s including the new updated UP.

(Unexpected) Termination of UP does not cause any issues because no other user process can transfer the signature (from Vrf) on behalf of another process to SP. In Vrf's view, no response will arrive (in a certain amount of time) so it can deduce that UP or Prv are no longer running. This is similar to any $\mathcal{R}$A protocol.

SP Stack Erasure is obviated in $\mathcal{P}$ARseL because SP is never terminated at run-time and seL4's inter-process isolation guarantees that only SP has access to its own stack.

2.8 Related Work

$\mathcal{R}$A*:* techniques can be classified into SW-based, HW-based, and hybrid (HW/SW co-design) architectures. Although SW-based methods such as [190, 274, 208, 152] require minimal overall costs, they rely on strong assumptions about precise time-based checksum, which is mostly unsuitable for the IoT ecosystem with the multi-hop network. HW-based methods [238, 290, 198], on the other hand, rely on some additional hardware support for $\mathcal{R}$A, e.g., some dedicated hardware components [290], or extension of existing instruction sets [180], which introduce cost and other barriers, especially for low-end and mid-range devices. Hybrid approach [108, 195, 67, 114] is considered to be more suitable for IoT ecosystems because it aims for minimal hardware changes while keeping the same security levels as HW-based $\mathcal{R}$A. Using the hybrid $\mathcal{R}$A as a building block, many security services have been also suggested, such as proof of execution [110, 120], control-flow and data-flow attestation [26, 121, 115, 116, 304, 284, 149], and secure software updates [39, 109, 113]. Since $\mathcal{P}$ARseL also provides a hybrid $\mathcal{R}$A, it can be also used for such security services. Several recent papers on hybrid $\mathcal{R}$A/$\mathcal{R}$A-based security services [110, 108, 109, 113, 114] provide formal verification of their suggested architectures/implementations. They use model checking with temporal logic to verify their implementations while they use theorem proving to show that their proved properties are sufficient for their security goal(s).

Verfied security applications in F^*: [9] lists all the papers that apply F^* in security and cryptography, including HACL* [312]. DICE* [287] is a notable paper related to $\mathcal{P}$ARseL, which proposes a verified implementation of *Device Identifier Composition Engine* (DICE), an industry-standard *measured boot* protocol, for low-cost IoT devices. Similar to $\mathcal{P}$ARseL, it has layered architecture with static components whose implementations are verified over

*Low**. The main difference is how to guarantee the $\mathcal{K}$ confidentiality. DICE enforces the access control to the master secret key by locating it in a read-only and latchable memory so that only a hardware reset can disable/restore access to it. The first hardware layer (called DICE engine) only has access to the secret, and it authenticates the next layer (L0) and derives the secret for L0 from its master secret and L0 measurement. This ensures the same derived secret only when L0 firmware is not compromised. Once received control, L0 uses this secret to derive a unique key pair from its secret and the next-layer firmware (L1) for L1 attestation and secure key exchange. Although $\mathcal{P}$ARseL assumes a secure boot for correct seL4 deployment, both $\mathcal{P}$ARseL and DICE* present verified implementations for the static root of trust for embedded devices, with different ways of guaranteeing the access control.

Architectures/applications over seL4: After being released in 2009 [194], seL4 has been actively implanted and used in both academia and industries in various domains, including automotive [21], aviation [92], and medical devices [263]. Apart from massive research from the Trustworthy Systems group in UNSW Sydney, many projects such as [135, 248] leverage their architecture atop seL4.

2.9 Summary

This chapter presented $\mathcal{P}$ARseL, a provable attestation root-of-trust over seL4 for mid-range IoT/embedded devices. We implemented $\mathcal{P}$ARseL on SabreLite and demonstrated its overall feasibility and practicality. In addition, we formally verified the runtime component of $\mathcal{P}$ARseL with respect to functional correctness, memory safety, and secret independence, using the *Low** tool-chain. Our source code including verification proofs is available at [33].

Chapter 3

Privacy-from-Birth: Protecting Sensed Data from Malicious Sensors with VERSA

Now, this chapter moves on to the low-end devices that usually run a single core and single process over the bare metal. Due to the lack of security features, such as secure boot, MMU, and MPU, we claim that the privacy of data generated from such devices must be ensured "from birth" so that the data can be protected even if the device software is compromised. We formalize this notion and propose a secure architecture realizing it with formal verification.

3.1 Introduction

As mentioned in Section 2.1, increasing number of IoT/embedded devices become pervasive in many aspects of everyday life, and they also represent increasingly attractive attack targets for exploits and malware. In particular, low-end (cheap, small, and simple) micro-controller units (MCUs) are designed with strict cost, size, and energy limitations, and therefore, it is hard to offer any concrete guarantees for tasks performed by these MCUs. This is due to their lack of sophisticated security and privacy features, compared to higher-end computing devices, such as smartphones or general-purpose IoT controllers, e.g., Amazon Echo or Google Nest. As MCUs increasingly permeate private spaces, exploits that abuse their sensing capabilities to obtain sensitive data represent a significant privacy threat.

Over the past decade, the IoT privacy issues have been recognized and explored by the research community [297, 291, 210, 310, 257]. Many techniques (e.g., [236, 203]) were developed to secure sensor data from active attacks that impersonate users, IoT back-ends, or servers. Another research direction focused on protecting private data from passive in-network observers that intercept traffic [288, 34, 35, 36] or perform traffic analysis based on unprotected packet headers and other metadata, e.g., sizes, timings, and frequencies. However, security of sensor data **on the device** which originates that data has not been investigated. We consider this to be a crucial issue, since all software on the device can be compromised and leak (exfiltrate) sensed data. Whereas, aforementioned techniques assume that sensing device runs the expected **benign** software.

We claim that in order to solve this problem, privacy of sensed data must be ensured "from birth". This corresponds to two requirements: (1) access to sensing interfaces must be strictly controlled, such that only authorized code is allowed to read data, (2) sensed data must be protected as soon as it is converted to digital form. Even the simplest devices (e.g., motion sensors, thermostats, and smart plugs) should be protected since prior work [86, 234, 283, 186] amply demonstrates that private – and even safety-critical – information can be inferred from sensed data. It is also well-known that even simple low-end IoT devices are subject to malware attacks. This prompts a natural question: *Can privacy of sensed data be guaranteed if the device software is compromised?* We refer to this guarantee as *Privacy-from-Birth* (*PfB*).

Some previous results considered potential software compromise in low-end devices and proposed methods to enable security services, such as remote verification of device software state (remote attestation) [252, 238, 108, 32, 67, 195], proofs of remote software execution [110], control- & data-flow attestation [120, 26, 121, 304, 284, 116, 115], as well as proofs of remote software updates, memory erasure, and system reset [109, 31, 39].

Regardless of their specifics, such techniques only detect of violations or compromises **after the fact**. In the context of *PfB*, that is too late since leakage of private sensed data likely already occurred. Notably, SANCUS [238] specifically discusses the problem of access control to sensor peripherals (e.g., GPIO) and proposes attestation of software accessing (or controlling access to) these peripherals. However, this only allows detection of compromised peripheral-accessing software and does not prevent illegal peripheral access.

To bridge this gap and obtain *PfB*, we construct the Verified Remote Sensing Authorization (VERSA) architecture. It provably prevents leakage of private sensor data even when the underlying device is software-compromised. At a high level, VERSA combines three key features: (1) *Mandatory Sensing Operation Authorization*, (2) *Atomic Sensing Operation Execution*, and (3) *Data Erasure on Boot* (see Section 3.3). To attain these features, VERSA implements a minimal and formally verified hardware monitor that runs independently from (and in parallel with) the main CPU, without modifying the CPU core. We show that VERSA is an efficient and inexpensive means of guaranteeing *PfB*.

This work makes the following contributions:

- Formulates *PfB* with a high-level specification of requirements, followed by a game-based formal definition of the *PfB* goal.
- Constructs VERSA, an architecture that guarantees *PfB*.
- Implements and deploys VERSA on a commodity low-end MCU, which demonstrates its cost-effectiveness and practicality.
- Formally verifies VERSA implementation and proves security of the overall construction, hence obtaining provable security at both architectural and implementation levels. VERSA implementation and its computer proofs are publicly available in [25].

3.2 Preliminaries

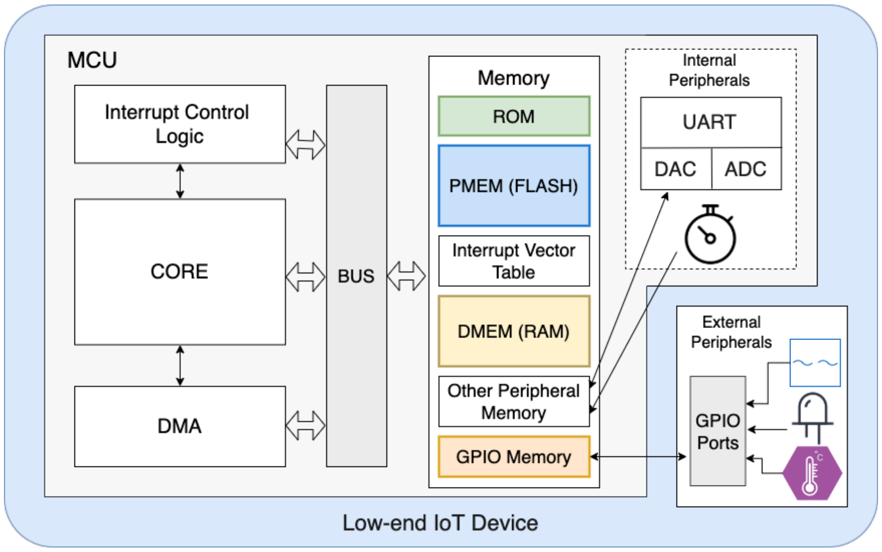


Figure 3.1: System Architecture of an MCU-based IoT Device

3.2.1 Scope & MCU-based devices

This work focuses on low-end CPS/IoT/smart devices with low computing power and meager resources. These are some of the smallest and weakest devices based on low-power single-core MCUs with only a few kilobytes (KB) of program and data memory. Two prominent examples are Atmel AVR ATmega and TI MSP430: 8- and 16-bit CPUs, respectively, typically running at 1-16MHz clock frequencies, with ≈ 64KB of addressable memory.

Figure 3.1 illustrates a generic architecture representing such MCUs. The CPU core and the Direct-Memory Access (DMA) controller access memory through a bus.[1] Memory can be divided into 5 logical regions: (1) Read-only memory (ROM), if present, stores critical software such as a bootloader, burnt into the device at manufacture time and not modifiable thereafter; (2) program memory (PMEM), usually realized as flash, is non-volatile and stores program instructions; (3) interrupt vector table (also in flash and often considered as

[1]DMA is a hardware controller that can read/write to memory in parallel with the CPU.

part of PMEM), stores interrupt configurations; (4) data memory (DMEM), usually implemented with DRAM, is volatile and used to store program execution state, i.e., its stack and heap; and, (5) peripheral memory region (also in DRAM and often considered as a part of DMEM), contains memory-mapped I/O interfaces, i.e., addresses in the memory layout that are mapped to hardware components, e.g., timers, UART, and GPIO. In particular, GPIO are peripheral memory addresses hardwired to physical ports that interface with external circuits, e.g., analog sensors/circuits.

We note that small MCUs usually come in one of two memory architectures: Harvard and von Neumann. The former isolates PMEM and DMEM by maintaining two different buses and address spaces, while the latter keeps both PMEM and DMEM in the same address space and accessible via a single bus.

Low-end MCUs execute instructions in place, i.e., directly from flash memory. They have neither memory management units (MMUs) to support virtualization/isolation, nor memory protection units (MPUs). Therefore, privilege levels and isolation used in higher-end devices and generic enclaved execution systems (e.g., Intel SGX [180] or MIT SANCTUM [93]) are not applicable.

We believe that a *PfB*-agile architecture that is sufficiently inexpensive and efficient for such low-end devices can be later adapted to more powerful devices. Whereas, going in the opposite direction is more challenging. Furthermore, simpler devices are easier to model and reason about formally. Thus, we believe that they represent a natural starting point for the design and verification of a *PfB*-agile architecture. To this end, our prototype implementation of VERSA is integrated with MSP430, due in part to public availability of an open-source MSP430 hardware design from OpenCores [151].[2]

[2]Nevertheless, the generic machine model and methodology of VERSA are applicable to other low-end MCUs of the same class, e.g., Atmel AVR ATmega.

3.2.2 `GPIO` & MCU Sensing

A GPIO port is a set of GPIO pins arranged and controlled together, as a group. The MCU-addressable memory for a GPIO port is physically mapped (hard-wired) to physical ports that can be connected to a variety of external circuits, such as analog sensors and actuators, as shown in Figure 3.1. Each GPIO pin can be set to function as either an input or output, hence called "general purpose". Input signals produced by external circuits can be obtained by the MCU software by reading from GPIO-mapped memory. Similarly, egress electric signals (high or low voltage) can be generated by the MCU software by writing (logical 1 or 0) to GPIO-mapped memory.

Remark: “GPIO-mapped memory” includes the set of all software-readable memory regions connected to external sensors. In some cases, this set may even include multiple physical memory regions for a single physical pin. For instance, if a given GPIO pin is also equipped with an Analog-to-Digital Converter (ADC), a GPIO input could be reflected on different memory regions depending on whether the ADC is active or inactive. All such regions are considered “GPIO-mapped memory” and we refer to it simply as `GPIO`. Using this definition, in order to access sensor data, software running on the MCU must read from `GPIO`.

We also note that various applications require different sensor regimes [19]: event-driven, periodic, and on-demand. Event-driven sensors report sensed data when a trigger event occurs, while periodic sensors report sensor data at fixed time intervals. On-demand (or query-driven) sensors report sensor data whenever requested by an external entity. Although we initially consider on-demand sensing, as discussed in Section 3.3, the proposed design is applicable to other regimes.

3.2.3 *VRASED*

VRASED [108] is a verified hybrid (hardware/software) $\mathcal{R}\mathsf{A}$ architecture for for low-end MCUs. It comprises a set of (individually) verified hardware and software sub-modules; their composition provably satisfies formal definitions of $\mathcal{R}\mathsf{A}$ soundness and security. *VRASED* software component implements the authenticated integrity function computed over a given "Attested Region" (AR) of *Prv*'s memory. *VRASED* hardware component assures that its software counterpart executes securely and that no function of the secret key is ever leaked. In short, $\mathcal{R}\mathsf{A}$ soundness states that the integrity measurement must accurately reflect a snapshot of *Prv*'s memory in AR, disallowing any modifications to AR during the actual measurement. $\mathcal{R}\mathsf{A}$ security defines that the measurement must be unforgeable, implying protection of secret key $\mathcal{K}$ used for the measurement.

In order to prevent DoS attacks on *Prv*, the RA protocol may involve authentication of the attestation request, before *Prv* performs attestation. If this feature is used, an authentication token must accompany every attestation request.[3] For example, in *VRASED*, Vrf computes this token as an HMAC over $\mathcal{C}\mathsf{hal}$, using $\mathcal{K}$. Since $\mathcal{K}$ is only known to *Prv* and Vrf, this token is unforgeable. To prevent replays, $\mathcal{C}\mathsf{hal}$ is a monotonically increasing counter, and the latest $\mathcal{C}\mathsf{hal}$ used to successfully authenticate Vrf is stored by *Prv* in persistent and protected memory. In each attestation request, incoming $\mathcal{C}\mathsf{hal}$ must be greater that the stored value. Once an attestation request is successfully authenticated, the stored value is updated accordingly.

VRASED software component is stored in ROM and realized with a formally verified HMAC implementation from the HACL* cryptographic library [312], which is used to compute: $H = HMAC(KDF(\mathcal{K}, \mathcal{C}\mathsf{hal}), AR)$, where $KDF(\mathcal{K}, \mathcal{C}\mathsf{hal})$ is a one-time key derived from the received $\mathcal{C}\mathsf{hal}$ and $\mathcal{K}$ using a key derivation function.

[3] By saying "this feature is used", we mean that its usage (or lack thereof) is fixed at the granularity of a Vrf-*Prv* setting, and not per single $\mathcal{R}\mathsf{A}$ instance.

As discussed later in Section 3.6, in VERSA, *VRASED* is used as a means of authorizing a binary to access GPIO.

3.2.4 LTL, Model Checking, & Verification

Our verification and proof methodologies are in-line with prior work on the design and verification of security architectures proving code integrity and execution properties for the same class of MCUs [108, 110, 114, 29]. However, to the best of our knowledge, no prior work tackled formal models and definitions, or designed services, for guaranteed sensed data privacy. This section overviews our verification and proof methodologies that allow us to later show that VERSA achieves required *PfB* properties and end-goals.

Computer-aided formal verification typically involves three steps. First, the system of interest (e.g., hardware, software, or communication protocol) is described using a formal model, e.g., a Finite State Machine (FSM). Second, properties that the model should satisfy are formally specified. Third, the system model is checked against formally specified properties to guarantee that the system retains them. This can be done via Theorem Proving [216] or Model Checking [90]. We use the latter to verify the implementation of system sub-modules, and the former to prove new properties derived from the combination (conjunction) of machine model axioms and sub-properties that were proved for the implementation of individual sub-modules.

In one instantiation of model checking, properties are specified as *formulae* using Linear Temporal Logic (LTL) and system models are represented as FSMs. Hence, a system is represented by a triple: (σ, σ_0, T), where σ is the finite set of states, $\sigma_0 \subseteq \sigma$ is the set of possible initial states, and $T \subseteq \sigma \times \sigma$ is the transition relation set, which describes the set of states that can be reached in a single step from each state. Such usage of LTL allows representing a system behavior over time.

Our verification strategy benefits from the popular model checker NuSMV [89], which can verify generic hardware or software models. For digital hardware described at Register Transfer Level (RTL) – which is the case in this work – conversion from Hardware Description Language (HDL) to NuSMV models is simple. Furthermore, it can be automated [182] as the standard RTL design already relies on describing hardware as FSMs. LTL specifications are particularly useful for verifying sequential systems. In addition to propositional connectives, such as conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$), and implication ($\rightarrow$), LTL extends propositional logic with **temporal quantifiers**, thus enabling sequential reasoning. In this paper, we are interested in the following LTL quantifiers:

- $\mathbf{X}\phi$ – neXt ϕ: holds if ϕ is true at the next system state.
- $\mathbf{G}\phi$ – Globally ϕ: holds if for all future states ϕ is true.
- $\phi\ \mathbf{U}\ \psi$ – ϕ Until ψ: holds if there is a future state where ψ holds and ϕ holds for all states prior to that.
- $\phi\ \mathbf{W}\ \psi$ – ϕ Weak until ψ: holds if, assuming a future state where ψ holds, ϕ holds for all states prior to that. If ψ never becomes true, ϕ must hold forever. Or, more formally: $\phi\ \mathbf{W}\ \psi \equiv (\phi\ \mathbf{U}\ \psi) \vee\ \mathbf{G}(\phi)$.
- $\phi\ \mathbf{B}\ \psi$ – ϕ Before ψ: holds if the existence of state where ψ holds implies the existence of at least one earlier state where ϕ holds. Equivalently: $\phi\ \mathbf{B}\ \psi \equiv \neg(\neg\phi\ \mathbf{U}\ \psi)$.

NuSMV works by exhaustively enumerating all possible states of a given system FSM and by checking each state against LTL specifications. If any desired specification is found not to hold for specific states (or transitions between states), the model checker provides a trace that leads to the erroneous state, which helps correct the implementation accordingly. As a consequence of exhaustive enumeration, proofs for complex systems that involve complex properties often do not scale due to the so-called "state explosion" problem. To cope with it, our verification approach is to specify smaller LTL sub-properties separately and verify each respective hardware sub-module for compliance. In this process, our verification pipeline automatically converts digital hardware, described at RTL using Verilog, to Symbolic Model Verifier (SMV) [227] FSMs using Verilog2SMV [182]. The SMV representation is then fed to NuSMV for verification. Then, the composition of LTL sub-properties (verified in the model-checking phase) is proven to achieve a desired end-to-end implementation goal, also specified in LTL. This step uses an LTL theorem prover [131].

In our case, we show that the end-to-end goal of VERSA, in composition with *VRASED*, is sufficient to achieve *PfB* via cryptographic reduction from the formal security definition of *VRASED*. These steps are discussed in detail in Section 3.7.

3.3 VERSA Overview

VERSA involves two entities: a trusted remote controller ($\mathcal{C}$trl) and a device ($\mathcal{D}$ev). We expect $\mathcal{C}$trl to be a relatively powerful computing entity, e.g., a home gateway, a backend server or even a smartphone. VERSA protects sensed data on $\mathcal{D}$ev by keeping it (and any function thereof) confidential. This implies: (1) controlling GPIO access by blocking attempted reads by unauthorized software, and (2) keeping execution traces (i.e., data allocated by GPIO-authorized software) confidential. Therefore, access to GPIO is barred by default. GPIO is unlocked only for benign binaries that are pre-authorized by $\mathcal{C}$trl. Whenever a binary is deemed to be authorized on $\mathcal{D}$ev, VERSA creates for it an ephemeral isolated execution environment and permits its one-time execution. This isolated environment lasts until execution ends, which corresponds to reaching the legal exit point of the authorized binary. Therefore, by including a clean-up routine immediately before the legal exit, we can assure that all execution traces, including all sensitive information, are erased. Any attempt to interrupt, or tamper with, isolated execution causes an immediate system-wide reset, which erases all data traces.

We use the term "Sensing Operation", denoted by $\mathcal{S}$, to refer to a self-contained and logically independent binary (e.g., a function) that is responsible for processing data obtained through one or more reads from GPIO.

VERSA achieves *PfB* via three key features:

[A] *Mandatory Sensing Operation Authorization* requires explicit authorization issued by $\mathcal{C}$trl before any $\mathcal{D}$ev software reads from `GPIO`. Recall that access to `GPIO` is blocked by default. Each authorization token (ATok) coming from $\mathcal{C}$trl allows one execution of a specific sensing operation $\mathcal{S}$, although a single execution of $\mathcal{S}$ can implement several `GPIO` reads. ATok has the following properties:

1. It can be authenticated by $\mathcal{D}$ev as having been issued by $\mathcal{C}$trl; this includes freshness;
2. It grants privileges **only to a specific** $\mathcal{S}$ to access `GPIO` during its execution; and
3. It can only be used once.

$\mathcal{C}$trl can authorize multiple executions of $\mathcal{S}$ by issuing a batch of tokens, i.e., $\mathsf{ATok}_1, ..., \mathsf{ATok}_n$, for up to n executions of $\mathcal{S}$. Although supporting multiple tokens is unnecessary for on-demand sensing, it might be useful for periodic or event-driven sensing regimes discussed in Section 3.2.2.

[B] *Atomic Sensing Operation Execution* ensures that, once authorized by $\mathcal{C}$trl, $\mathcal{S}$ is executed with the following requirements:

1. $\mathcal{S}$ execution starts from its legal entry point (first instruction) and runs until its legal exit point (last instruction). This assumes a single pair of entry-exit points;
2. $\mathcal{S}$ execution can not be interrupted and its intermediate results cannot be accessed by external means, e.g., via DMA controllers; and
3. An immediate MCU reset is triggered if either (1) or (2) above is violated.

[C] *Data Erasure on Reset/Boot* works with **[B]** to guarantee that, sensed data (or any function thereof) obtained during $\mathcal{S}$ execution is not leaked due to errors or violations of security properties, which cause MCU reset per item (3) above. This feature must guarantee that all values that remain in RAM after a hard reset and the subsequent boot process, are erased before any unprivileged software can run. While some architectures already provide memory erasure on boot, for those MCUs that do not do so, it can be obtained by calling

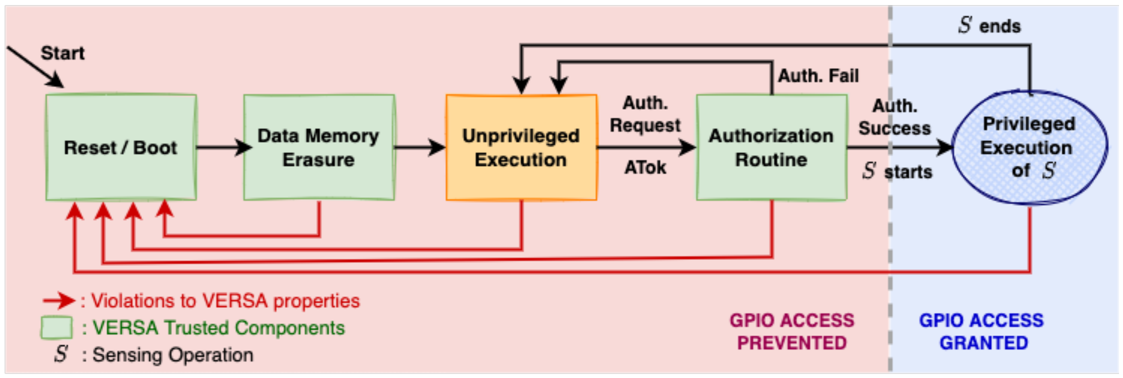


Figure 3.2: MCU execution workflow with VERSA.

a secure RAM erasure function at boot time, e.g., as a part of a ROM-resident bootloader code. We discuss this further in Section 3.9.

At a high level, correct implementation of aforementioned three features suffices to obtain *PfB*, because:

- Any compromised/modified binary can not access GPIO since it has no authorization from $\mathcal{C}$trl.
- Any authorized binary $\mathcal{S}$ must be invoked properly and run atomically, from its first, and until its last, instruction.
- Since $\mathcal{S}$ is invoked properly, intended behavior of $\mathcal{S}$ is preserved. Code reuse attacks are not possible, unless they occure as a result of bugs in $\mathcal{S}$ implementation itself. $\mathcal{C}$trl can always check for such bugs in $\mathcal{S}$ prior to authorization; see Section 3.5.2.
- $\mathcal{S}$ runs uninterrupted, meaning that it can erase all traces of its own execution from the stack before passing control to unprivileged applications. This guarantees that no sensor data remains in memory when $\mathcal{S}$ terminates.
- VERSA assures that any violation of aforementioned requirements causes an MCU reset, triggering erasure of all data memory. Therefore, malware that attempts to interrupt $\mathcal{S}$ before completion, or tamper with $\mathcal{S}$ execution integrity, will cause all data used by $\mathcal{S}$ to be erased.

Support for Output Encryption: $\mathcal{S}$ might process and use sensor data locally as part of its own execution, or generate some output that needs to be returned to $\mathcal{C}\mathsf{trl}$. In the latter case, encryption of $\mathcal{S}$ output is necessary. For this reason, VERSA supports the generation of a fresh key derived from ATok (thus implicitly shared between $\mathcal{C}\mathsf{trl}$ and $\mathcal{S}$). This key is only accessible to $\mathcal{S}$ during authorized execution. Hence, $\mathcal{S}$ can encrypt any data to be exported with this key and ensure that encrypted results can only be decrypted by $\mathcal{C}\mathsf{trl}$.

Since we assume that the encryption function is part of $\mathcal{S}$, it cannot be interrupted (or tampered with) by any unprivileged software or external means. Importantly, the encryption key is only accessible to $\mathcal{S}$ (similar to GPIO) and shielded from all other software. Furthermore, the choice of the encryption algorithm is left up to the specific $\mathcal{S}$ implementation.

Figure 3.2 illustrates MCU execution workflow discussed in this section.

3.4 MCU Machine Model

3.4.1 Execution Model

To enable formal specification of *PfB* guarantees, we formulate the MCU execution model in Definition 3.1. It represents MCU operation as a discrete sequence of MCU states, each corresponding to one clock cycle – the smallest unit of time in the system. We say that the subsequent MCU state is defined based on the current MCU state (which includes current values in memory/registers, as well as any hardware signals and effects, such as external inputs, actions by DMA controller(s), and interrupts) and the current instruction being executed by the CPU core. Similarly, the instruction to be executed in the next state is determined by the current state and the current instruction being executed.

For example, an arithmetic instruction (e.g., `add` or `mult`) causes the program counter

Definition 3.1 (MCU Execution Model).

*1 – **Execution** is modeled as a sequence of MCU states $\boldsymbol{S} := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$ and a sequence of instructions $\boldsymbol{I} := \{\mathsf{i}_0, ..., \mathsf{i}_n\}$. Since the <u>next</u> MCU state and the <u>next</u> instruction to be executed are determined by the <u>current</u> MCU state and the <u>current</u> instruction being executed, these discrete transitions are denoted as shown in the following example:*

$$(\mathsf{s}_1, \mathsf{i}_j) \leftarrow \boldsymbol{EXEC}(\mathsf{s}_0, \mathsf{i}_0); \quad (\mathsf{s}_2, \mathsf{i}_k) \leftarrow \boldsymbol{EXEC}(\mathsf{s}_1, \mathsf{i}_j); \quad \ldots \quad (\mathsf{s}_m, \perp) \leftarrow \boldsymbol{EXEC}(\mathsf{s}_{m-1}, \mathsf{i}_l)$$

The sequence $\boldsymbol{I}$ represents the physical order of instructions in memory, which is not necessarily the order of their execution. The next instruction and state are also affected by current external inputs, current data-memory values, and current hardware events, e.g., interrupts or resets, which are modeled as properties of each execution state in $\boldsymbol{S}$. The MCU always starts execution (at boot or after a reset) from state s_0 and initial instruction i_0. EXEC produces $\perp$ as the next instruction if there is no instructions left to execute.

*2 – **State Properties as Sets:** sets are used to model relevant execution properties and characterize effects/actions occurring within a given state s_t. We are particularly interested in the behaviors corresponding to the following sets:*

1. ***READ:*** *all states produced by the execution of an instruction i that reads the value from memory to a register.*
2. ***WRITE:*** *all states produced by the execution of an instruction i that writes the value from a register to memory.*
3. ***DMA^R:*** *all states produced as a result of DMA reading from memory.*
4. ***DMA^W:*** *all states s_t produced as a result of DMA writing to memory.*
5. ***IRQ:*** *all states s_t where an interrupt is triggered.*
6. ***RESET:*** *all states s_t wherein an MCU reset is triggered.*

Note that these sets are not disjoint, i.e., s_t can belong to multiple sets. Also, the aforementioned sets do not aim to model all possible MCU behaviors, but only the ones relevant to PfB. Finally, we further subdivide sets that model memory access into subsets relating to memory regions of interest. For example, considering a contiguous memory region $\mathcal{M} = [\mathcal{M}_{min}, \mathcal{M}_{max}]$, $READ_{\mathcal{M}}$ is a subset of READ containing only the states produced through EXEC of instructions that read from the memory region $\mathcal{M}$. We use the same notation to refer to other subsets, e.g., $WRITE_{\mathcal{M}}$, $DMA^R_{\mathcal{M}}$, and $DMA^W_{\mathcal{M}}$.

Definition 3.2 (Hardware Model).

$\mathcal{M}$ denotes a contiguous memory region within addresses $\mathcal{M}_{min}$ and $\mathcal{M}_{max}$ in physical memory of $\mathcal{D}\mathsf{ev}$, i.e., $\mathcal{M} := [\mathcal{M}_{min}, \mathcal{M}_{max}]$. s represents the system execution state at a given CPU cycle.

Program counter & instruction execution:

$$\mathbf{G} : \{[\mathbf{X}(\mathsf{s}) \leftarrow EXEC(\mathsf{s}, \mathsf{i}_k) \wedge \mathsf{i}_k \in \mathcal{M})] \rightarrow (PC \in \mathcal{M})\} \tag{3.1}$$

Memory Reads/Writes:

$$\mathbf{G} : \{\mathbf{X}(\mathsf{s}) \in READ_{\mathcal{M}} \rightarrow (R_{en} \wedge D_{addr} \in \mathcal{M})\} \tag{3.2}$$

$$\mathbf{G} : \{\mathbf{X}(\mathsf{s}) \in WRITE_{\mathcal{M}} \rightarrow (W_{en} \wedge D_{addr} \in \mathcal{M})\} \tag{3.3}$$

$$\mathbf{G} : \{(\mathbf{X}(\mathsf{s}) \in DMA^R_{\mathcal{M}} \vee \mathbf{X}(\mathsf{s}) \in DMA^W_{\mathcal{M}}) \rightarrow (DMA_{en} \wedge DMA_{addr} \in \mathcal{M})\} \tag{3.4}$$

Interrupts (irq) and Resets:

$$\mathbf{G} : \{\mathsf{s} \in IRQ \leftrightarrow irq\} \tag{3.5}$$

$$\mathbf{G} : \{\mathsf{s} \in RESET \leftrightarrow reset\} \tag{3.6}$$

(PC) to point to the subsequent address in physical memory. However, an interrupt (which is a consequence of the current MCU state) may occur and deviate the normal execution flow. Alternatively, a branching instruction may be executed and cause PC to jump to some arbitrary instruction that is not necessarily located at the subsequent position in the MCU flash memory.

To reason about events during the MCU operation, we say that each MCU state can belong to one or more sets. Belonging to a given set implies that the state has a given property of interest. Definition 3.1 introduces six sets of interest, representing states in which memory is read/written by CPU or DMA, as well as states in which an interrupt or reset occurs.

3.4.2 Hardware Signals

We now formalize the effects of execution, modeled in Definition 3.1, to the values of concrete hardware signals that can be monitored by VERSA hardware in order to attain *PfB* guarantees. Informally, we model the following simple axioms:

[A1] PC: contains the memory address containing the instruction being executed at a given cycle.

[A2] CPU Memory Access: Whenever memory is read or written, a data-address signal (D_{addr}) contains the address of the corresponding memory location. A data read-enable bit (R_{en}) must be set for a read access and a data write-enable bit (W_{en}) must be set for a write access.

[A3] DMA: Whenever a DMA controller attempts to access the main memory, a DMA-address signal (DMA_{addr}) contains the address of the accessed memory location and a DMA-enable bit (DMA_{en}) must be set.

[A4] Interrupts: When hardware interrupts or software interrupts happen, the irq signal is set.

[A5] MCU reset: At the end of a successful reset routine, all registers (including PC) are set to zero before restarting software execution. The reset handling routine cannot be modified, as resets are handled by MCU in hardware. When a reset happens, the corresponding $reset$ signal is set. The same signal is also set when the MCU initializes for the first time.

This model strictly adheres to MCU specifications, assumed to be correctly implemented by the underlying MCU core.

Definition 3.2 presents formal specifications for aforementioned axioms in LTL. Instead of explicitly quantifying time, LTL embeds time within the logic by using temporal quantifiers (see Section 3.2). Hence, rather than referring to execution states using temporal variables (i.e., state t, state $t+1$, state $t+2$), a single variable (s) and LTL quantifiers suffice to specify, e.g., "current", "next", "future" system states (s). For this part of the model, we are mostly interested in: (1) describing MCU state at the next CPU cycle ($\mathbf{X}(\mathsf{s})$) as a function of the MCU state at the current CPU cycle (s), and (2) describing which particular MCU signals must be triggered in order for $\mathbf{X}(\mathsf{s})$ to be in each of the sets defined in Definition 3.1.

LTL statements in Definition 3.2 formally model axioms **[A1]-[A5]**, i.e., the subset of MCU behavior that is relevant to, and sufficient for formally verifying, VERSA. LTL (3.1) models **[A1]**, (3.2) and (3.3) model **[A2]**, and each (3.4), (3.5), and (3.6) models **[A3]**, **[A4]**, and **[A5]**, respectively.

3.5 *PfB* Definitions

Based on the specified machine model, we now proceed with the formal definition of *PfB*.

DEFINITION 3.3 (Syntax: *PfB* scheme).

A Privacy-from-Birth (PfB) scheme is a tuple of algorithms [Authorize, Verify, XSensing]*:*

1. $\mathsf{Authorize}^{\mathcal{C}\mathsf{trl}}(\mathcal{S},\cdots)$*: an algorithm executed by* $\mathcal{C}\mathsf{trl}$ *taking as* <u>*input*</u> *at least one executable* $\mathcal{S}$ *and* <u>*producing*</u> *at least one authorization token* $\boldsymbol{ATok}$ *which can be sent to* $\mathcal{D}\mathsf{ev}$ *to authorize one execution of* $\mathcal{S}$ *with access to* $\boldsymbol{GPIO}$*.*
2. $\mathsf{Verify}^{\mathcal{D}\mathsf{ev}}(\mathcal{S},\boldsymbol{ATok},\cdots)$*: an algorithm (with possible hardware-support), executed by* $\mathcal{D}\mathsf{ev}$*, that takes as* <u>*input*</u> $\mathcal{S}$ *and* $\boldsymbol{ATok}$*. It uses* $\boldsymbol{ATok}$ *to check whether* $\mathcal{S}$ *is pre-authorized by* $\mathcal{C}\mathsf{trl}$ *and outputs* $\top$ *if verification succeeds, and* $\bot$ *otherwise.*
3. $\mathsf{XSensing}^{\mathcal{D}\mathsf{ev}}(\mathcal{S},\cdots)$*: an algorithm (with possible hardware-support) that executes* $\mathcal{S}$ *in* $\mathcal{D}\mathsf{ev}$*, producing a sequence of states* $E := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$*. It returns* $\top$*, if sensing successfully occurs during* $\mathcal{S}$ *execution, i.e.,* $\exists \mathsf{s} \in E$ *such that* $(\mathsf{s} \in READ_{\boldsymbol{GPIO}}) \wedge (\mathsf{s} \notin RESET)$*); it returns* $\bot$*, otherwise.*

Remark: *In the parameter list,* $(\cdots)$ *means that additional/optional parameters might be included depending on the specific PfB construction.*

DEFINITION 3.4 (*PfB* Game-based Definition).

3.4.1 Auxiliary Notation & Predicate(s):

- *Let* $\mathcal{K}$ *be a secret string of bit-size* $|\mathcal{K}|$*;* λ *be the sec. param., determined by* $|\mathcal{K}|$*, i.e.,* $\lambda = \Theta(|\mathcal{K}|)$*;*
- *Let* atomicExec *be a predicate evaluated on some sequence of states* $\boldsymbol{S}$ *and some software – i.e., some sequence of instructions* $\boldsymbol{I}$*.*
  - $\mathsf{atomicExec}(\boldsymbol{S} := \{\mathsf{s}_1, ..., \mathsf{s}_m\}, \boldsymbol{I} := \{\mathsf{i}_0, ..., \mathsf{i}_n\}) \equiv \top$ *iff the following hold; otherwise,* $\bot$*.*
    1. ***Legal Entry Instruction:*** *The first execution state* s_1 *in* $\boldsymbol{S}$ *is produced by the execution of the first instruction* i_0 *in* $\boldsymbol{I}$*.*
       i.e., $(\mathsf{s}_1 \leftarrow EXEC(\mathsf{i}_0, \mathsf{s}*)) \;\vee\; (\mathsf{s}_1 \in RESET)$*, where* $\mathsf{s}*$ *is any state prior to* s_1*.*
    2. ***Legal Exit Instruction:*** *The last execution state* s_m *in* $\boldsymbol{S}$ *is produced by the execution of the last instruction* i_n *in* $\boldsymbol{I}$*.*
       i.e., $(\mathsf{s}_m \leftarrow EXEC(\mathsf{i}_n, \mathsf{s}_{m-1})) \;\vee\; (\mathsf{s}_m \in RESET)$*.*
    3. ***Self-Contained Execution:*** *For all* s_j *in* $\boldsymbol{S}$*,* s_j *is produced by the execution of an instruction* i_k *in* $\boldsymbol{I}$*, for some* k*.*
       i.e., $(\mathsf{s}_j \leftarrow EXEC(\mathsf{i}_k, \mathsf{s}_{j-1})) \;\vee\; (\mathsf{s}_j \in RESET)$*, for some* $\mathsf{i}_k \in \boldsymbol{I}$*.*
    4. ***No Interrupts, No DMA:*** *For all* s_j *in* $\boldsymbol{S}$*,* s_j *is neither in the IRQ or DMA. i.e.,* $[(\mathsf{s}_j \notin IRQ) \wedge (\mathsf{s}_j \notin DMA)] \;\vee\; (\mathsf{s}_j \in RESET)$*.*

3.4.2 PfB-Game: *The challenger plays the following game with* $\mathcal{A}\mathsf{dv}$*:*

1. $\mathcal{A}\mathsf{dv}$ *is given full control over* $\mathcal{D}\mathsf{ev}$ *software state, implying* $\mathcal{A}\mathsf{dv}$ *can execute any (polynomially sized) sequence of arbitrary instructions* $\{\mathsf{i}_0^{\mathsf{Adv}}, ..., \mathsf{i}_n^{\mathsf{Adv}}\}$*, inducing the associated changes in* $\mathcal{D}\mathsf{ev}$*'s sequence of execution states;*
2. $\mathcal{A}\mathsf{dv}$ *has oracle access to polynomially many calls to* Verify*.* $\mathcal{A}\mathsf{dv}$ *also has access to the set of software executables,* $\mathsf{SW} := \{S_1, ..., S_l\}$*, and the set of all corresponding authorization "tokens",* $\mathsf{T} := \{\boldsymbol{ATok}_1, ..., \boldsymbol{ATok}_l\}$*, ever produced by any prior* $\mathcal{C}\mathsf{trl}$ *calls to* Authorize *up until time* t*. i.e.,* $\boldsymbol{ATok}_j \leftarrow \mathsf{Authorize}(\mathcal{S}_j, ...)$*, for all* j*.*
3. *Let* $\mathsf{U} \subset \mathsf{T}$ *be the set of all "used" authorization tokens up until time* t*, i.e.,* $\boldsymbol{ATok}_j \in \mathsf{U}$*, if a call to* $\mathsf{XSensing}(\mathcal{S}_j, ...)$ *returned* $\top$ *up until time* t*; Let* P *be the set of "pending" (issued but not used) authorization tokens, i.e.,* $\mathsf{P} := \mathsf{T} \setminus \mathsf{U}$*.*
4. *At any arbitrary time* t*,* $\mathcal{A}\mathsf{dv}$ *wins if it can perform an* ***unauthorized or tampered sensing execution****, i.e.:*
   – $\mathcal{A}\mathsf{dv}$ *triggers an* $\mathsf{XSensing}(\mathcal{S}_{\mathcal{A}\mathsf{dv}}, ...)$ *operation that returns* $\top$*, for* $\forall \mathcal{S}_{\mathcal{A}\mathsf{dv}} \notin \mathsf{SW}$*, or*
   – $\mathcal{A}\mathsf{dv}$ *triggers* $(\boldsymbol{S}, \top) \leftarrow \mathsf{XSensing}(\mathcal{S}_j, ...)$ *such that* $\mathsf{atomicExec}(\boldsymbol{S}, \mathcal{S}_j) \equiv \bot$*, for some* $\mathcal{S}_j \in \mathsf{SW}$ *and* $\boldsymbol{ATok}_j \in \mathsf{U}$ *.*

3.4.3 PfB-Security: *A scheme is considered PfB-Secure iff, for all PPT adversaries* $\mathcal{A}\mathsf{dv}$*, there exists a negligible function* negl *such that:*

$$Pr[\mathcal{A}\mathsf{dv}, \textit{PfB-Game}] \leq \mathsf{negl}(\mathsf{l})$$

3.5.1 *PfB* Syntax

A *PfB* scheme involves two parties: $\mathcal{C}\mathsf{trl}$ and $\mathcal{D}\mathsf{ev}$. $\mathcal{C}\mathsf{trl}$ authorizes $\mathcal{D}\mathsf{ev}$ to execute some software $\mathcal{S}$ which accesses $\mathtt{GPIO}$. It should be impossible for any software different from $\mathcal{S}$ to access $\mathtt{GPIO}$ data, or any function thereof (see Definition 3.4). $\mathcal{C}\mathsf{trl}$ is trusted to only authorize functionally correct code. The goal of a *PfB* scheme is to facilitate sensing-dependent execution while keeping all sensed data private from all other software.

Definition 3.3 specifies a syntax for *PfB* scheme composed of three functionalities: $\mathsf{Authorize}$, Verify, and $\mathsf{XSensing}$. $\mathsf{Authorize}$ is invoked by $\mathcal{C}\mathsf{trl}$ to produce an authorization token, $\mathtt{ATok}$, to be sent to $\mathcal{D}\mathsf{ev}$, enabling $\mathcal{S}$ to access $\mathtt{GPIO}$. Verify is executed at $\mathcal{D}\mathsf{ev}$ with $\mathtt{ATok}$ as input, and it checks whether $\mathtt{ATok}$ is a valid authorization for the software on $\mathcal{D}\mathsf{ev}$. If and only if this check succeeds, Verify returns $\top$. Otherwise, it returns $\perp$. The verification success indicates one execution of $\mathcal{S}$ granted on $\mathcal{D}\mathsf{ev}$ via $\mathsf{XSensing}$. $\mathsf{XSensing}$ is considered successful (returns $\top$), if there is at least one MCU state produced by $\mathsf{XSensing}$ where a $\mathtt{GPIO}$ read occurs <u>without</u> causing an MCU *reset*, i.e., $(\mathsf{s} \in READ_{GPIO}) \land \neg(\mathsf{s} \in RESET)$. Otherwise, $\mathsf{XSensing}$ returns $\perp$. That is, $\mathsf{XSensing}$ models execution of any software in the MCU and its return symbol indicates whether a $\mathtt{GPIO}$ read occurred during its execution. Therefore, invocation of $\mathsf{XSensing}$ on any input software that does not read from $\mathtt{GPIO}$ returns $\perp$. Figure 3.3 illustrates a benign *PfB* interaction between $\mathcal{C}\mathsf{trl}$ and $\mathcal{D}\mathsf{ev}$.

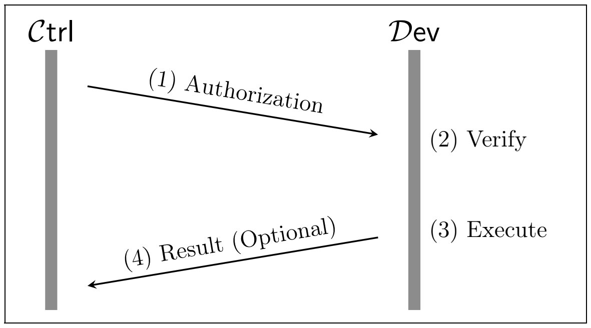


Figure 3.3: *PfB* interaction between $\mathcal{C}\mathsf{trl}$ and $\mathcal{D}\mathsf{ev}$

3.5.2 Assumptions & Adversarial Model

We consider an adversary, $\mathcal{A}\mathsf{dv}$, that controls the entire software state of $\mathcal{D}\mathsf{ev}$, including PMEM (flash) and DMEM (DRAM). It can attempt to modify any writable memory (including PMEM) or read any memory, including peripheral regions, such as `GPIO`, unless explicitly protected by verified hardware. It can launch code injection attacks to execute arbitrary instructions from PMEM or even DMEM (if the MCU architecture supports execution from DMEM). It also has full control over any DMA controllers on $\mathcal{D}\mathsf{ev}$ that can directly read/write to any part of the memory independently of the CPU. It can induce interrupts to pause any software execution and leak information from its stack, or change its control-flow. We consider Denial-of-Service (DoS) attacks, whereby $\mathcal{A}\mathsf{dv}$ abuses *PfB* functionality in order to render $\mathcal{D}\mathsf{ev}$ unavailable, to be out-of-scope. These are attacks on $\mathcal{D}\mathsf{ev}$ availability and not on sensed data privacy.

Executable Correctness: we stress that **VERSA** aims to guarantee that $\mathcal{S}$, as specified by $\mathcal{C}\mathsf{trl}$, is the only software that can access and process `GPIO` data. Similar to other trusted hardware architectures, *PfB* does not check for lack of implementation bugs within $\mathcal{S}$; thus it is not concerned with run-time (e.g., control-flow and data-only) attacks. As a relatively powerful and trusted entity $\mathcal{C}\mathsf{trl}$ can use various well-known vulnerability detection methods, e.g., fuzzing [85], static analysis [94], and even formal verification, to scrutinize $\mathcal{S}$ before authorizing it.

Physical Attacks: physical and hardware-focused attacks are considered out of scope. We assume that $\mathcal{A}\mathsf{dv}$ cannot modify code in ROM, induce hardware faults, or retrieve $\mathcal{D}\mathsf{ev}$'s secrets via side-channels that require $\mathcal{A}\mathsf{dv}$'s physical presence. Protection against such attacks can be obtained via standard physical security techniques [260]. This assumption is in line with related work on trusted hardware architectures for embedded systems [252, 108, 195, 67].

3.5.3 *PfB* Game-based Definition

Definition 3.4 starts by introducing an auxiliary predicate atomicExec. It defines whether a particular sequence of execution states (produced by the execution of some software $\mathcal{S}$) adheres to all necessary execution properties for *Atomic Sensing Operation Execution* discussed in Section 3.3.

In atomicExec (in Definition 3.4.1), conditions 1-3 guarantee that a given $\mathcal{S}$ is executed as a whole and no external instruction is executed between its first and last instructions. Condition 4 assures that DMA is inactive during execution, hence protecting intermediate variables in DMEM against DMA tampering. Additionally, malicious interrupts could be leveraged to illegally change the control-flow of $\mathcal{S}$ during its execution. Therefore, condition 4 stipulates that both cases cause atomicExec to return $\perp$.

PfB-Game in Definition 3.4.2 models $\mathcal{A}$dv's capabilities by allowing it to execute any sequence of (polynomially many) instructions. This models $\mathcal{A}$dv's full control over software executed on the MCU, as well as its ability to use software to modify memory at will. It can also call Verify any (polynomial) number of times in an attempt to gain an advantage (e.g., learn something) from Verify executions.

To win the game, $\mathcal{A}$dv must succeed in executing some software that does not cause an MCU reset, and either: (1) is unauthorized, yet reads from GPIO, or (2) is authorized, yet violates atomicExec predicate conditions during its execution.

3.6 **VERSA**: Realizing *PfB*

VERSA runs in parallel with the MCU core and monitors a set of MCU signals: PC, D_{addr}, R_{en}, W_{en}, DMA_{en}, DMA_{addr}, and irq. It also monitors ER_{min} and ER_{max}, the boundary

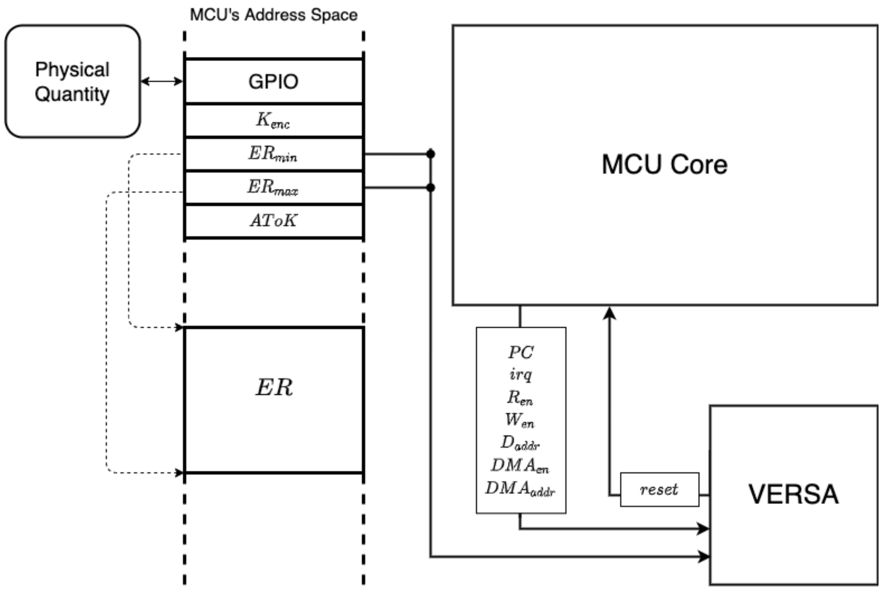


Figure 3.4: VERSA Architecture

memory addresses of ER where $\mathcal{S}$ is stored; these are collectively referred to as "*METADATA*". VERSA hardware module detects privacy violations in real-time, based on aforementioned signals and *METADATA* values, causing an immediate MCU reset. Figure 3.4 shows the VERSA architecture. For quick reference, MCU signals and memory regions relevant to VERSA are summarized in Table 3.1. To facilitate specification of VERSA properties, we introduce the following two macros:

$$Read\_Mem(i) \equiv (R_{en} \land D_{addr} = i) \lor (DMA_{en} \land DMA_{addr} = i)$$
$$Write\_Mem(i) \equiv (W_{en} \land D_{addr} = i) \lor (DMA_{en} \land DMA_{addr} = i)$$

representing read/write from/to a particular memory address i by either CPU or DMA. For reads/writes from/to some continuous memory region (composed of multiple addresses) $\mathcal{M} = [\mathcal{M}_{min}, \mathcal{M}_{max}]$, we instead say $D_{addr} \in \mathcal{M}$ to denote that $D_{addr} = i \land (i \geq \mathcal{M}_{min}) \land (i \leq \mathcal{M}_{max})$. The same holds for notation $DMA_{addr} \in \mathcal{M}$.

Construction 1. *VERSA instantiates a PfB* = [Authorize, Verify, XSensing] *scheme as follows:*
– $\mathcal{K}$ is a symmetric key pre-shared between $\mathcal{C}$trl and VRASED secure architecture in $\mathcal{D}$ev;

1. $\mathsf{Authorize}^{\mathcal{C}\mathsf{trl}}(\mathcal{S})$: *$\mathcal{C}$trl produces an authorization message $\boldsymbol{M} := (\mathcal{S}, \mathcal{C}\mathsf{hal}, \boldsymbol{ATok})$, where $\mathcal{S}$ is a software, i.e., a sequence of instructions $\{\mathsf{i}_1, ..., \mathsf{i}_n\}$, that $\mathcal{C}$trl wants to execute on $\mathcal{D}$ev; $\mathcal{C}$hal is a monotonically increasing challenge; and $\boldsymbol{ATok} := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S})$ is an authentication token computed as below. $\mathcal{C}$trl sends $\boldsymbol{M}$ to $\mathcal{D}$ev. Upon receiving $\boldsymbol{M}$, $\mathcal{D}$ev is expected to parse $\boldsymbol{M}$, find the memory region for $\mathcal{S}$, and execute* Verify *(see below).*

2. $\mathsf{Verify}^{\mathcal{D}\mathsf{ev}}(ER, \boldsymbol{ATok}, \mathcal{C}\mathsf{hal})$: *calls VRASED functionality [108] on memory region $ER := [ER_{min}, ER_{max}]$ to securely compute σ as below. If $\sigma = \boldsymbol{ATok}$, output $\top$; O.w., output $\bot$.*

$$\sigma := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), ER) \tag{3.7}$$

3. $\mathsf{XSensing}^{\mathcal{D}\mathsf{ev}}(ER)$: *starts execution of software in ER by jumping to ER_{min} (i.e., setting $PC = ER_{min}$). A benign call to* XSensing *with input ER is expected to occur after one successful computation of* Verify *for the same ER region and contents therein. Otherwise, VERSA hardware support (see below) will cause the MCU to reset when $\boldsymbol{GPIO}$ is read.* XSensing *produces $\boldsymbol{E} := \{\mathsf{s}_0, ..., \mathsf{s}_m\}$, the set of states produced by executing ER, and outputs $\top$ or $\bot$ as follows:*

$$\mathsf{XSensing}(ER) = \begin{cases} (\boldsymbol{E}, \top), & \textit{if } \exists \mathsf{s} \in \boldsymbol{E} \textit{ such that } (\mathsf{s} \in READ_{\boldsymbol{GPIO}}) \wedge (\mathsf{s} \notin RESET) \\ (\boldsymbol{E}, \bot), & \textit{otherwise} \end{cases} \tag{3.8}$$

4. HardwareMonitor*: <u>At all times</u>, VERSA verified hardware enforces all following LTL properties :*

A – Read-Access Control to *GPIO*:

$$\mathbf{G}: \{(Read\_Mem(\boldsymbol{GPIO}) \wedge \neg(PC \in ER)) \rightarrow reset\} \tag{3.9}$$

$$\mathbf{G}: \{[(PC = ER_{max}) \vee reset] \rightarrow (\neg Read\_Mem(\boldsymbol{GPIO}) \vee reset) \;\boldsymbol{W}\; (PC = \mathsf{i}_{Auth})\} \tag{3.10}$$

B – Ephemeral Immutability of ER and *METADATA*

$$\mathbf{G}: \{(PC = \mathsf{i}_{Auth}) \wedge (Write\_Mem(ER) \vee Write\_Mem(METADATA)) \rightarrow reset\} \tag{3.11}$$

$$\mathbf{G}: \{((Write\_Mem(ER) \vee Write\_Mem(METADATA) \rightarrow (\neg Read\_Mem(\boldsymbol{GPIO}) \vee reset) \;\boldsymbol{W}\; (PC = \mathsf{i}_{Auth}))\} \tag{3.12}$$

$$\text{[Optional] } \mathbf{G}: \{((Write\_Mem(ER) \vee Write\_Mem(METADATA) \rightarrow (\neg Read\_Mem(eKR) \vee reset) \;\boldsymbol{W}\; (PC = \mathsf{i}_{Auth}))\} \tag{3.13}$$

C – Atomicity and Controlled Invocation of ER:

$$\mathbf{G}: \{\neg reset \wedge (PC \in ER) \wedge \neg\mathbf{X}(PC \in ER) \rightarrow (PC = ER_{max}) \vee \mathbf{X}(reset)\} \tag{3.14}$$

$$\mathbf{G}: \{\neg reset \wedge \neg(PC \in ER) \wedge \mathbf{X}(PC \in ER) \rightarrow \mathbf{X}(PC = ER_{min}) \vee \mathbf{X}(reset)\} \tag{3.15}$$

$$\mathbf{G}: \{(PC \in ER) \wedge (irq \vee DMA_{en}) \rightarrow reset\} \tag{3.16}$$

[Optional] Read/Write-Access Control to Encryption Key ($\mathcal{K}_{enc}$) in eKR:

$$\mathbf{G}: \{(Read\_Mem(eKR) \wedge \neg(PC \in ER)) \rightarrow reset\} \tag{3.17}$$

$$\mathbf{G}: \{[(PC = ER_{max}) \vee reset] \rightarrow (\neg Read\_Mem(eKR) \vee reset) \;\boldsymbol{W}\; (PC = \mathsf{i}_{Auth})\} \tag{3.18}$$

$$\mathbf{G}: \{[Write\_Mem(eKR) \wedge \neg(PC \in VR)] \rightarrow reset\} \tag{3.19}$$

Remark: *[Optional] properties are needed only if support for encryption of outputs is desired.*

Figure 3.5: Verified Remote Sensing Authorization (VERSA) Scheme

Table 3.1: Notation used in Chapter 3

Notation	Description
PC	Current program counter value
R_{en}	1-bit signal that indicates if MCU is reading from memory
W_{en}	1-bit signal that indicates if MCU is writing to memory
D_{addr}	Memory address of an MCU memory access
DMA_{en}	1-bit signal that indicates if DMA is active
DMA_{addr}	Memory address being accessed by DMA, when active
irq	1-bit signal that indicates if an interrupt is happening
$reset$	Signal that reboots the MCU when set to logic '1'
ER	A configurable memory region where the sensing operation $\mathcal{S}$ is stored, $ER = [ER_{min}, ER_{max}]$
$METADATA$	Metadata memory region; contains ER_{min} and ER_{max}
ATok	Fixed memory region from which Verify reads the authorization token when called
GPIO	Memory region that is mapped to GPIO port
VR	Memory region storing Verify code which instantiates *VRASED* software and its hardware protection
i_{Auth}	A fixed address in ROM, only be reachable (i.e., $PC = \mathsf{i}_{Auth}$) by a successful Verify call (i.e., Verify returns $\top$)
eKR	(Optional) memory region for the encryption key $\mathcal{K}_{enc}$ necessary to encrypt the $\mathcal{S}$ output (relevant to sensed data)

3.6.1 VERSA: Construction

Recall the key features of VERSA from Section 3.3. To guarantee *Mandatory Sensing Operation Authorization* and *Atomic Sensing Operation Execution*, VERSA constructs $PfB =$ (Authorize, Verify, XSensing) algorithms as in Construction 1.

Authorize: To authorize $\mathcal{S}$, $\mathcal{C}$trl picks a monotonically increasing $\mathcal{C}$hal and generates ATok $:= HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S})$ (this follows *VRASED* authentication algorithm – see VERSA Verify specification below). ATok is computed over $\mathcal{S}$ with a one-time key derived from $\mathcal{K}$ and $\mathcal{C}$hal, where $\mathcal{K}$ is the master secret key shared between $\mathcal{C}$trl and $\mathcal{D}$ev.

Verify: To securely verify that an executable $\mathcal{S}$', installed in ER, matches authorized $\mathcal{S}$, $\mathcal{D}$ev invokes *VRASED*[4] to compute $\sigma := HMAC(KDF(\mathcal{C}\mathsf{hal}, \mathcal{K}), \mathcal{S}')$. Verify outputs $\top$, iff $\sigma =$ ATok. In this case, PC reaches a fixed address, called i_{Auth}. Otherwise, it outputs $\perp$.

[4]$\mathcal{D}$ev and $\mathcal{C}$trl act as Prv and Vrf in *VRASED* respectively.

In the rest of this section, we use "authorized software" to refer to software located in ER, for which $\mathsf{Verify}(ER, \mathsf{ATok})$ outputs $\top$. Whereas, "unauthorized software" refers to any software for which $\mathsf{Verify}(ER, \mathsf{ATok})$ outputs $\perp$.

XSensing: When XSensing (ER) is invoked, PC jumps to ER_{min}, and starts executing the code in ER. It produces a set $\boldsymbol{E}$ of states by executing ER, and outputs $\top$, if there is at least one state that reads GPIO without triggering an MCU reset. Otherwise, it outputs $\perp$.

HardwareMonitor: VERSA HardwareMonitor is verified to enforce LTL specifications (3.9)–(3.19) in Construction 1.

A – Read-Access Control to GPIO is jointly specified by LTLs (3.9) and (3.10). LTL (3.9) states that GPIO can only be read during execution of ER ($PC \in ER$), requiring an MCU reset otherwise. LTL (3.10) forbids all GPIO reads (even those within ER execution) before successful computation of Verify on ER binary using a valid ATok. Successful Verify computation is captured by condition $PC = \mathsf{i}_{Auth}$. A new successful computation of $\mathsf{Verify}(ER, \mathsf{ATok})$ is necessary whenever ER execution completes ($PC = ER_{max}$) or after reset/boot. Hence, each legitimate ATok can be used to authorize ER execution once.

B – Ephemeral Immutability of ER and *METADATA* is specified by LTLs (3.11)-(3.13). From the time when ER binary is authorized until it starts executing, no modifications to ER or $METADATA$ are allowed. LTL (3.11) specifies that no such modification is allowed at the moment when verification succeeds ($PC = \mathsf{i}_{Auth}$); LTL (3.12) requires ER to be re-authorized from scratch if ER or $METADATA$ are ever modified. Whenever these modifications are detected ($Write\_Mem(ER) \lor Write\_Mem(METADATA)$) further reads to GPIO are immediately blocked ($\neg Read\_Mem(\mathsf{GPIO}) \lor reset$) until subsequent re-authorization of ER is completed (... $\mathbf{W}$ ($PC = \mathsf{i}_{Auth}$)). LTL (3.13) specifies the same requirement in order to read VERSA-provided encryption key ($\mathcal{K}_{enc}$) which is stored in memory region eKR. This property is only required when support for encryption of outputs is desired.

C – Atomicity & Controlled Invocation of ER are enforced by LTLs (3.14), (3.15), and (3.16). They specify that ER execution must start at ER_{min} and end at ER_{max}. Specifically, they use the relation between *current* and *next* PC values. The only legal PC transition from currently outside of ER to next inside ER is via $PC = ER_{min}$. Similarly, the only legal PC transition from currently inside ER to next outside ER is via $PC = ER_{max}$. All other cases trigger an MCU reset. In addition, LTL (3.16) requires an MCU reset whenever interrupts or DMA activity is detected during ER execution. This is done by simply checking irq and DMA_{en} signals.

We note that XSensing relies on the HardwareMonitor to reset the MCU when violations to ER atomic execution are detected. Upon reset all data is erased from memory. However, when execution of $\mathcal{S}$ completes successfully **VERSA** does not trigger resets. In this case, $\mathcal{S}$ is responsible for erasing its own stack before completion (reaching of ER its last instruction). We discuss how this self-clean-up routine can be implemented as a part of $\mathcal{S}$ behavior in Section 3.9.

3.6.2 Encryption & Integrity of ER Output

As mentioned in Section 3.3, after reading and processing `GPIO` inputs, $\mathcal{S}$ might need to encrypt and send the result to $\mathcal{C}$trl. **VERSA** supports encryption of this output, regardless of the underlying encryption scheme. For that purpose, Verify implementation derives a fresh one-time encryption key ($\mathcal{K}_{enc}$) from $\mathcal{K}$ and $\mathcal{C}$hal. To assure confidentiality of $\mathcal{K}_{enc}$, the following properties are required for the memory region (eKR) reserved to store $\mathcal{K}_{enc}$:

1. eKR is writable only by Verify (i.e., $PC \in VR$); and
2. eKR is readable only by ER after authorization.

LTLs (3.17)-(3.19) and (3.13) specify the confidentiality requirements of $\mathcal{K}_{enc}$. In sum, these properties establish the same read access-control policy for eKR and `GPIO` regions. Therefore, only authorized $\mathcal{S}$ is able to retrieve $\mathcal{K}_{enc}$.

3.7 Verified Implementation & Security Analysis

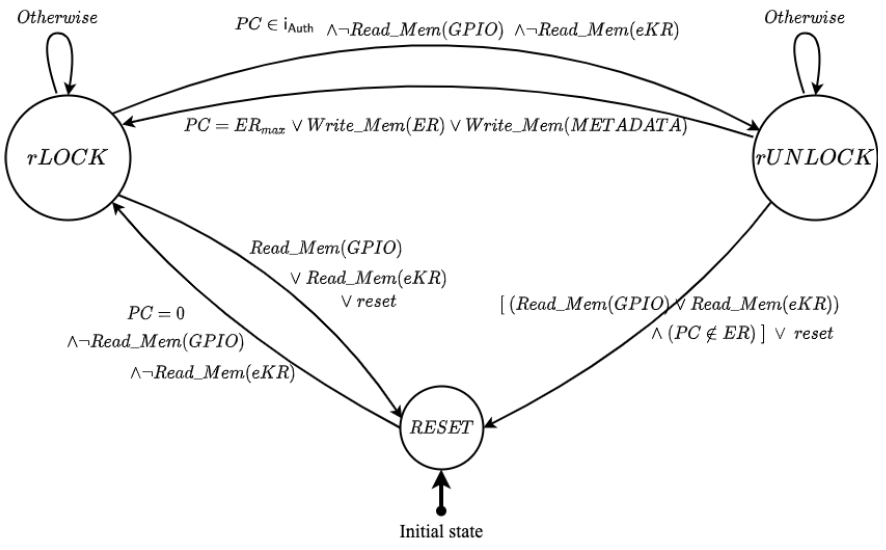
3.7.1 Sub-module Implementation & Verification

`VERSA` sub-modules are represented as FSMs and individually verified to hold for LTL properties from Construction 1. They are implemented in Verilog HDL as Mealy machines, i.e., their output is determined by both their current state and current inputs. Each FSM has a single output: a local $reset$. `VERSA` global output $reset$ is given by the disjunction (logic OR) of all local $reset$-s. For simplicity, instead of explicitly representing the output $reset$ value for each state, we use the following convention:

1. $reset$ is 1 whenever an FSM transitions to $RESET$ state;
2. $reset$ remains 1 while on $RESET$ state;
3. $reset$ is 0 otherwise.

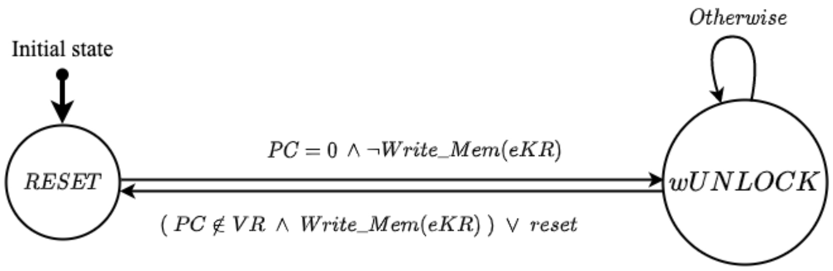
Note that all FSMs remain in $RESET$ state until $PC = 0$ which indicates that the MCU reset routine finished.

Fig. 3.6 illustrates the `VERSA` sub-module that implements read-access control to `GPIO` and eKR (when applicable). It guarantees that such reads are only possible when they emanate from execution of authorized software $\mathcal{S}$ contained in ER. It also assures that no modifications to ER or $METADATA$ occur between authorization of $\mathcal{S}$ and its subsequent execution. The Verilog implementation of this FSM is formally verified to adhere to LTLs (3.9)-(3.13) and (3.17)-(3.18). It has 3 states: (1) $rLOCK$, when reads to `GPIO` (and



Otherwise
$PC \in i_{Auth} \wedge \neg Read\_Mem(GPIO) \wedge \neg Read\_Mem(eKR)$
Otherwise
$PC = ER_{max} \vee Write\_Mem(ER) \vee Write\_Mem(METADATA)$
rLOCK
rUNLOCK
$Read\_Mem(GPIO) \vee Read\_Mem(eKR) \vee reset$
$PC = 0 \wedge \neg Read\_Mem(GPIO) \wedge \neg Read\_Mem(eKR)$
$[(Read\_Mem(GPIO) \vee Read\_Mem(eKR)) \wedge (PC \notin ER)] \vee reset$
RESET
Initial state

Figure 3.6: Verified FSM for **GPIO** and eKR Read-Access Control (LTL (3.9)-(3.13) & LTL (3.17)-(3.18))

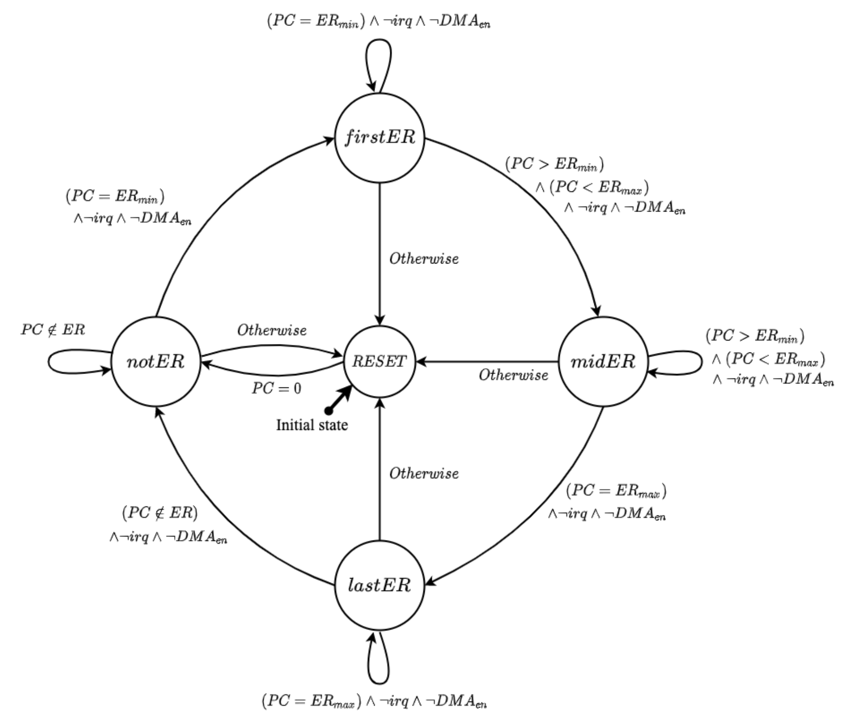


Otherwise
Initial state
RESET
$PC = 0 \wedge \neg Write\_Mem(eKR)$
wUNLOCK
$(PC \notin VR \wedge Write\_Mem(eKR)) \vee reset$

Figure 3.7: Verified FSM for eKR Write-Access Control (LTL (3.19))

possibly eKR) are disallowed; (2) $rUNLOCK$, when such reads are allowed to ER; and (3) $RESET$. The initial state (after reset or boot) is $RESET$, and it switches to $rLOCK$ state when $PC = 0$. It switches to $rUNLOCK$ when $PC = i_{Auth}$ (with no reads to **GPIO** and eKR), indicating that Verify was successful. Note that $rUNLOCK$ transitions to $RESET$ when reads are attempted from outside ER, thus preventing reads by any unauthorized software. Once PC reaches ER_{max}, indicating that ER execution has finished, the FSM transitions back to $rLOCK$. Also, any attempted modifications to $METADATA$ or ER in $rUNLOCK$ state bring the FSM back to $rLOCK$. Note that $rUNLOCK$ is only reachable after authorization of ER, i.e., $PC = i_{Auth}$.

The FSM in Figure 3.7 enforces LTL (3.19) to protect eKR from external writes. It has two



$(PC = ER_{min}) \land \neg irq \land \neg DMA_{en}$
$firstER$
$(PC > ER_{min})$ $\land (PC < ER_{max})$ $\land \neg irq \land \neg DMA_{en}$
$(PC = ER_{min})$ $\land \neg irq \land \neg DMA_{en}$
$Otherwise$
$PC \notin ER$
$notER$
$Otherwise$
$RESET$
$midER$
$(PC > ER_{min})$ $\land (PC < ER_{max})$ $\land \neg irq \land \neg DMA_{en}$
$PC = 0$
$Otherwise$
Initial state
$Otherwise$
$(PC = ER_{max})$ $\land \neg irq \land \neg DMA_{en}$
$(PC \notin ER)$ $\land \neg irq \land \neg DMA_{en}$
$lastER$
$(PC = ER_{max}) \land \neg irq \land \neg DMA_{en}$

Figure 3.8: ER Atomicity and Controlled Invocation FSM (LTL (3.14)-(3.16))

states: (1) $wUNLOCK$, when writes to eKR are allowed; and (2) $RESET$. At boot/after reset ($PC = 0$), this FSM transitions from $RESET$ to $wUNLOCK$. It transitions back to $RESET$ state whenever writes to eKR are attempted, unless these writes come from Verify execution ($PC \in VR$).

Figure 3.8 shows the FSM verified to enforce ER atomicity and controlled invocation: LTLs (3.14)-(3.16). It has five states; $notER$ and $midER$ correspond to PC being outside and within ER (not including ER_{min} and ER_{max}), respectively. $firstER$ and $lastER$ are states in which PC points to ER_{min} and ER_{max}, respectively. The only path from $notER$ to $midER$ is via $firstER$. Likewise, the only path from $midER$ to $notER$ is via $lastER$. The FSM transitions to $RESET$ whenever PC transitions do not follow aforementioned paths. It also transitions to $RESET$ (from any state other than $notER$) if irq or DMA_{en} signals are set.

3.7.2 Sub-module Composition and VERSA End-To-End Security

To demonstrate security of VERSA according to Definition 3.4, our strategy is two-pronged:

A) We show that LTL properties from Construction 1 are sufficient to imply that GPIO (and eKR) is only readable by $\mathcal{S}$ and any XSensing operation that returns $\top$ (i.e., performs sensing) is executed atomically. The former is formally specified in Definition 3.6, and the latter in Definition 3.5. For this part, we write an LTL computer proof using SPOT LTL proof assistant [131].

B) We use a cryptographic reduction to show that, as long as item **A** holds, *VRASED* security can be reduced to VERSA security according to Definition 3.4.

Definition 3.5. *Atomic Sensing Operation Execution:*

$$\begin{aligned} &\boldsymbol{G}\{\ (PC \in ER) \rightarrow [(PC \in ER) \land \neg irq \land \neg DMA_{en}]\ \ \boldsymbol{W}\ [(PC = ER_{max}) \lor reset]\ \} \\ \land\quad &\boldsymbol{G}\{\ \neg reset \land \neg(PC \in ER) \land \boldsymbol{X}(PC \in ER) \rightarrow \boldsymbol{X}(PC = ER_{min}) \lor \boldsymbol{X}(reset)\ \} \end{aligned}$$

Definition 3.6. *Mandatory Sensing Operation Authorization:*

$$\begin{aligned} &\boldsymbol{G}\{\ (Read\_Mem(\mathit{GPIO}) \land \neg reset) \rightarrow (PC \in ER)\ \} \land \\ &\Big\{(PC = \mathsf{i}_{Auth}) \land \{(PC = \mathsf{i}_{Auth}) \rightarrow \\ &\quad [\neg Write\_Mem(ER) \land \neg Write\_Mem(METADATA) \land (Write\_Mem(eKR) \rightarrow (PC \in VR))]\ \boldsymbol{U}\ (PC = ER_{min})\} \\ &\Big\}\ \boldsymbol{B}\ \{Read\_Mem(\mathit{GPIO}) \land \neg reset\} \end{aligned}$$

The intuition for this strategy is that, to win *PfB*-game in Definition 3.4, $\mathcal{A}$dv must either break the atomicity of XSensing (which is in direct conflict with Definition 3.5) or execute XSensing with unauthorized software and read GPIO without causing an MCU *reset*. Definition 3.6 guarantees that the latter is not possible without a prior successful call to Verify. On the other hand, Verify is implemented using *VRASED* verified architecture, which guarantees the unforgeability of ATok. Hence, breaking VERSA requires either violating VERSA verified guarantees or breaking *VRASED* verified guarantees, which should be infeasible to any PPT $\mathcal{A}$dv. To this end, we derive the following theorems showing the security of VERSA.

For the rest of this section, we first describe VERSA end-to-end implementation goals cap-

Theorem 3.1. *Definition 3.2 $\wedge$ LTL 3.14, 3.15, 3.16 $\rightarrow$ Definition 3.5*

Theorem 3.2. *Definition 3.2 $\wedge$ LTL 3.9, 3.10, 3.11, 3.12, 3.15, 3.19 $\rightarrow$ Definition 3.6*

Theorem 3.3. *VERSA is secure according to the PfB-game in Definition 3.4, as long as VRASED is a secure $\mathcal{R}$A architecture according to VRASED security game from [108].*

tured by LTLs in Definitions 3.5 and 3.6 as well as their relation to **VERSA** high-level features discussed in Section 3.3. Then, we prove the Theorems 3.1, 3.2, and 3.3 according to this proof strategy.

[Definition 3.5] states that it **globally** (always) holds that ER is atomically executed with controlled invocation. That is, whenever an instruction in ER executes ($PC \in ER$), it keeps executing instructions within ER ($PC \in ER$), with no interrupts and no DMA enabled, **until** PC reaches the last instruction in ER_{max} or an MCU reset occurs. Also, if an instruction in ER starts to execute, it always begins with the first instruction in ER_{min}. This formally specifies the *Atomic Sensing Operation Execution* feature discussed in Section 3.3.

[Definition 3.6] **globally** requires that whenever GPIO is successfully read (i.e., without a $reset$), this read must come from the CPU while ER is being executed. In addition, **before** this read operation, the following must have happened at least once:

(1) Verify succeeded (i.e., $PC = \mathsf{i}_{Auth}$);

(2) From the time when $PC = \mathsf{i}_{Auth}$ **until** ER starts executing (i.e., $PC = ER_{min}$), no modification to ER and $METADATA$ occurred; and

(3) If there was any write to eKR from the time when $PC = \mathsf{i}_{Auth}$, **until** $PC = ER_{min}$, it must have been from Verify, i.e., while $PC \in VR$.

This formally specifies the intended behavior of the *Mandatory Sensing Operation Authorization* feature, discussed in Section 3.3.

Now, we show that VERSA is a secure *PfB* architecture according to Definition 3.4, as long as **(a)** the sub-properties in Construction 1 hold (Theorem 3.1, 3.2) and **(b)** *VRASED* is a secure remote attestation ($\mathcal{R}$A) architecture according to the *VRASED* security definition in [108] (Theorem 3.3). Informally, part **(a)** shows that if the machine model and all LTLs in Construction 1 hold, then the end-to-end goals for secure *PfB* architecture are met, while this does not include the goal of prevention of forging authorization tokens. Part **(b)** handles the latter using a cryptographic reduction, i.e., it shows that an adversary able to forge the authorization token (with more than negligible probability) can also break *VRASED* according to the $\mathcal{R}$A-game, which is a contradiction assuming the security of *VRASED*. Therefore, Theorems 3.1-3.3 prove that VERSA is a secure *PfB* architecture as long as *VRASED* is a secure $\mathcal{R}$A architecture.

For part **(a)**, computer-checked LTL proofs are performed using SPOT LTL proof assistant [131]. These proofs are available at [25]. We present the intuition behind them below.

Proof of Theorem 3.1 (Intuition). LTL (3.15) states the legal entry instruction requirement, while LTL (3.14) states the legal exit instruction requirement in atomicExec. Also, since LTL (3.14) states that ER_{max} is the only possible exit of the ER without a reset, it implies self-contained execution of ER. Lastly, LTL (3.16) enforces MCU reset if any interrupt or DMA occurs, which naturally prevents interrupts and DMA actions, as required by atomicExec. These imply the LTL in Definition 3.5 which stipulates that execution of ER must start with ER_{min} and stays within ER with no interrupts nor DMA actions, until PC reaches ER_{max} (causing a reset otherwise). □

Proof of Theorem 3.2 (Intuition). Definition 3.6 **(i)** requires at least one successful verification of ER before GPIO can be read successfully (without triggering a reset); and **(ii)** disallows modifications to ER, $METADATA$, and $\mathcal{K}_{enc}$ (other than by VR) in between ER verification subsequent ER execution. LTLs (3.9) and (3.10) state that PC must be within ER to read GPIO and disallow GPIO reads by default (including when MCU reset occurs) and after the execution of ER is over ($PC = ER_{max}$). Also, LTL (3.10) requires (re-)authorization ($PC = \mathsf{i}_{Auth}$) of ER after the execution of ER is over ($PC = ER_{max}$). LTL (3.12) disallows **GPIO** reads until the (re-)verification whenever ER or $METADATA$ are written. LTL (3.11) disallows changes to ER and $METADATA$ at the exact time when verification succeeds. LTL (3.15) guarantees that the execution of ER starts with ER_{min} and LTL (3.19) guarantees that only the VR code can modify the value in eKR. Thus, these are sufficient imply Definition 3.6. □

For part **(b)**, we construct a reduction from the security game of *VRASED* in [108] to the security game of **VERSA** according to the Definition 3.4. i.e., The ability to break the *PfB*-game of **VERSA** allows to break the $\mathcal{R}\mathsf{A}$-game of *VRASED*, and therefore, as long as *VRASED* is a secure $\mathcal{R}\mathsf{A}$ architecture according to the $\mathcal{R}\mathsf{A}$-game, **VERSA** is secure according to the *PfB*-game.

Proof of Theorem 3.3. Denote by $\mathcal{A}\mathsf{dv}_{PfB}$, an adversary who can win the security game in Definition 3.4 against **VERSA** with more than negligible probability. We show that if such $\mathcal{A}\mathsf{dv}_{PfB}$ exists, then it can be used to construct $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ that wins the $\mathcal{R}\mathsf{A}$-game with more than negligible probability.

Recall that, to win the *PfB*-game, $\mathcal{A}\mathsf{dv}_{PfB}$ must trigger $\top$ as a result of $\mathsf{XSensing}$, which means it reads the sensed data without MCU reset. From the *PfB*-game step 4 in Definition 3.4, it can be done in either of the following two ways:

`Case1.` $\mathcal{A}\mathsf{dv}_{PfB}$ executes a new, unauthorized software $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$ causing $\mathsf{XSensing}(\mathcal{S}_{\mathcal{A}\mathsf{dv}}) \rightarrow \top$; or

`Case2.` $\mathcal{A}\mathsf{dv}_{PfB}$ breaks the atomic execution of an authorized, but not yet executed software, $\mathcal{S}_j$, so that it causes $\mathsf{XSensing}(\mathcal{S}_j) \rightarrow (E, \top)$ such that $\mathsf{atomicExec}(E, \mathcal{S}_j) \equiv \perp$.

Recall that for the instruction set I_j of $\mathcal{S}_j$ and a set E_j of execution states , to have $\mathsf{atomicExec}(I_j, E_j) \equiv \perp$, at least one of four requirements in Definition 3.4.1 must be false.

Note that the atomic sensing operation execution goal in Definition 3.5 rules out the probability of $\mathtt{Case2}$. Specifically, LTL (3.15) enforces 1), while 2) and 3) are guaranteed by LTL (3.14). Lastly, 4) is covered by LTL (3.16).

For $\mathtt{Case1}$, i.e., to trigger $\top$ by running $\mathsf{XSensing}$, $\mathcal{A}\mathsf{dv}_{PfB}$ needs to read $\mathtt{GPIO}$ without causing an MCU reset. Recall that the *Mandatory Sensing Operation Authorization* in Definition 3.6 requires Verify (with input executable in ER) to succeed at least once before reading $\mathtt{GPIO}$. According to VERSA construction, $\mathcal{A}\mathsf{dv}_{PfB}$ causes $\mathsf{Verify}(ER, \mathsf{ATok}^*, \mathcal{C}\mathsf{hal}^*)$ to output $\top$, where ER contains $\mathcal{S}_{\mathcal{A}\mathsf{dv}}$ which is an unauthorized software, ATok^* is a valid issued (but never used) token, and $\mathcal{C}\mathsf{hal}^*$ is its corresponding challenge. Since Verify function is implemented using *VRASED* to compute HMAC of $\mathcal{C}\mathsf{hal}$ and ER, such $\mathcal{A}\mathsf{dv}_{PfB}$ can be directly used as $\mathcal{A}\mathsf{dv}_{\mathcal{R}\mathsf{A}}$ to win the $\mathcal{R}\mathsf{A}$-game of *VRASED*. Thus, assuming secure $\mathcal{R}\mathsf{A}$ architecture *VRASED*, this is a contradiction, which implies the security of VERSA according to the *PfB*-game. □

3.8 Evaluation

In this section, we discuss VERSA implementation details and evaluation. VERSA source code and verification/proofs are publicly available at [25].

3.8.1 Toolchain & Prototype Details

VERSA is built atop OpenMSP430 [151]: an open source implementation of TI-MSP430 [179]. We use Xilinx Vivado to synthesize an RTL description of $\mathsf{HardwareMonitor}$ and deploy it on Diligent Basys3 prototyping board for Artix7 FPGA. For the software part (mostly to implement Verify), VERSA extends *VRASED* software (which computes $HMAC$ over $\mathcal{D}\mathsf{ev}$ memory) to include a comparison with the received ATok (See Section 3.8.3 for extension details). We use the NuSMV model checker to formally verify that HardwareMonitor implementation adheres to LTL specifications (3.9)-(3.19). See Section 3.8.4 for details on the verification setup and costs.

3.8.2 Hardware Overhead

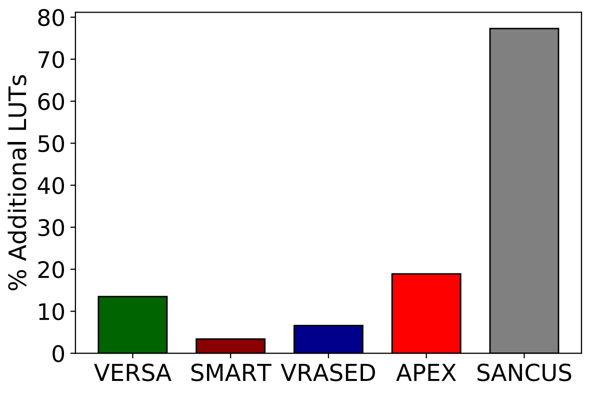
Table 3.2 reports on VERSA hardware overhead, as compared to unmodified OpenMSP430 and *VRASED*. Similar to other schemes [108, 110, 238, 252], we consider hardware overhead in terms of additional Look-Up Tables (LUTs) and registers. Extra hardware in terms of LUTs gives an estimate of additional chip cost and size required for combinatorial logic, while extra hardware in terms of registers gives an estimate of memory overhead required by sequential logic in VERSA FSMs. Compared to *VRASED*, VERSA requires 10% additional LUTs and 2% additional registers. In actual numbers, it adds 255 LUTs and 50 registers to the underlying MCU as shown in Table 3.2.

Table 3.2: Hardware Overhead and Verification cost

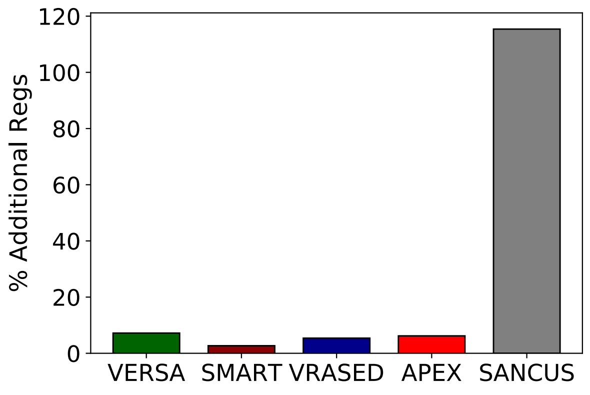
Architecture	Hardware		Reserved	Verification			
	LUTs	Regs	RAM (bytes)	LoC	#(LTLs)	Time (s)	RAM (MB)
OpenMSP430	1854	692	0	-	-	-	-
VRASED	1891	724	2332	481	10	0.4	13.6
VERSA + *VRASED*	2109	742	2336	1118	21	13956.4	1059.1

3.8.3 Runtime Overhead

VERSA requires any software piece seeking to access GPIO (and $\mathcal{K}_{enc}$) to be verified. Consequently, runtime overhead is due to Verify computation which instantiates *VRASED*. This runtime includes: **(1)** time to compute σ from equation 3.7; **(2)** time to check if $\sigma =$ ATok; and **(3)** time to write $\mathcal{K}_{enc}$ to eKR, when applicable. Naturally the runtime overhead is dominated by the computation in **(1)** which is proportional to the size of ER.



(a) Additional HW overhead (%) in the number of Look-Up Tables



(b) Additional HW overhead (%) in the number of Registers

Figure 3.9: Hardware overhead comparisons with other low-end security architectures.

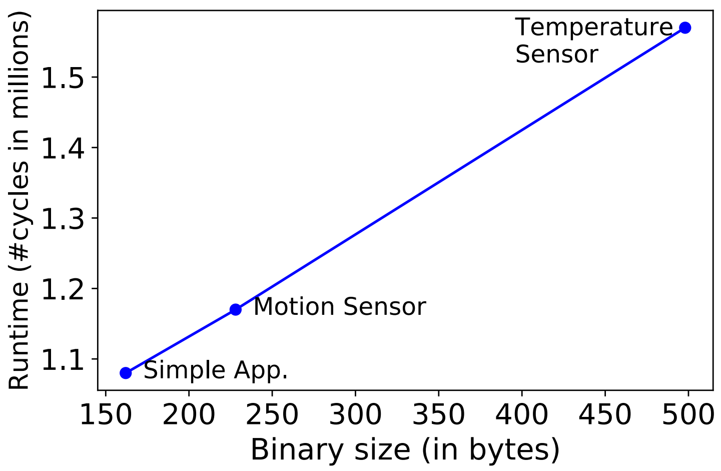


Figure 3.10: Runtime overhead of VERSA due to Verify

We measure Verify cost on three sample applications: (1) Simple Application, which reads 32-bytes of GPIO input and encrypts it using One-Time-Pad (OTP) with $\mathcal{K}_{enc}$; (2) Motion Sensor – available at [7] – which continuously reads GPIO input to detect movements and actuates a light source when movement is detected; and (3) Temperature Sensor – adapted code from [14] to support encryption of its outputs – which reads ambient temperature via GPIO and encrypts this reading using OTP. We prototype using OTP for encryption for the sake of simplicity noting that VERSA does not mandate a particular encryption scheme. All of these sample applications also include a self-clean-up code executed immediately before reaching their exit point to erase their stack traces once their execution is over.

Figure 3.10 shows Verify runtimes on these applications. Assuming a clock frequency of 10MHz (a common frequency for low-end MCUs), Verify runtime ranges from $100 - 200$ milliseconds for these applications. The overhead is linear on the binary size.

3.8.4 Verification Cost

Formal verification costs are reported in Table 3.2. We use a Ubuntu 18.04 desktop machine running at 3.4GHz with 32GB of RAM for formal verification. Our verification pipeline converts Verilog HDL to SMV specification language and then verifies it against the LTL properties listed in Construction 1 using the NuSMV model checker (per Section 3.2). VERSA verification requires checking 11 extra invariants – LTLs (3.9) to (3.19) – in addition to *VRASED* LTL invariants. It also incurs higher run-time and memory usage than *VRASED* verification. This is due to two additional 16-bit hardware signals (ER_{min}, ER_{max}) which increase the space of possible input combinations and thus the complexity of model checking process. However, verification is still manageable in a commodity desktop – it takes around 5 minutes and consumes 340MB of memory.

3.8.5 Comparison with Other Low-End Architectures:

To the best of our knowledge, VERSA is the first architecture related to *PfB*. However, to provide a point of reference in terms of performance and overhead, we compare VERSA with other low-end trusted hardware architectures, such as SMART [252], VRASED [108], APEX [110], and SANCUS [238]. All these architectures provide $\mathcal{R}$A-related services to attest integrity of software on $\mathcal{D}$ev either statically or at runtime. Since *PfB* also checks software integrity before granting access to GPIO, we consider these architectures to be related to VERSA.

Figure 3.9 compares VERSA hardware overhead with the aforementioned architectures in terms of additional LUTs and registers. Percentages are relative to the plain MSP430 core total cost.

VERSA builds on top of *VRASED*. As such, it is naturally more expensive than hybrid $\mathcal{R}$A architectures such as SMART and *VRASED*. Similar to VERSA, APEX also monitors execution properties and also builds on top of $\mathcal{R}$A (in APEX case, with the goal of producing proofs of remote software execution). Therefore, VERSA and APEX exhibit similar overheads. SANCUS presents a higher cost because it implements $\mathcal{R}$A and isolation features in hardware.

3.9 Discussion & Limitation

3.9.1 Clean-up after Program Termination

While VERSA guarantees the confidentiality of sensing operations, it requires the authorized executable $\mathcal{S}$ to erase its own stack/heap before its termination. This ensures that unauthorized software can not extract and leak sensitive information from $\mathcal{S}$ execution and allocated data. Erasure in this case can be achieved via a single call to libc's `memset` function with start address matching the base of $\mathcal{S}$ stack and size equal to the maximum size reached by $\mathcal{S}$ execution.

The maximum stack size can be determined manually by counting the allocated local variables in small and simple $\mathcal{S}$ implementations. To automatically determine this size in more complex $\mathcal{S}$ implementations, all functions called within $\mathcal{S}$ must update the highest point reached by their respective stacks. Figure 3.11 shows a sample application that reads 32 bytes of sensor data, encrypts this data using `VERSA` one-time key $\mathcal{K}_{enc}$, and cleans-up the stack thereafter. Line 12 is $\mathcal{S}$ entry point (ER_{min}). $\mathcal{S}$ first saves the stack pointer to `STACK_MIN` address. Then, the `application` function is called, in line 17. `application` implements $\mathcal{S}$ intended behavior. After the `application` is done, the clean up code (lines 51-53) is called with `STACK_MIN` as the start pointer and size of 32 + 4 bytes (32 bytes for `data` variable (in line 39) and 4 bytes for stack metadata).

3.9.2 Data Erasure on Reset/Boot

Violations to `VERSA` properties trigger an MCU reset. A reset immediately stops execution and prepares the MCU core to reboot by clearing all registers and pointing the program counter (PC) to the first instruction of PMEM. However, some MCUs may not guarantee erasure of DMEM as a part of this process. Therefore, traces of data allocated by $\mathcal{S}$ (including sensor data) could persist across resets.

In MCUs that do not offer DMEM erasure on reset, a software-base Data Erasure (DE) can be implemented and invoked it as soon as the MCU starts, i.e., as a part of the bootloader code. In particular, DE can be implemented using `memset` (similar to lines 51-53) with constant arguments matching the entirety of the MCU's DMEM.DE should be immutable

```
#include <string.h>
// Sensor is at P3IN
#define P3IN          (*(volatile unsigned char *) 0x0018)
#define BIT4          0x10
#define HIGH          0x1
#define LOW           0x0
#define eKR           0x360
#define STACK_MIN     0x1010 // App stack start pointer is stored here
#define RESULT        0x1030 // App result is written here

// ER_min
__attribute__ ((section (".er.entry"), naked)) void applicationEntry() {
    // Save the stack pointer to STACK_MIN at entry.
    __asm__ volatile("mov    r1,   &0x1010" "\n\t");

    // Call the application
    application();

    // Call the clean up code
    cleanUp((uint8_t*)STACK_MIN, 32 + 4);

    // Jump to applicationExit()
    __asm__ volatile( "br #__er_leave" "\n\t");
}

__attribute__ ((section (".er.body"))) int digitalRead() {
    if(P3IN & BIT4) return HIGH;
    else return LOW;
}

__attribute__ ((section (".er.body"))) void encrypt(uint8_t* data, int
        data_size, uint8_t* key, int key_size) {
    for (int i = 0; i < data_size; i++) {
        data[i] = data[i] ^ key[i];
    }
}

__attribute__ ((section (".er.body"))) void application() {
    // Read sensor data
    uint8_t data[32];
    for (int i = 0; i < 32; i++) {
        data[i] = digitalRead();
    }

    // One-Time-Pad encryption of data using the enc_key in eKR.
    encrypt(data, 32, (uint8_t*)eKR, 32);

    // Copy encrypted result to RESULT address.
    memcpy(RESULT, data, 32);
}

__attribute__ ((section (".er.body"))) void cleanUp(uint8_t* start, int size){
    memset((uint8_t*)start, 0, size);
}

//ER_max
__attribute__ ((section (".er.exit"), naked)) void applicationExit() {
    __asm__ volatile("ret" "\n\t");
}
```

Figure 3.11: Sample sensing operation that reads GPIO input, encrypts it, and cleans up its stack after execution.

(e.g., stored in ROM) which is often the case for bootloader binaries. Upon reset, PC must always point to the first instruction of DE. The normal MCU start-up proceeds normally after DE execution is completed.

3.9.3 VERSA Limitations

Shared Libraries

to verify $\mathcal{S}$, $\mathcal{C}\mathsf{trl}$ must ensure that $\mathcal{S}$ spans one contiguous memory region (ER) on $\mathcal{D}\mathsf{ev}$. If any code dependencies exist outside of ER, VERSA will reset the MCU according to LTL (3.16). To preclude this situation, $\mathcal{S}$ must be made self-contained by statically linking all of its dependencies within ER.

Atomic Execution & Interrupts

per Definition 3.4, VERSA forbids interrupts during execution of XSensing. This can be problematic, especially on a $\mathcal{D}\mathsf{ev}$ with strict real-time constraints. In this case, $\mathcal{D}\mathsf{ev}$ must be reset in order to allow servicing the interrupt after DMEM erasure. This can cause a delay that could be harmful to real-time settings. Trade-offs between privacy and real-time constraints should be carefully considered when using VERSA. One possibility to remedy this issue is to allow interrupts as long as all interrupt handlers are: (1) themselves immutable and uninterruptible from the start of XSensing until its end; and (2) included in ER memory range and are thus checked by Verify.

Possible Side-channel Attacks

MSP430 and similar MCU-s allow configuring some GPIO ports to trigger interrupts. If one of such ports is used for triggering an interrupt, $\mathcal{A}$dv could possibly look at the state of $\mathcal{D}$ev and learn information about `GPIO` data. For example, suppose that a button press mapped to a GPIO port triggers execution of a program that sends some fixed number of packets over the network. Then, $\mathcal{A}$dv can learn that the GPIO port was activated by observing network traffic. To prevent such attacks, privacy-sensitive quantities should always be physically connected to GPIO ports that are not interrupt sources (these are usually the vast majority of available GPIO ports). Other popular timing attacks related to cache side-channels and speculative execution, are not applicable to this class of devices, as these features are not present in low-end MCUs.

Flash Wear-Out

VERSA implements Verify using *VRASED*. As discussed in Section 3.2.3, the authentication protocol suggested by *VRASED* requires persistent storage of the highest value of a monotonically increasing challenge/counter in flash. We note that flash memory has a limited number of write cycles (typically at least 10,000 cycles [23, 18]). Hence, a large number of successive counter updates may wear-out flash causing malfunction. In *VRASED* authentication, the persistent counter stored in flash is only updated following successful authentication of $\mathcal{C}$trl. Therefore, only legitimate requests from $\mathcal{C}$trl cause these flash writes, limiting the capability of an attacker to exploit this issue. Nonetheless, if the number of expected legitimate calls to Verify is high, one must select the persistent storage type or (alternatively) use different flash blocks once a given flash block storing the counter reaches its write cycles' limit. For a more comprehensive discussion of this matter, see [37].

VERSA Alternative Use-Case:

VERSA can be viewed as a general technique to control access to memory regions based on software authorization tokens. We apply this framework to GPIO in low-end MCUs. Other use-cases are possible. For example, a VERSA-like architecture could be used to mark a secure storage region and grant access only to explicitly authorized software. This could be useful if $\mathcal{D}$ev runs multiple (mutually distrusted) applications and data must be securely shared between subsets thereof.

3.10 Related Work

There is a considerable body of work (overviewed in Section 3.1) on IoT/CPS privacy. However, to the best of our knowledge, this paper is the first effort specifically targeting *PfB*, i.e., sensor data privacy on potentially compromised MCUs. Nonetheless, prior work has proposed trusted hardware/software co-designs – such as VERSA – offering other security services. We overview them in this section.

Trusted components, commonly referred to as Roots of Trust (RoTs), are categorized as software-based, hardware-based, or hybrid (i.e., based on hardware/software co-designs). Their usual purpose is to verify software integrity on a given device. Software-based RoTs [190, 274, 273, 272, 148, 209, 152] usually do not rely on any hardware modifications. However, they are insecure against compromises to the entire software state of a device (e.g., in cases where $\mathcal{A}$dv can physically re-program $\mathcal{D}$ev). In addition, their inability to securely store cryptographic secrets imposes reliance on strong assumptions about precise timing and constant communication delays to enable device authentication. These assumptions can be unrealistic in the IoT ecosystem. Nonetheless, software-based RoTs are the only viable choice for legacy devices that have no security-relevant hardware support. Hardware-based methods [247, 290, 199, 267, 226, 225, 238] rely on security provided by dedicated hardware components (e.g., TPM [290] or ARM TrustZone [38]). However, the cost of such hardware is normally prohibitive for low-end MCUs. Hybrid RoTs [252, 110, 108, 67, 195] aim to achieve security equivalent to hardware-based mechanisms, yet with lower hardware costs. They leverage minimal hardware support while relying on software to reduce additional hardware complexity.

Other architectures, such as SANCTUM [93] and Notary [43], provide strong memory isolation and peripheral isolation guarantees, respectively. These guarantees are achieved via hardware support or external hardware agents. They are also hybrid architectures where trusted hardware works in tandem with trusted software. However, we note that such schemes are designed for application computers that support MMUs and are therefore unsuitable for simple MCUs.

In terms of functionality, such embedded RoTs focus on integrity. Upon receiving a request from an external trusted *Verifier*, they can generate unforgeable proofs for the state of the MCU or that certain actions were performed by the MCU. Security services implemented by them include: (1) memory integrity verification, i.e., remote attestation [252, 238, 108, 32, 67, 195]; (2) verification of runtime properties, including control-flow and data-flow attestation [110, 120, 26, 121, 304, 284, 116, 115, 149]; and (3) proofs of remote software updates, memory erasure, and system-wide resets [109, 31, 39]. As briefly mentioned in Section 3.1, due to their reactive nature, they can be used to *detect* whether $\mathcal{D}\mathsf{ev}$ has been compromised *after the fact*, but *cannot prevent* the compromised entity from exfiltrating private sensor data. VERSA, on the other hand, enforces mandatory authorization before any sensor data access and thus prevents leakage even when a compromise has already happened.

Formalization and formal verification of RoTs for MCUs have gained attention due to the benefits discussed in Sections 3.1 and 3.2. VRASED [108] implemented a verified hybrid $\mathcal{R}\mathsf{A}$ scheme. APEX [110] built atop VRASED to implement and formally verify an architecture for proofs of remote execution of attested software. PURE [109] implemented provably secure services for software update, memory erasure, and system-wide reset. Another recent result [70] formalized and proved the security of a hardware-assisted mechanism to prevent leakage of secrets through timing side-channels due to MCU interrupts. Inline with the aforementioned work, VERSA also formalizes its assumptions along with its goals and implements the first formally verified design assuring *PfB*.

3.11 Summary

We formulated the notion of *Privacy-from-Birth* (*PfB*) and proposed VERSA: a formally verified architecture realizing *PfB*. VERSA ensures that only duly authorized software can access sensed data even if the entire software state of the sensor is compromised. To attain this, VERSA enhances the underlying MCU with a small hardware monitor, which is shown sufficient to achieve *PfB*. The experimental evaluation of VERSA publicly available prototype [25] demonstrates its affordability on a typical low-end IoT MCU: TI MSP430.

Chapter 4

Element Distinctness and Bounded Input Size in Private Set Intersection and Related Protocols

This chapter shows when the lack of input correctness matters in MPC (see Chapter 5 for the details), with a special case, PSI, with input size limits. It suggests how to prove benign inputs to obtain the result, without revealing them to the other party. This motivates checking the input validity as a security guarantee in MPC with input conditions.

4.1 Introduction

Private Set Intersection (PSI) is a special case of secure multi-party computation (MPC) that computes the intersection of the private input sets, without revealing any information about the set elements outside the intersection. It attracted a lot of attention from various privacy-preserving applications, such as contact tracing [130, 286], online targeted advertising [181], genomic testing [197], botnet detection [233], TV program history matching [187], private contact discovery [118, 160], and private matchmaking [309].

Due to its functionalities applicable to numerous real-world applications, there has been a long line of work in PSI and its variants (details in Section 4.2), starting from the earliest forms in 1980s [228, 277]. While most PSI protocols reveal the input sizes as part of the protocol, Ateniese et. al. [42] constructed the first PSI variant – *Size-Hiding PSI (SH-PSI)* – that allows one party (*Client*) to learn the intersection while keeping its input set size private against the other party (*Server*). Building upon this size-hiding property, Bradley et. al. [66] and Cerulli et. al. [80] suggested upper-bounding *Client*'s input set size to prevent it from learning too much information about *Server*'s input set.

This work started from a related question: lower-bounding *Client*'s input set size while keeping it private. Suppose that *Server* requires *Client* to have at least l elements in its input set to run the PSI protocol with *Server*'s set. This requirement might be useful in social network settings, such as Facebook and LinkedIn, where a popular/prominent user would agree to connect to another user only if the latter has at least l genuine friends/followers to e.g., block the stalkers who keep creating bogus accounts and requesting to connect.

If we relax the size-hiding property, it seemed straightforward to lower-bound *Client* input size: *Server* simply checks whether the *Client* set size (revealed as part of the PSI interaction) is $\geq l$, and if not, aborts the protocol. However, this only works if *Client* is honest. A dishonest *Client* can bypass this requirement by (1) generating and using fake set elements, and/or (2) duplicating its genuine set elements. Then, since PSI protocols typically obfuscate (often by blinding) *Client* set elements, *Server* cannot distinguish between the genuine and fake input elements.

One intuitive way to mitigate this misbehavior is via auditing: a trusted third party (TTP) regularly verifies the *Client* input by examining the transcripts of PSI protocol and looking for duplicate or spurious elements. However, this would be too late since the dishonest *Client* already obtained the intersection.

To deal with the type-(1) misbehavior, so-called *Authorized PSI* (APSI) techniques [105, 106, 299] have been proposed. This is achieved with an offline TTP that pre-authorizes *Client* input by signing each element. Later, during PSI interaction, *Server* (implicitly or explicitly) verifies these signatures without learning *Client* input. This way, *Client* cannot obtain signatures of spurious elements, and thus, cannot learn the intersection using fake elements. However, APSI protocols cannot cope with the type-(2) misbehavior, i.e., *Client* can still bypass the requirement by using duplicated (TTP-authorized) signed elements. This prompts a natural question:

Can Client prove that each of its private input elements is not duplicated, i.e., all input elements are distinct while keeping them private?

To answer this question, we first investigate if current PSI protocols can detect duplicates (see Section 4.2.4). A few prior results [59, 192] proposed *Private Multiset*[1] *Intersection (PMI)* protocols which allow *multiset* inputs. However, we note that their goal is different because it outputs the intersection *multiset*, not the intersection *set*, which yields more information than PSI, e.g., the number of occurrences (i.e., multiplicities) of common elements.

Then we show how to prove *element distinctness* in two-party settings, whereby one party convinces the other that its input elements are all distinct, without revealing any information about them. We use the term *element distinctness* (a.k.a. *element uniqueness*) problem from the computational complexity theory: given n numbers $x_1, ..., x_n$, return "Yes" if all x_i's are distinct, and "No" otherwise. To the best of our knowledge, there is no prior work in the two-party settings where one party *proves element distinctness of its private input* to the other party. We call this ***P**roofs **o**f **E**lement-**D**istinctness (**PoED**).*

[1]Recall that a multiset allows duplicate elements, while a set does not.

We propose a concrete PoED construction by generalizing the two billiard balls problem, which can be an independent interest. Using this PoED construction as a building block, we propose a new PSI variant, called *All-Distinct Private Set Intersection (AD-PSI)*, and its construction. Informally speaking, AD-PSI allows *Client* to learn the intersection only if all of its input elements are distinct. It additionally guarantees that *Client* learns no information, not even *Server* input size, if it uses any duplicates as input.

Then, we extend AD-PSI to three PSI variants where using duplicates can be more problematic: (1) *AD-PSI-Cardinality (AD-PSI-CA)* that outputs the cardinality of the intersection; (2) *Existential AD-PSI (AD-PSI-X)* that outputs whether the intersection is non-empty; and (3) *AD-PSI with Data Transfer (AD-PSI-DT)* that transfers associated data along with the intersection; only when *Client* inputs all distinct elements. Finally, we construct a *Bounded-Size-Hiding-PSI (B-SH-PSI)* protocol with *both* upper and lower bound on *Client* input, combining our AD-PSI with prior work on upper-bounded size-hiding PSI (U-SH-PSI) [66, 80]. This also shows the applicability of PoED and AD-PSI.

Note that the protocols above work where *Client* cannot generate fake elements, and to expand *Client*'s capabilities to both type-(1) and type-(2) misbehavior, including a TTP is unavoidable. To fill this gap, we lastly present an *All-Distinct Authorized PSI (AD-APSI)* protocol that prevents both duplicate and spurious elements by ensuring the validity of *Client* input. We specify desired security properties for AD-APSI and prove that the proposed protocol satisfies them.

To summarize, the contributions of this work are:

- A PoED protocol with security analysis;
- Definition of AD-PSI and concrete construction with security proofs;
- Three AD-PSI variants: AD-PSI-CA, AD-PSI-X, AD-PSI-DT;
- Extension of U-SH-PSI to B-SH-PSI with both upper and lower bounds on *Client*

input set size; and

- Definition of AD-APSI and concrete construction with security proofs;

Organization: After overviewing related work and preliminaries in Section 4.2, Section 4.3 presents a PoED construction and its analysis. Then, Section 4.4 provides the definition and a construction of secure AD-PSI protocol, followed by the variants in Section 4.5. Section 4.6 shows a B-SH-PSI construction atop U-SH-PSI, and Section 4.7 presents an AD-APSI protocol and its security proofs.

4.2 Related Work & Background

4.2.1 Private Set Intersection

PSI in two-party computation is an interaction between *Client* and *Server* that computes the intersection of their private input sets. A long line of work on PSI can be classified according to the underlying cryptographic techniques: (1) Diffie-Hellman key agreement [66, 105, 174, 228, 277]; (2) RSA [104, 106, 107]; (3) cryptographic accumulators [42, 106]; (4) oblivious transfer (or oblivious pseudorandom function) [82, 164, 167, 184, 196, 237, 249, 256, 251, 253, 254, 255, 261]; (5) Bloom filter [117, 129, 250, 255]; (6) oblivious polynomial evaluation [97, 144, 146, 165, 192]; and (7) generic multiparty computation [129, 173, 255, 88, 219]. This work considers the *one-sided* PSI where *Client* learns the result. Most efficient protocols incur $O(n)$ computation/communication costs, where n is the input set size.

4.2.2 PSI Variants

Some PSI variants reveal less information than the actual intersection. For example, PSI-CA [104, 117, 130, 266, 292] outputs only the cardinality of the intersection, and PSI-

X [77] outputs a one-bit value reflecting whether the intersection is non-empty. On the other hand, some reveal more information, such as associated data for each intersecting element [106, 308] or additional private computation results (e.g., sum or average) along with the intersection [181, 229, 202, 219]. The latter is more interesting because of their realistic applications, such as statistical analysis for, e.g., advertisement conversion rate [181], of intersecting data.

4.2.3 PSI with Restrictions

Certain PSI variants place conditions for *Client* to obtain the result. For example, *threshold PSI* (t-PSI) reveals the intersection only if the cardinality of the intersection meets a *Server*-set threshold value [146, 150, 162, 254, 308, 309], and its variants, such as t-PSI-CA or t-PSI-DT (also called, *threshold secret transfer)* [308], reveals the intersection or additional data only when the threshold restriction is met or reveals only the cardinality, otherwise. Zhao and Chow [308] extend this to PSI with a generic access structure so that *Client* can learn the result only when the intersection satisfies a certain structure. Also, they build the below/over t-PSI [309] such that *Client* can reconstruct the secret key used by *Server* only when the threshold condition is met, which inspires some steps in our protocols.

On the other hand, Bradley et. al. [66] first suggest the *Bounded-Size-Hiding-PSI* which restricts the *Client input*, i.e., *Client* learns the intersection only if the size of its *input* does not exceed a *Server*-set upper bound in the random oracle model, and later Cerulli et al. [80] improve it to be secure in the standard model. Compared to the other PSI literature that naturally reveals the input set sizes during the computation, [66] and [80] also hide cardinality information. We note that there has been no PSI variant that places a lower bound (or both lower and upper bounds) on *Client* input.

4.2.4 PSI with Multiset Input

We now consider how current PSI protocols handle multisets. Note that adversary models in PSI literature do include malicious *input*. Loosely speaking, Honest-but-Curious (HbC) (a.k.a. semi-honest) adversaries try to learn as much as possible while honestly following the protocol, whereas malicious adversaries arbitrarily deviate from the protocol. However, according to Lindell and Pinkas [211], such adversaries can not be prevented from refusing to participate in a protocol, supplying arbitrary input, or prematurely aborting a protocol instance. Since PSI security is generally based on *sets*, *multisets* can be viewed as malicious inputs. PSI protocols do not offer security against multiset inputs. i.e., Security against malicious adversaries does not mean that multiset inputs are “automatically" handled.

It turns out that some PSI protocols are incompatible with multiset inputs because they assume set input, i.e., distinctness of all elements. For example, [59] and [173] obliviously sort elements and compare the adjacent elements to compute the intersection by checking for equality [173] or erasing each element once [59]. Thus, these protocols output incorrect results with multiset inputs. Furthermore, PSI protocols based on Cuckoo hashing [144, 256, 254, 255] can encounter unexpected errors with multiset inputs. Cuckoo hashing maps each input element into a hash table using some hash functions such that each bin contains at most one element. Since the hash of the same input value is always the same, duplicates can cause an infinite loop (to find an available bin) or result in a waste of resources, e.g., repeating steps until a certain threshold and increasing the stash size.

There exist some PSI protocols that either enforce input element distinctness or are compatible with multiset inputs. For example, the party creates a polynomial that has roots on its input values in [97, 146, 192] to perform oblivious polynomial evaluation, which by nature filters the duplicates. [250] also guarantees the set input by a new data structure, called PaXoS, which disables encoding any non-distinct elements. On the other hand, the security of [164, 184] is unaffected by duplicates because it uses an oblivious pseudo-random function to obtain some (random-looking) numbers for its private elements and then compare the received messages.

Our work focuses on PSI protocols incompatible with multiset inputs and suggests adding some simple steps to ensure the element distinctness of private input.

4.2.5 Zero-Knowledge Proofs

The notion of Zero-Knowledge Proof (ZKP) is first introduced by [156] which is the zero-knowledge interactive proof system. Informally, an *interactive proof system* for a language L is defined between a prover (Prv) and a verifier (Vrf) with a common input string x and unbiased coins, where Prv tries to convince Vrf that x is indeed in L while keeping their coin tosses private. It must be *complete*, i.e., for any $x \in L$, Vrf accepts, and *sound*, i.e., for any $x \notin L$, Vrf rejects no matter what Prv does. The interactive proof is *zero-knowledge* if given only x, Vrf could simulate the entire protocol transcript by itself without interacting with Prv. A *proof-of-knowledge* [142, 54] is an interactive proof where Prv tries to convince Vrf that it has "knowledge" tying the common input x, which requires the *completeness* and *knowledge extractibility* (stronger notion of *soundness*) properties. *Knowledge extractibility* (a.k.a.*validity*) is that for any Prv who can make Vrf accept its claim with non-negligible probability, there exists an efficient program K called *knowledge extractor*, such that K can interact with Prv and output a witness w of the statement $x \in L$. *Zero-Knowledge Proof of Knowledge (ZKPoK)* adds the *zero-knowledge* property on top of them. Compared to ZKP, ZKPoK keeps the one-bit information (whether $x \in L$ or not) private from Vrf, thus realizing "zero"-knowledge.

4.2.6 Homomorphic Encryption

Homomorphic encryption (HE) is a special type of encryption that allows users to perform certain arithmetic operations on encrypted data such that results are meaningfully reflected in the plaintext. It is called *Fully Homomorphic Encryption (FHE)* when a HE supports *both* unlimited addition and multiplication of ciphertexts. Whereas, a scheme that supports a limited number of operations of either type is called *Somewhat Homomorphic Encryption (SWHE)* and a scheme that supports only one operation type is called *Partially Homomorphic Encryption (PHE).* There are many PHE schemes such as [91, 96, 100, 136, 155, 242, 232, 240, 189, 262]. For example, ElGamal encryption scheme [136] is a well-known PHE supporting multiplication, and a variant of ElGamal [96] having g^m instead of m and Paillier [242] are well-known PHE schemes supporting addition.

4.3 Proving Element Distinctness

We first define ***P**roofs **o**f **E**lement-**D**istinctness (PoED)* in the two-party settings where Prv proves element distinctness of its private input elements to Vrf. i.e., PoED is an interactive proof system, where Prv tries to convince Vrf that its input $\mathcal{C} := [c_1, ..., c_n]$ consists of distinct elements, without revealing any other information about each c_i. As a result, Vrf *accepts* or *rejects* the Prv's claim. Following the notation for ZKPoK introduced by [73], PoED is denoted as:

$$PK\{\mathcal{C} \ \mid e_i = f(c_i) \text{ for each } c_i \in \mathcal{C}, \text{ and } c_i \neq c_j \text{ for } \forall c_i, c_j \in \mathcal{C} \text{ such that } i \neq j\},$$

where f is a function that “hides" c_i so that Vrf does not learn any information about c_i from e_i, while it “binds" c_i to e_i so that Prv cannot change c_i once e_i is computed, e.g., via randomized encryption or cryptographic commitments.

Proving Element Distinctness with λ Puzzles

Public: $G = \langle g \rangle$, a group with operator $\cdot$, and λ: a sec. param.
pk : the public key of Prv, while correlated sk kept private

```
$Prv\ (\mathcal{C} = [c_1, ..., c_n])$                                 $Vrf\ (\perp)$
for $i = 1, ..., n$ :
  $e_i := Enc_{pk}(c_i)$      $\xrightarrow{(e_1, ..., e_n)}$      for $k = 1, ..., \lambda$ :
                                                              $\pi_k \in_R \mathcal{P}_n$
                                                              for $i = 1, ..., n$ :
                                                                $e_{i,k} := e_i \cdot Enc_{pk}(u)$
                                                              $E_k := \pi_k(e_{1,k}, ..., e_{n,k})$
for $k = 1, ..., \lambda$ :    $\xleftarrow{E_1, ..., E_\lambda}$      $key \leftarrow H(\pi_1, ..., \pi_\lambda)$
  Determine $\pi'_k$ s.t.
  $Dec_{sk}(E_k) = \pi'_k(\mathcal{C})$
$key' \leftarrow H(\pi'_1, ..., \pi'_\lambda)$   $\xrightarrow{key'}$      return $Accept$, if $key = key'$
                                                            return $Reject$, otherwise
```

Figure 4.1: The PoED-puzzle Protocol

4.3.1 Puzzle-Based PoED Construction

The main idea starts from the well-known *two billiard balls* problem, where Prv has two billiard balls, and (honest) Vrf is color-blind. To convince Vrf that two balls have different colors, the following "*puzzle*" is repeated k times:

1. Prv puts a ball in each hand of Vrf
2. Vrf puts both hands behind its back and decides (at random) whether to switch the balls or not
3. Vrf shows the balls to Prv
4. Prv declares whether a switch occurred
5. If Prv answers incorrectly, Vrf concludes that Prv cheated

If Prv answers correctly k times, Vrf concludes that, with probability 2^{-k}, the balls have different colors.

Extending this problem to many balls, we construct a PoED protocol and call it *PoED-puzzle* protocol. Instead of the color-blind Vrf, Prv encrypts each element with its public key under an encryption scheme satisfying the ciphertext indistinguishability (IND) property. Since all IND-secure encryption schemes are non-deterministic, Prv can hide the information about the input elements.

To form the puzzles such that Vrf can generate while Prv can solve only when all input elements are distinct, PoED-puzzle needs a PHE scheme over a cyclic group G[2] of prime order with a generator g. i.e., Assume that each of Prv input values is a group element in G, or we can assume a deterministic map that maps each input value c_i to a group element in G. Since any PHE allows Vrf to re-randomize received ciphertexts by multiplying the encryption of the unit element $u \in G$ (under Prv's public key), this computation gives a new ciphertext of the same plaintext without learning/requiring anything about the plaintext.

Finally, Vrf chooses a random permutation π from $\mathcal{P}_n$, the set of all permutations of length n, and shuffles the re-randomized ciphertexts with π, as if it “switches or not" in the two billiard balls problem. Once it receives a puzzle, Prv decrypts each ciphertext with its private key, determines the permutation π' that shuffles original elements to received elements, and forwards π' to Vrf. Vrf accepts if $\pi' = \pi$.

There is a probability that Prv can solve the puzzle without having all distinct elements. In the worst case, when Prv uses only one duplicate, Prv can correctly solve the puzzle with 50% probability. To make the cheating probability low, the puzzle should be repeated λ times, such that $1/2^\lambda$ becomes negligible.

Since each puzzle is independent, Vrf can hash the puzzles (using a suitable cryptographic hash function H) and check this hash value, instead of repeating this three-message exchange multiple times for each puzzle. This reduces the number of communication rounds and associated delays. Figure 4.1 presents the PoED-puzzle protocol described above.

[2]We sometimes denote G as a subgroup of $\mathbb{Z}_p^*$ whose prime order is known, which will be explicitly indicated in such case.

Table 4.1: Cost Analysis of the PoED-puzzle Protocol

Computation Cost			
Operation \ Entity	Prv		Vrf
	Offline	Online	Online
Encryption	n	0	λn
Decryption	0	λn	0
Modular multiplication	0	0	λn
Random permutations (of length n)	0	0	λ
Cryptographic hash (input length)	0	λn	λn
Equality check	0	0	1

Group	**Communication Cost**
C	$(\lambda + 1)n$
$\{0,1\}^\kappa$	1

4.3.2 Analysis of PoED-puzzle Protocol

Theorem 4.1. *Assuming an IND-secure PHE scheme (Enc, Dec) over a cyclic group G, a secure cryptographic hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\kappa$, and the statistical security parameter λ, the PoED-puzzle protocol described in Section 4.3.1 is a secure PoED protocol.*

(*Sketch Proof*) *Completeness* is straightforward because only one correct permutation π exists for distinct elements, and honest Prv can easily determine π after decrypting the ciphertexts. For the *knowledge extractability*, the private key of the underlying encryption scheme can be seen as the witness. Suppose Prv does not know the private key. Then, by the IND and homomorphic property, re-randomized and shuffled ciphertexts from Vrf are indistinguishable from random strings in the ciphertext space. Furthermore, after decryption, the probability of having duplicates and solving the puzzle correctly is *at most* $2^{-\lambda}$ which is set to be negligible by the security parameter λ. Lastly, *zero-knowledgeness* naturally follows from the IND property, since all Vrf can observe are the ciphertexts encrypted by an IND secure PHE. □

Table 4.1 summarizes the computation and communication complexities of the PoED-puzzle protocol with λ puzzles. C is denoted by the ciphertext space of *Enc* and H generates a κ-bit hash result. Overall, both complexities are $O(\lambda n)$, where n is the Prv input size.

4.4 PSI with Element Distinctness Check

Using PoED as a building block, we propose a new variant of PSI that requires all the input elements to be distinct, which we call *All-Distinct Private Set Intersection (AD-PSI)*.

4.4.1 Adversary Model

Among the two parties participating in the computation, *Client* and *Server*, we consider *Server* to be HbC while *Client* can be malicious. This assumption is reasonable in real-life scenarios because *Server* is the one who provides the service to *Client*, and multiple barriers (e.g., law/regulation, security systems for their data, and loss of trust deriving loss of customers) exist for them to be malicious. However, *Client* typically maintains less secure systems and much less data than *Server*, so they may be eager to learn more from *Server*'s large dataset. Note that we consider a stronger guarantee than normal malicious security in PSI literature, as now we aim to enforce the input correctness for *Client*.

4.4.2 Definition of AD-PSI

We define AD-PSI directly instead of defining PSI and adding features. We follow the definitions of client and server privacy in related work [145, 146, 153, 164]. Let $View^{\Pi}_{\mathcal{A}^*}(\mathcal{C}, \mathcal{S}, \lambda)$ denotes a random variable representing the view of adversary $\mathcal{A}^*$ (acting as either *Client* or *Server*) during an execution of Π on inputs $\mathcal{C}$ and $\mathcal{S}$ and the security parameter λ.

Definition 4.1 (All-Distinct Private Set Intersection (AD-PSI)). *consists of two algorithms: {Setup, Interaction}, where:*

- *Setup: an algorithm selecting global/public parameters;*
- *Interaction: a protocol between Client and Server on respective inputs: a multiset*

$\mathcal{C} = [c_1, ..., c_n]$ and a set $\mathcal{S} = \{s_1, ..., s_m\}$, resulting in Client obtaining the intersection of the two inputs;

An AD-PSI scheme satisfies the following properties:

- *Correctness: At the end of Interaction, Client outputs the exact intersection of two inputs only when the elements in $\mathcal{C}$ are distinct. It outputs $\perp$, o.w.*
- *Server Privacy: For every PPT adversary $\mathcal{A}^*$ acting as Client, we say that a AD-PSI scheme Π guarantees the server privacy if there exists a PPT algorithm P_C such that*

$$\{P_C(\mathcal{C}, \mathcal{C} \cap \mathcal{S}, \lambda)\}_{(\mathcal{C},\mathcal{S},\lambda)} \overset{c}{\approx} \{View^{\Pi}_{\mathcal{A}^*}(\mathcal{C}, \mathcal{S}, \lambda)\}_{(\mathcal{C},\mathcal{S},\lambda)}$$

  i.e., on each possible pair of inputs $(\mathcal{C}, \mathcal{S}, \lambda)$, Client's view can be efficiently simulated by P_C on input $(\mathcal{C}, \mathcal{C} \cap \mathcal{S}, \lambda)$.
- *Client Privacy: For every PPT adversary $\mathcal{A}^*$ acting as Server, we say that a AD-PSI scheme Π guarantees the client privacy if there exists a PPT algorithm P_S such that*

$$\{P_S(\mathcal{S}, \lambda)\}_{(\mathcal{C},\mathcal{S},\lambda)} \overset{c}{\approx} \{View^{\Pi}_{\mathcal{A}^*}(\mathcal{C}, \mathcal{S}, \lambda)\}_{(\mathcal{C},\mathcal{S},\lambda)}$$

  i.e., on each possible pair of inputs $(\mathcal{C}, \mathcal{S}, \lambda)$, Server's view can be efficiently simulated by P_S on input $(\mathcal{S}, \lambda)$.

We note that the security definition above is equivalent to the generic “real-vs-ideal" world simulation definition in the semi-honest model, as shown in [153], with the ideal functionality $\mathcal{F}$ below:

1. Wait for an input multiset $\mathcal{C} = [c_1, .., c_n]$ from *Client*.
2. Wait for an input set $\mathcal{S} = \{s_1, ..., s_m\}$ from *Server*.
3. Give output $(|\mathcal{S}|, \mathcal{C} \cap \mathcal{S})$ to *Client* if $\mathcal{C}$ includes all distinct elements, or $(|\mathcal{S}|)$, otherwise.
4. Give output $(|\mathcal{C}|)$ to *Server*.

Figure 4.2: Ideal Functionality $\mathcal{F}$ for AD-PSI

According to the definition above, we propose a construction using PoED-puzzle protocol, so-called *AD-PSI-puzzle*, in the following section.

4.4.3 A Construction for AD-PSI based on PoED-puzzle

AD-PSI-puzzle protocol starts with the PoED-puzzle protocol, i.e., *Client* encrypts each input element in G (or the mapped values for each input element to G with a public map) under a PHE and sends the ciphertexts to *Server*, and *Server* generates a secret key *key*, derived from multiple puzzles that shuffle re-randomized ciphertexts with random permutations. The underlying PHE scheme must now be multiplicatively homomorphic (instead of any PHE) for the correct computation below.

AD-PSI based on PoED-puzzle (AD-PSI-puzzle)

```
Public: $(p, g, h, G)$ where $G = \langle g \rangle$, a subgroup of $\mathbb{Z}_p^*$ of prime order $q$,
        $\lambda$ : statistical secur ity parameter, $pk$ : $Prv$'s public key,
Private: $sk$ : $Prv$'s secret key correlated to $pk$
Client $(\mathcal{C} = [c_1, ..., c_n])$                         Server $(\mathcal{S} = \{s_1, ..., s_m\})$
for $i = 1, ..., n$ :
  $e_i := Enc_{pk}(c_i)$        $\xrightarrow{(e_1, ..., e_n)}$        for $k = 1, ..., \lambda$ :
                                                                  $\pi_k \in_R \mathcal{P}_n$
                                                                  for $i = 1, ..., n$ :
                                                                    $e_{i,k} := e_i \cdot Enc_{pk}(1)$
                                                                  $E_k := \pi_k(e_{1,k}, ..., e_{n,k})$
                                                                $key \leftarrow H(\pi_1, ..., \pi_\lambda)$
                                                                $R \in_R \mathbb{Z}_p^*$
                                                                for $i = 1, ..., n$ :
                                                                  $\hat{e_i} := e_i^R (= e_i \cdot ... \cdot e_i)$
                                                                for $j = 1, ..., m$ :
for $k = 1, ..., \lambda$ :   $\xleftarrow[(t_1, ..., t_m)]{E_1, ..., E_\lambda, (\hat{e_1}, ..., \hat{e_n}),}$    $t_j := SEnc(key, H'(s_j^R))$
  Determine $\pi'_k$ s.t.
  $Dec_{sk}(E_k) = \pi'_k(\mathcal{C})$
$key' \leftarrow H(\pi'_1, ..., \pi'_\lambda)$
for $j = 1, ..., m$ : $t'_j := SD\,ec(key', t_j)$
for $i = 1, ..., n$ : $d_i := H'(\ Dec_{sk}(\hat{e_i}))$
return $\{c_i \in \mathcal{C} \mid d_i \in \{t'_1,\ ..., t'_m\}\}$
```

Figure 4.3: AD-PSI-puzzle Protocol

For computing the intersection without revealing the other elements, *Server* hides *Client*'s ciphertexts and its own input values with the same random element $R \in \mathbb{Z}_p^*$. i.e., *Server* first homomorphically exponentiates *Client* elements with R by e_i^R, which is defined by R homomorphic operations, for each $e_i = Enc(c_i), \forall i$ (by multiplying e_i R times or directly exponentiating R[3]). For its input values, *Server* computes s_j^R for each $s_j \in \mathcal{S}$ so that if some c_i and s_j match, then the randomized c_i^R and s_j^R can also be matched. Then, it hashes each s_j^R using a cryptographic hash function H' and encrypts them under a symmetric encryption scheme with a key $key \in \{0,1\}^\kappa$, i.e., $t_j := SEnc(key, H'(s_j^R))$. Thus, unless *Client* can derive the correct key, it cannot decrypt/learn any information about *Server* elements.

When receiving the messages from *Server*, *Client* first derives the symmetric key key' by solving all the puzzles, as in PoED-puzzle. Then, using the derived key, *Client* decrypt t_j's, obtains $\{t'_j := H'(s_j^R)\}_j$, and compares them with $d'_i s$, the hash values of the decryption of re-randomized ciphertexts, i.e., $d_i := H'(c_i^R)$ for all i. Finally, *Client* outputs all c_i's such that d_i matches for some t_j.

The protocol described above is depicted in Figure 4.3.

Theorem 4.2. *Assuming the hardness of the decisional Diffie-Hellman problem, the protocol described in Figure 4.3 is a secure AD-PSI scheme, satisfying the Definition 4.1 in ROM.*

Proof. **Correctness:** For an honest *Client* with distinct input elements, only one permutation π_k exists such that $\pi_k(\mathcal{C}) = Dec(E_k)$. This is because the decryption results remain the same after the re-randomization due to the homomorphic property of the ElGamal scheme on multiplication. Thus, honest *Client* derives the same permutations as the ones *Server* used, and the derived key', the hash of these permutations, is equal to key. *Client* gets the *Server*'s tags, $\{t'_j = H'(s_j^R)\}_j$ by symmetric-decrypting each of them. Since

[3]Usually, exponentiation of the underlying plaintext can be done more efficiently than multiplying the ciphertext R times. For example, in ElGamal, encryption of x is $Enc(x) = (g^r, xh^r)$ for some random r, and exponentiating to c can be done either $Enc(x) \cdot ... \cdot Enc(x) = (g^R, x^c h^R) = Enc(x^c)$ or $Enc(x)^c = (g^{cr}, x^c h^{cr}) = Enc(x^c)$.

$d_i = H'(Dec(\hat{e}_i)) = H'(Dec(e_i^R)) = H'(c_i^R)$, with overwhelming probability (due to the collision resistance of the cryptographic hash functions), we have $t'_j = d_i \Leftrightarrow s_j^R = c_i^R \Leftrightarrow s_j = c_i$. Therefore, *Client* obtains correct intersection $\{c_i\}_{i \in I}$, with $I := \{i \mid d_i \in \{t'_1, ..., t'_w\}\}$ with distinct input elements.

On the other hand, we show that clients with duplicated elements in their input cannot obtain the intersection with overwhelming probability. Let's look at the case where a corrupted *Client* has the highest probability of successfully cheating, i.e., with $\mathcal{C} = [c_1, ..., c_n]$ with $(n-1)$ distinct items and one duplicate. Without loss of generality, let's say $c_1 = c_2$, and the others are all distinct. In this case, the probability that *Client* obtains the intersection is the same as that of *Client* guesses λ correct permutations, so $2^{-\lambda}$, which is negligible with a sufficiently large λ.

Client Privacy: Assume that *Server* is corrupted. Showing the client privacy is relatively easy: it only sends to *Server* the encryption of the element in its set. Assuming two input sets with the same sizes, if the adversary corrupting *Server* can distinguish whether *Client* used which set as an input, then it can be used for IND-CPA of the ElGamal encryption system. Since it is well-known that the ElGamal encryption system is semantically secure [137] assuming the hardness of the decisional Diffie-Hellman problem, which is reduced to DLP, the adversary cannot distinguish which set is used as well as learn anything about the *Client*'s set elements.

Server Privacy: Assume that *Client* is corrupted, denoted by *Client**. To claim server privacy, we need to show that the *Client*'s view can be efficiently simulated by a PPT algorithm $\mathtt{Sim}_{\mathcal{C}}$. The simulator $\mathtt{Sim}_{\mathcal{C}}$ can be constructed as follows:

1. $\mathtt{Sim}_{\mathcal{C}}$ builds two tables $T = ((\pi_1, ..., \pi_\lambda), k)$ and $T' = (m, h')$ to answer the H and H' queries, respectively.

2. After getting the message $(\mathbb{G}, p, g, h)$ and $\{e_i\}_{i=1}^n$ of a corrupted real-world client $Client^*$, $\mathtt{Sim}_{\mathcal{C}}$ picks λ random permutations from $\mathcal{P}_n$ and n random numbers $r_{i,j}$ from $\mathbb{Z}$ where $i = 1, ..., n$ for each $j = 1, ..., \lambda$. Then, $\mathtt{Sim}_{\mathcal{C}}$ re-randomizes and shuffles $\{e_i\}_{i=1}^n$ by multiplying $(g^{r_{i,j}}, h^{r_{i,j}})$ to each e_i's for $i = 1, ..., n$, say $e_{i,j}$, and applying the permutation π_j to $\{e_{i,j}\}_{i=1}^n$, for each j, say $E_j := \pi_j(e_{1,j}, ..., e_{n,j})$.
3. Also, $\mathtt{Sim}_{\mathcal{C}}$ picks random $R \in \mathbb{Z}$, and exponentiates each component of e_i's, i.e., $\hat{e_i} := e_i^R = (e_{i,1}^R, e_{i,2}^R)$ for $i = 1, ..., n$. $\mathtt{Sim}_{\mathcal{C}}$ also picks m random elements from $\mathcal{M}$, say $u_1, ..., u_m$.
4. $\mathtt{Sim}_{\mathcal{C}}$ encrypts each u_j using $SymE$ with the key, $key := H(\pi_1, ..., \pi_\lambda)$, i.e., $t_j := SymE(key, u_j)$, and replies $\{E_k\}_{k=1}^{\lambda}$, $\{\hat{e_i}\}_{i=1}^n$, $\{t_j\}_{j=1}^m$ to $Client^*$.
5. Then, $\mathtt{Sim}_{\mathcal{C}}$ answers the H, H' queries as follows:
   – For each query $(\pi_1, ..., \pi_\lambda)$ to H, $\mathtt{Sim}_{\mathcal{C}}$ checks if $\exists$ $((\pi_1, ..., \pi_\lambda), key) \in T$ and returns key if so. Otherwise, $\mathtt{Sim}_{\mathcal{C}}$ picks a random $key \in_R \mathcal{K}$ and checks if $\exists((\pi'_1, ..., \pi'_\lambda), key') \in T$ such that $key' = key$. If so, output $\mathtt{fail}_1$ and aborts. Otherwise, it adds $((\pi_1, ..., \pi_\lambda), key)$ to T and returns key to $Client^*$ as $H(\pi_1, ..., \pi_\lambda)$.
   – For each query m to H', $\mathtt{Sim}_{\mathcal{C}}$ checks if $(m, h') \in T'$. If so, $\mathtt{Sim}_{\mathcal{C}}$ returns h'. Otherwise, $\mathtt{Sim}_{\mathcal{C}}$ picks a random $h' \in_R \mathcal{M}$, and checks if $\exists(m'', h'')$ in T' where $h'' = h'$ and $m'' \neq m$. If so, $\mathtt{Sim}_{\mathcal{C}}$ outputs $\mathtt{fail}_2$ and aborts. Otherwise, $\mathtt{Sim}_{\mathcal{C}}$ adds (m, h') to T' and returns h' to $Client^*$ as $H'(m)$.

This finishes the construction $\mathtt{Sim}_{\mathcal{C}}$. The ideal-world server $\overline{Server}$ that interacts with the ideal function f, which answers the queries from $\mathtt{Sim}_{\mathcal{C}}$ as the ideal-world client $\overline{Client}$, gets $\perp$ from f, and the real-world server $Server$ which interacts with $Client^*$ in the real protocol also outputs $\perp$. We now argue that $Client^*$'s view in the interaction with $Server$ and with $\mathtt{Sim}_{\mathcal{C}}$ constructed as above are indistinguishable. The $Client^*$'s view is different only if one of the following happens:

- **$\mathtt{fail}_1$ occurs:** This happens if $\exists(Q' := (\pi'_1, ..., \pi'_\lambda), key')$ such that $key' = key$ but

$Q' \neq Q$ existing in T, for a randomly chosen *key* from $\mathcal{K}$ for the query $Q = (\pi_1, ..., \pi_\lambda)$ to H. This means a collision of H is found, i.e., $H(Q) = H(Q')$ where $Q \neq Q'$. This occurs with negligible probability by the collision resistance of H.

- $\texttt{fail}_2$ **occurs:** This happens if there exists the entry (m'', h'') such that $h'' = h'$ but $m'' \neq m$ existing in T', for a randomly chosen h' from $\mathcal{M}$ for the query m to H'. This means a collision of H' is found, i.e., $H'(m'') = H'(m)$ where $m'' \neq m$. This happens with negligible probability due to the collision resistance of H'.

Since all events above happen with negligible probability, *Client**'s views in the real protocol with the real-world server *Server* can be efficiently simulated by $\mathsf{Sim}_{\mathcal{C}}$ in the ideal world.

□

Though we define AD-PSI such that it does not reveal whether $\mathcal{C}$ satisfies the element distinctness or not to *Server*, this one-bit information may be favored by *Server* to save its computing resources. In the following section, we discuss this alternative definition and an idea of modifying the AD-PSI-puzzle protocol.

4.4.4 Alternative AD-PSI and Modified Construction

Checking the distinctness of $\mathcal{C}$ before proceeding to the next steps may be preferable by *Server* with a large set $\mathcal{S}$ because the rest computation cost is linear to $|\mathcal{S}|$. Whereas, *Client* may be reluctant as it reveals whether *Client* used all distinct elements to *Server*, i.e., a trade-off between client privacy and server efficiency. For this alternative design, AD-PSI *Correctness* can be defined with *Server* outputs $(|\mathcal{C}|, b)$ in Definition 4.1 instead, where b is a boolean result of whether $\mathcal{C}$ satisfies the element distinctness. Likewise, $\mathcal{F}$ is modified as below:

To meet this definition, the AD-PSI-puzzle protocol (in Figure 4.3) can be modified as in

1. Wait for an input multiset $\mathcal{C} = [c_1, .., c_n]$ from *Client*. Abort if $\mathcal{C}$ includes any duplicates.
2. Wait for an input set $\mathcal{S} = \{s_1, ..., s_m\}$ from *Server*.
3. Give output $(|\mathcal{S}|, \mathcal{C} \cap \mathcal{S})$ to *Client*.
4. Give output $(|\mathcal{C}|)$ to *Server*.

Figure 4.4: Ideal Functionality $\mathcal{F}$ for Alternative AD-PSI

Figure 4.5. i.e., Before the intersection computation phase, *Server* first sends all the puzzles to *Client* and proceeds to the next phase only if *Client* corrects all puzzles. Although this modification increases the number of communication rounds, *Server* can save its computation resources for the clients who do not cheat and have enough elements (by size checking) and use this one-bit information in another application (See Section 4.6).

AD-PSI-puzzle Alternative

```
$Client\ (\mathcal{C} = [c_1, ..., c_n])$                                          $Server\ (\mathcal{S} = \{s_1, ..., s_m\})$
for $i = 1, ..., n$ :
  $e_i := Enc_{pk}(c_i)$          $\xrightarrow{(e_1, ..., e_n)}$          for $k = 1, ..., \lambda$ :
                                                                $\pi_k \in_R \mathcal{P}_n$
                                                                for $i = 1, ..., n$ :
                                                                  $e_{i,k} := e_i \cdot Enc_{pk}(1)$
for $k = 1, ..., \lambda$ :          $\xleftarrow{E_1, ..., E_\lambda}$          $E_k := \pi_k(e_{1,k}, ..., e_{n,k})$
  $\pi'_k$ s.t. $Dec_{sk}(E_k) = \pi'_k(\mathcal{C})$                    $key \leftarrow H(\pi_1, ..., \pi_\lambda)$
$key' \leftarrow H(\pi'_1, ..., \pi'_\lambda)$          $\xrightarrow{key'}$          Abort if $key' \neq key$
                                                              $R \in_R \mathbb{Z}_p^*$
                                                              $\hat{e_i} := e_i^R, \forall i \in [1, n]$
for $i = 1, ..., n$ :          $\xleftarrow{\{\hat{e_i}\}_i, \{t_j\}_j,}$          $t_j := H'(s_j^R), \forall j \in [1, m]$
  $d_i := H'(Dec(\hat{e_i}))$
return $\{c_i \in \mathcal{C} \mid d_i \in \{t_1, ..., t_m\}\}$
```

Figure 4.5: Alternative AD-PSI Protocol

Table 4.2 summarizes the computation and communication complexities of the AD-PSI-puzzle protocols with λ puzzles. We denote the cost of the alternative protocol in parentheses only when it has a different cost from the original one. HE denotes the partial homomorphic encryption scheme, and SE denotes the symmetric encryption scheme used in the protocol(s). C_Π represents the ciphertext space of a scheme Π and cryptographic hash functions H and H' generate a κ-bit and κ'-bit hash result, respectively. Overall, both complexities are

Table 4.2: Cost Analysis of AD-PSI-puzzle Protocols. We present the cost of the alternative protocol in $(\cdot)$ only when it is different from the original cost.

Computation Cost of AD-PSI-puzzle (and its alternative)				
Operation Entity		*Client*		*Server*
		Offline	Online	Online
HE.Encryption		n	0	λn
HE.Decryption		0	$(\lambda+1)n$	0
Modular Multiplication		0	0	$(\lambda+R)n$
Random number generation (in $\mathbb{Z}_p^*$)		0	0	1
Random permutations (of length n)		0	0	λ
Cryptographic hash	of input length λn	0	1	1
	of input length $\lvert\mathcal{M}\rvert$	0	n	m
Equality check		0	0	0 (1)
Involvement check (i.e., if $a \in A$)		0	n	0
SE.Encryption		0	0	m (0)
SE.Decryption		0	m (0)	0

Group	Communication Cost
C_{HE}	$(\lambda+2)n$
C_{SE}	m
$\{0,1\}^{\kappa}$	0 (1)
$\{0,1\}^{\kappa'}$	0 (m)
#(rounds)	1 (2)

$O(\lambda n + m)$, where n is the *Client* input size (including duplicates, if any) and m is the *Server* input size.

4.5 AD-PSI Variants

As mentioned in Sections 4.1 and 4.2.4, duplication can be more problematic in PSI variants that give additional/restricted information. In this section, we further discuss how duplication can leak more information and propose a solution for each variant using AD-PSI. Although the solutions are simple, we provide the figures for each protocol for better presentation.

Note that we follow the convention in PSI literature and do not consider the information leakage after multiple executions, which will naturally reveal more than the one they are supposed to disclose in a single execution. For example, when *Client* deliberately adjusts its input elements to PSI-X and the protocol outputs 'No' in the previous rounds and 'Yes' in the next round, then *Client* learns that the exact element added in the last round is in the *Server* set. Though this is interesting, we consider it as a future work.

4.5.1 PSI-CA with Element Distinctness (AD-PSI-CA)

Recall that PSI-CA outputs only the cardinality of the intersection set. Suppose *Client* uses a single element as input to PSI-CA. In that case, although it is not malicious behavior, *Client* can learn if that exact element is in $\mathcal{S}$, which is more information than it is supposed to know. Furthermore, repeating PSI-CA with different single elements can eventually learn the intersection set or the entire $\mathcal{S}$ if the message space is small enough. To prevent this, *Server* may want to restrict the minimum input set size as l and check if $|\mathcal{C}| > l$ during the computation phase.

However, *Client* still can bypass this simple check by duplicating a single element n times where n is greater than l. Although *Server* does not abort as the *Client* set size n is larger than l, the PSI-CA result with this input will be either '0' or '1', which reveals if the single element is in $\mathcal{S}$ or not. Thus, the simple size check is not enough, and *Server* needs a way to check the element distinctness of $\mathcal{C}$, which we call *AD-PSI-Cardinality (AD-PSI-CA)*.

The definition of AD-PSI-CA is similar to the one of AD-PSI, except that $(|\mathcal{S}|, |\mathcal{S} \cap \mathcal{C}|)$ is the *Client* output for correctness, and what the ideal functionality gives to *Client* as output. This feature can be added by modifying the AD-PSI-puzzle protocol as follows: *Server* additionally chooses a random permutation π and sends the permuted ciphertexts $\hat{e_i} := e^R_{\pi(i)}$ instead of $\hat{e_i} := e^R_i$. Since the ciphertexts are randomized with R by *Server*, and *Client* does not know π, now *Client* cannot match the d_i's to the original c_i's. Furthermore, AD-PSI-puzzle guarantees that *Client* cannot solve the puzzle correctly with overwhelming probability when using duplicated inputs. Therefore, *Client* learns $|\mathcal{C} \cap \mathcal{S}|$, only when it uses all distinct input elements.

```
AD-PSI-CA

Public: $(p, g, h, G)$ whe re $G = \langle g \rangle$, a subgroup of $\mathbb{Z}_p^*$ of order $q$,
    $\lambda :$ statistical sec urity parameter, $pk : Prv$'s public key,
Private: $sk : Prv$'s secret key correlated to $pk$
Client $(\mathcal{C} = [c_1, ..., c_n]\ )$                              Server $(\mathcal{S} = \{s_1, ..., s_m\})$
for $i = 1, ..., n :$
  $e_i := Enc_{pk}(c_i)$        $\xrightarrow{(e_1, ..., e_n)}$        for $k = 1, ..., \lambda :$
                                                                  $\pi_k \in_R \mathcal{P}_n$
                                                                  for $i = 1, ..., n :$
                                                                    $e_{i,k} := e_i \cdot Enc_{pk}(u)$
                                                                  $E_k := \pi_k(e_{1,k}, ..., e_{n,k})$
                                                                $key \leftarrow H(\pi_1, ..., \pi_\lambda)$
                                                                $R \in_R \mathbb{Z}_p^*, \pi \in_R \mathcal{P}_n$
                                                                for $i = 1, ..., n :$
                                                                  $\hat{e_i} := e_{\pi(i)}^R$
                                                                for $j = 1, ..., m :$
for $k = 1, ..., \lambda :$    $\xleftarrow[(t_1, ..., t_m)]{E_1, ..., E_\lambda, (\hat{e_1}, ..., \hat{e_n}),}$        $t_j := SEnc(key, H'(s_j^R))$
  Determine $\pi'_k$ s.t.
  $Dec_{sk}(E_k) = \pi'_k(\mathcal{C})$
$key' \leftarrow H(\pi'_1, ..., \pi'_\lambda)$
for $j = 1, ..., m : t'_j :=\ SDec(key', t_j)$
for $i = 1, ..., n : d_i :=\ H'(Dec_{sk}(\hat{e_i}))$
return $|d_i \mid d_i \in \{t'_1, ...\ , t'_m\}|$
```

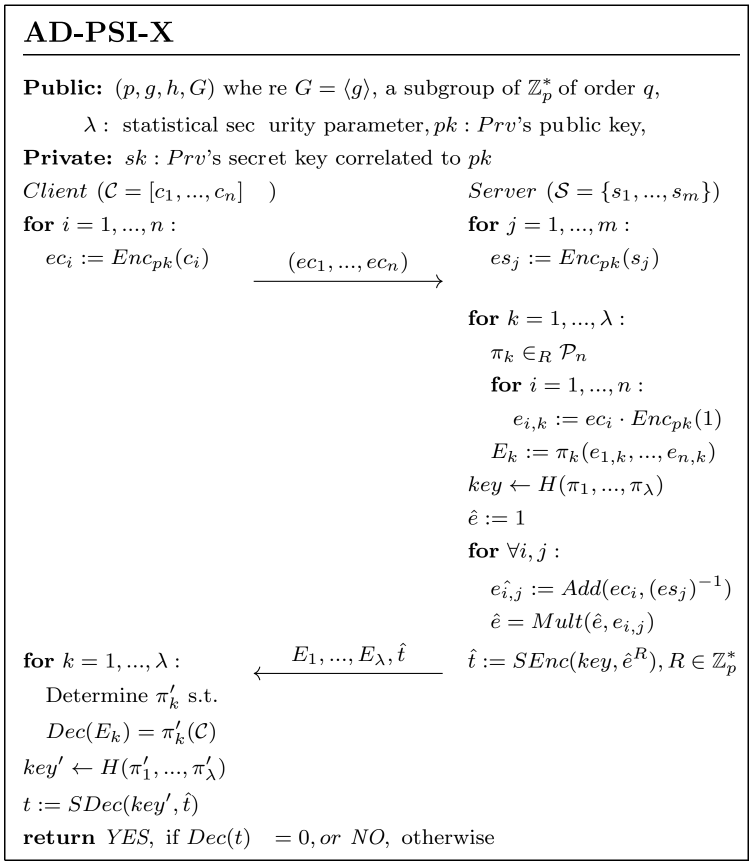
Figure 4.6: AD-PSI-Cardinality (AD-PSI-CA) Protocol

4.5.2 PSI-X with Element Distinctness (AD-PSI-X)

PSI-X outputs minimal information, only the boolean result of whether the intersection of two private input sets is non-empty. Likewise, although *Server* decides on a lower-bound restriction on the size of $\mathcal{C}$, *Client* can obtain more information than the boolean result by using a small input set because if the result is '1' (i.e., intersection exists), each element is in $\mathcal{S}$ with the probability of $1/|\mathcal{C}|$. *Server*, thus, may have more motivation to restrict the size of $\mathcal{C}$ to reduce this probability.

One way to construct a AD-PSI-Existence (AD-PSI-X) protocol is to add our PoED phase to the FHE-based PSI-X protocol. The basic idea of the FHE-based PSI-X protocol is to encrypt each element under an FHE over G that satisfies $Add(Enc(a), Enc(b)) = Enc(a+b)$

and $Mult(Enc(a), Enc(b)) = Enc(a * b)$. Then, compute the subtraction of every pair of $\mathcal{C}$ and $\mathcal{S}$, and multiply all subtractions (with a random number) so that the decryption result can be zero if any of the pairs match. i.e., It computes the encryption of $R \cdot \Pi_{i,j}(c_i - s_j)$ for a random R, which becomes the encryption of zero if any pair of c_i and s_j matches. The recent benchmark [157] on FHE libraries shows that the addition can be done within 100 ms while multiplication requires about 1 second over the integer encoding in many libraries, such as Lattigo [24], PALISADE [22], SEAL [191], and TFHE [178]. The PoED phase can be easily added: *Server* can add the shuffling phase before the PSI-X steps and just encrypt the final message with the key derived from the puzzles as in the PoED-puzzle protocol.



AD-PSI-X

Public: (p, g, h, G) whe re $G = \langle g \rangle$, a subgroup of $\mathbb{Z}_p^*$ of order q,
λ : statistical sec urity parameter, pk : Prv's public key,
Private: sk : Prv's secret key correlated to pk

```
Client ($\mathcal{C} = [c_1, ..., c_n]$)                              Server ($\mathcal{S} = \{s_1, ..., s_m\}$)
for $i = 1, ..., n$ :                                                  for $j = 1, ..., m$ :
  $ec_i := Enc_{pk}(c_i)$     $\xrightarrow{(ec_1, ..., ec_n)}$          $es_j := Enc_{pk}(s_j)$

                                                                       for $k = 1, ..., \lambda$ :
                                                                         $\pi_k \in_R \mathcal{P}_n$
                                                                         for $i = 1, ..., n$ :
                                                                           $e_{i,k} := ec_i \cdot Enc_{pk}(1)$
                                                                         $E_k := \pi_k(e_{1,k}, ..., e_{n,k})$
                                                                       $key \leftarrow H(\pi_1, ..., \pi_\lambda)$
                                                                       $\hat{e} := 1$
                                                                       for $\forall i, j$ :
                                                                         $\hat{e_{i,j}} := Add(ec_i, (es_j)^{-1})$
                                                                         $\hat{e} = Mult(\hat{e}, e_{i,j})$
for $k = 1, ..., \lambda$ :   $\xleftarrow{E_1, ..., E_\lambda, \hat{t}}$   $\hat{t} := SEnc(key, \hat{e}^R), R \in \mathbb{Z}_p^*$
  Determine $\pi'_k$ s.t.
  $Dec(E_k) = \pi'_k(\mathcal{C})$
$key' \leftarrow H(\pi'_1, ..., \pi'_\lambda)$
$t := SDec(key', \hat{t})$
return $YES$, if $Dec(t) = 0$, or $NO$, otherwise
```

Figure 4.7: AD-PSI-Existence (AD-PSI-X) Protocol

4.5.3 PSI-DT with Element Distinctness (AD-PSI-DT)

PSI-DT transfers additional data associated with the intersecting elements. Since this gives more data other than the intersection, when *Server* restricts the *Client* input size, *Client* without enough elements may have more motivation to cheat and bypass the restriction to obtain them. AD-PSI with data transfer (AD-PSI-DT) is defined similarly to AD-PSI, except it outputs $(|\mathcal{S}|, I := \mathcal{S} \cap \mathcal{C}, \{D_j\}_{s_j \in I})$ for *Client*.

An AD-PSI-DT protocol can be constructed as follows: It is the same as AD-PSI-puzzle protocol until randomizing *Client* ciphertexts. Then, for *Server* input elements, *Server* computes one more hash (or a one-way function) H'' that maps k-bit messages to k-bit messages, and encrypts them under the key derived from the puzzles, i.e., $t_j := SEnc(key, H''(s'_j))$, where $s'_j := H'(s_j^R)$. For the associated data to transfer, *Server* encrypts each D_j using the pre-image of H'', i.e., $D'_j := SEnc(s'_j, D_j)$, and sends them along with the other messages. This prevents *Client* from trying all decryption results as key to decrypt the associated data.

Receiving the messages from *Server*, *Client* performs the same steps to learn the intersection as AD-PSI. To obtain the associated data, *Client* uses the matching d_i's for its own (randomized) values to decrypt and get the data. Security for the non-intersecting elements follows the security of AD-PSI, and the one-way property of H'' and the security of the underlying symmetric encryption scheme guarantee the security of the associated data.

4.6 Completing Bounded-Size-Hiding-PSI

As mentioned in Section 4.2.3, *Bounded-Size-Hiding-PSI* was introduced in [66], extending the concept of *Size-Hiding-PSI (SH-PSI)* from [42] by adding an upper bound on the size of *Client* input set $\mathcal{C}$, $|\mathcal{C}|$. For clarification, we denote this primitive by *Upper-bounded-SH-PSI (U-SH-PSI)*. Now we propose a *Bounded-Size-Hiding-PSI (B-SH-PSI) protocol* with

AD-PSI-DT

```
Public: $(p, g, h, G)$ whe re $G = \langle g \rangle$, a subgroup of $\mathbb{Z}_p^*$ of ord er $q$,
    $\lambda$ : statistical security parameter, $pk$ : $Prv$'s public ke y,
Private: $sk$ : $Prv$'s secr et key correlated to $pk$,
    $\mathcal{C} = (c_1, ..., c_n), \mathcal{S} = \{(s_1, D_1), ..., (s_m, D_m)\}$

Client ($\mathcal{C}$)                                   Server ($\mathcal{S}$)
for $i = 1, ..., n$ :
  $e_i := Enc_{pk}(c_i)$      $\xrightarrow{(e_1, ..., e_n)}$
                                                          for $k = 1, ..., \lambda$ :
                                                            $\pi_k \in_R \mathcal{P}_n$
                                                            for $i = 1, ..., n$ :
                                                              $e_{i,k} := e_i \cdot Enc_{pk}(1)$
                                                            $E_k := \pi_k(e_{1,k}, ..., e_{n,k})$
                                                          $key \leftarrow H(\pi_1, ..., \pi_\lambda)$
                                                          $R \in_R \mathbb{Z}_p^*$
                                                          for $i = 1, ..., n$ :
                                                            $\hat{e_i} := e_i^R$
                                                          for $j = 1, ..., m$ :
                                                            $s'_j := H'(s_j^R)$
                                                            $t_j := SEnc(key, H''(s'_j))$
for $k = 1, ..., \lambda$ :   $\xleftarrow[(t_1, ..., t_m), (D'_1, ..., D'_m)]{E_1, ..., E_\lambda, (\hat{e_1}, ..., \hat{e_n}),}$   $D'_j := SEnc(s'_j, D_j)$
  Determine $\pi'_k$ s.t.
  $Dec_{sk}(E_k) = \pi'_k(\mathcal{C})$
$key' \leftarrow H(\pi'_1, ..., \pi'_\lambda)$
for $j = 1, ..., m$ : $t'_j := SDec(key', t_j)$
for $i = 1, ..., n$ : $d_i := H'(Dec_{sk}(\hat{e_i}))$
return $\{(c_i, D_j) \mid c_i \in \mathcal{C}$ such that $H''(d_i) = t'_j$ for some $j \in \{1, ..., m\}$
           and $D_j := SDec(d_i, D'_j)$ for such $j\}$
```

Figure 4.8: AD-PSI-Data Transfer (AD-PSI-DT) Protocol

complete, both lower and upper, bounds on $|\mathcal{C}|$.

In B-SH-PSI, *Server* publishes its restriction rules, L for lower bound and U for upper bound, for $|\mathcal{C}|$. i.e., *Server* wants *Client* to obtain the intersection only when $L \leq |\mathcal{C}| \leq U$. On the other hand, *Client* wants to hide its input size as well as any information about its elements from *Server*. Figure 4.9 shows the ideal functionality $\mathcal{F}_B$ for B-SH-PSI described above.

We construct a B-SH-PSI protocol using U-SH-PSI and the AD-PSI-puzzle protocols as building blocks and briefly present it in Figure 4.10. To enforce that *Client* cannot learn

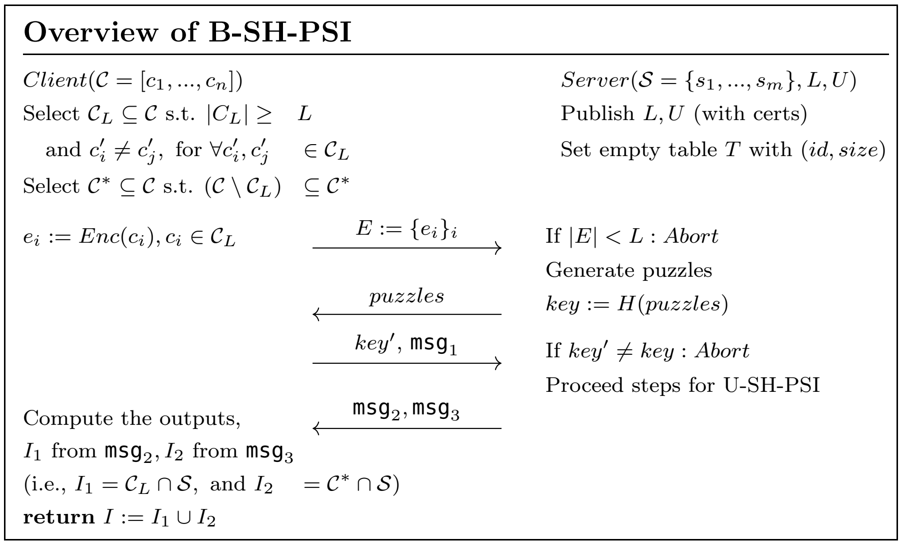
1. Wait for input $\mathcal{C} = [c_1, ..., c_n]$ from *Client*.
2. Wait for input $\mathcal{S} = \{s_1, ..., s_m\}$ from *Server*.
3. Abort if $\mathcal{C}$ does not contain at least L distinct elements, or $|\mathcal{C}| > U$. Give the output $(|\mathcal{S}|, \mathcal{C} \cap \mathcal{S})$ to *Client* only if $L \leq |\mathcal{C}| \leq U$.
4. Give output b to *Server*, where b is the boolean value of whether $|\mathcal{C}| \geq L$.

Figure 4.9: Ideal Functionality $\mathcal{F}_B$ for B-SH-PSI

any information about the intersection without satisfying both upper- and lower-bound requirements, we need the alternative AD-PSI-puzzle protocol (in Section 4.4.4) that reveals the one-bit information if $\mathcal{C}$ satisfies the lower-bound or not.

Recall that *Client* cannot obtain the next message from *Server* with overwhelming probability if $\mathcal{C}_L$ includes duplicates. Also, since *Server* can see the size of $\mathcal{C}_L$ during the AD-PSI phase, it can just abort (or send an error message to *Client*) if $\mathcal{C}_L$ does not satisfy the lower bound L. Otherwise, *Server* stores this size $|\mathcal{C}_L|$ and sends some puzzles for AD-PSI to *Client*. The honest *Client* can enclose the first message (the accumulator for the rest of the elements in $\mathcal{C}$, i.e., $\mathcal{C}^* := \mathcal{C} \setminus \mathcal{C}_L$), msg_1, along with the *key*$'$ derived from the given puzzles. If *key*$'$ is correct, *Server* proceeds to the steps for U-SH-PSI using msg_1 and the upper bound, $U' := (U - |\mathcal{C}_L|)$, or aborts, otherwise. *Client* obtains $I_1 := \mathcal{C}_L \cap \mathcal{S}$ from the response for AD-PSI (denoted by msg_2 in Figure 4.10), and $I_2 := \mathcal{C}^* \cap \mathcal{S}$ from the one for U-SH-PSI (denoted by msg_3 in Figure 4.10), which are combined to the final result, $I := I_1 \cup I_2$.

The security and efficiency of the idea above rely on the ones of underlying AD-PSI and U-SH-PSI protocols. The AD-PSI phase guarantees that $\mathcal{C}$ satisfies the lower bound L. Although there is no duplicate check in the U-SH-PSI phase, *Client* does not have the motivation for duplicating the elements because *Client* can learn the result only when $|\mathcal{C}^*| \leq U'$ (i.e., duplicates limit *Client* more, especially when $|\mathcal{C}|$ is close to U).



Overview of B-SH-PSI

$Client(\mathcal{C} = [c_1, ..., c_n])$ | $Server(\mathcal{S} = \{s_1, ..., s_m\}, L, U)$

Select $\mathcal{C}_L \subseteq \mathcal{C}$ s.t. $|C_L| \geq L$ and $c'_i \neq c'_j$, for $\forall c'_i, c'_j \in \mathcal{C}_L$

Select $\mathcal{C}^* \subseteq \mathcal{C}$ s.t. $(\mathcal{C} \setminus \mathcal{C}_L) \subseteq \mathcal{C}^*$

Publish L, U (with certs)

Set empty table T with $(id, size)$

$e_i := Enc(c_i), c_i \in \mathcal{C}_L$ $\xrightarrow{E := \{e_i\}_i}$ If $|E| < L$: *Abort*

Generate puzzles

$key := H(puzzles)$

$\xleftarrow{puzzles}$

$\xrightarrow{key', \mathtt{msg}_1}$ If $key' \neq key$: *Abort*

Proceed steps for U-SH-PSI

$\xleftarrow{\mathtt{msg}_2, \mathtt{msg}_3}$

Compute the outputs,

I_1 from $\mathtt{msg}_2$, I_2 from $\mathtt{msg}_3$

(i.e., $I_1 = \mathcal{C}_L \cap \mathcal{S}$, and $I_2 = \mathcal{C}^* \cap \mathcal{S}$)

return $I := I_1 \cup I_2$

Figure 4.10: Idea of B-SH-PSI with input bound $[L, U]$. $\mathtt{msg}_1$ and $\mathtt{msg}_3$ denote the first and responding messages for the U-SH-PSI protocol, whereas the others denote the messages for the alternative AD-PSI-puzzle protocol in Figure 4.5

4.7 Authorized PSI with Element Distinctness

So far, we have seen multiple PSI and its variant protocols that check the duplicity of input values. However, as noted in Section 4.1, malicious *Client* can still bypass these duplicity checks by generating random inputs instead of duplicating valid inputs. And what is the meaning of "valid" inputs? To examine if *Client* uses valid inputs, including a trusted third party (TTP) who signs on valid inputs and later audits and punishes any invalid inputs is inevitable. i.e., Authorized PSI (APSI) that not only checks the element distinctness but also the validity of the input values. This section presents two versions of APSI: (v1) *stateful APSI*, where TTP tracks *Client* input values, and (v2) *stateless APSI*, where TTP does not save/track any information about *Client* input values.

4.7.1 AD-APSI Definition

Adopting the definitions of APSI from the related work [105, 106, 281, 299] and referring to the definitions of general two-party computation from [139, 153], secure AD-APSI can be defined as below. Let $REAL^{\Pi}_{\mathcal{A}(z),P}(C, S, \lambda)$ be the output of honest party and the adversary $\mathcal{A}$ corrupting P (either *Client* or *Server*) after a real execution of an AD-APSI protocol Π, where *Client* has input (potentially multi)set $\mathcal{C}$, *Server* has input set $\mathcal{S}$, $\mathcal{A}$ has auxiliary input z, and the security parameter is λ. Let $IDEAL^{\mathcal{F}}_{Sim(z),P}(\mathcal{C}, \mathcal{S}, \lambda)$ be the analogous distribution in an ideal execution with a trusted party who computes the ideal functionality $\mathcal{F}$ defined below.

Definition 4.2 (All-Distinct Authorized PSI (AD-APSI)). *is a tuple of three algorithms: {Setup, Authorize, Interaction}, where*

- *Setup: an algorithm selecting global/public parameters;*
- *Authorize: a protocol between Client and TTP resulting in Client committing to its input, $\mathcal{C} = [c_1, ..., c_n]$, and TTP issuing authorizations, one for each element of $\mathcal{C}$; and*
- *Interaction: a protocol between Client and Server on respective inputs: a (multi)set $\mathcal{C}$ and a set $\mathcal{S}$, resulting in Client obtaining the intersection of two inputs;*

An AD-APSI scheme satisfies the following properties:

- *Correctness: At the end of Interaction, Client outputs the exact intersection of two inputs, only when the elements in $\mathcal{C}$ are all distinct and authorized by TTP. Otherwise, Client outputs $\perp$;*
- *Server Privacy: Client learns no information about the subset of $\mathcal{S}$ that is not in the intersection, except its size. More formally, an AD-APSI scheme securely realizes the server privacy in the presence of malicious adversaries corrupting Client if for every*

real-world adversary $\mathcal{A}$, there exists a simulator Sim such that, for every $\mathcal{C}$, $\mathcal{S}$, and auxiliary input z,

$$\{REAL^{\Pi}_{\mathcal{A}(z),Client}(\mathcal{C},\mathcal{S},\lambda)\}_{\lambda} \overset{c}{\approx} \{IDEAL^{\mathcal{F}}_{Sim(z),Client}(\mathcal{C},\mathcal{S},\lambda)\}_{\lambda}$$

- *Client Privacy: Server learns no information about Client input elements, except its size, authorization status, and element distinctness. More formally, an AD-APSI scheme securely realizes the client privacy in the presence of malicious adversaries corrupting Server if for every real-world adversary $\mathcal{A}$, there exists a simulator Sim such that, for every $\mathcal{C}$, $\mathcal{S}$, and z,*

$$\{REAL^{\Pi}_{\mathcal{A}(z),Server}(\mathcal{C},\mathcal{S},\lambda)\}_{\lambda} \overset{c}{\approx} \{IDEAL^{\mathcal{F}}_{Sim(z),Server}(\mathcal{C},\mathcal{S},\lambda)\}_{\lambda}$$

where the ideal functionality $\mathcal{F}$ is defined as follows:

- `Authorize` *: ($\mathcal{F}$ forwards the messages between Client and TTP and remembers the authorized elements for Client)*
  1. *Wait for an authorization request from Client, requesting TTP to authorize an element c*
  2. *Forward the request to TTP who either accepts or rejects it*
  3. *If TTP accepts, it forwards the messages from TTP to Client and remembers that TTP has authorized c for Client. Otherwise, it replies* `abort` *to Client*
- `Interaction` *: ($\mathcal{F}$ receives input elements from Client and Server and outputs the intersection to Client, only when Client inputs are all distinct and authorized, while giving Client input size and verification result (for authorization and duplication) to Server)*
  1. *Wait for an input (multi)set $\mathcal{C} = [c_1, .., c_n]$ from Client*
  2. *Wait for an input set $\mathcal{S} = \{s_1, ..., s_m\}$ from Server*

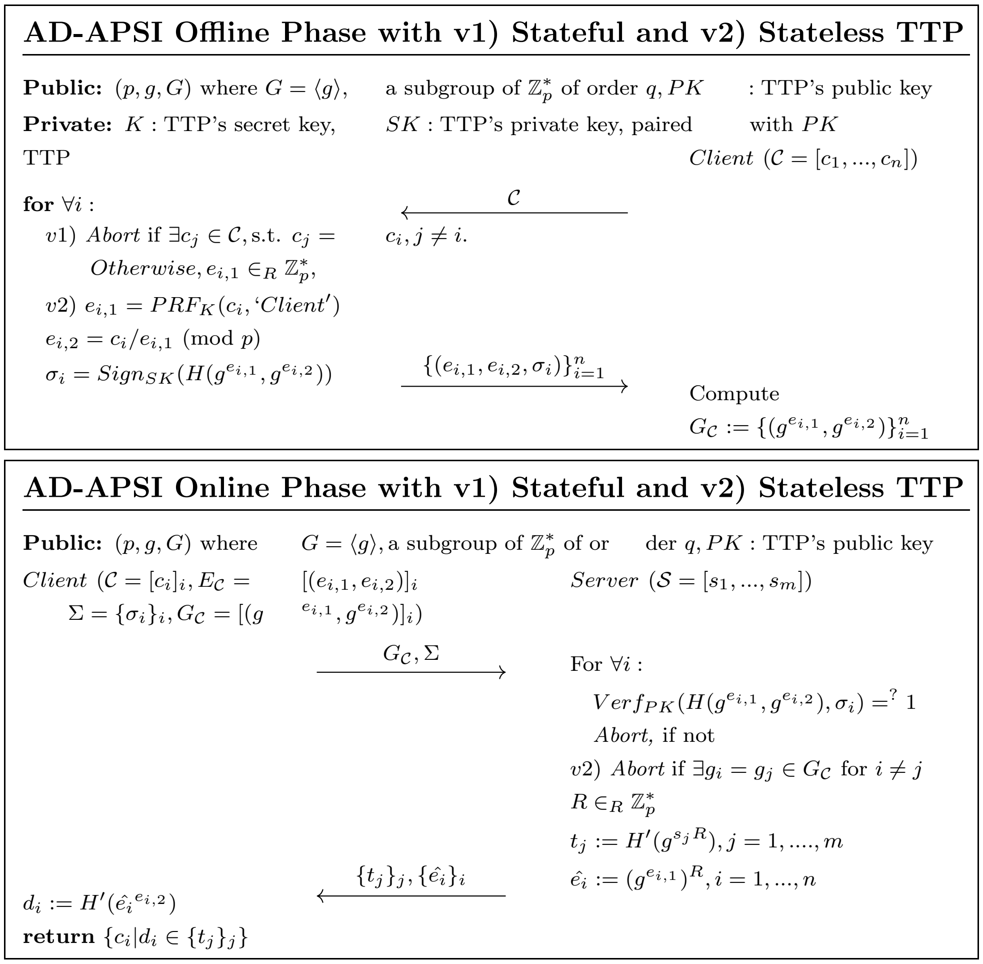
3. While sending $|\mathcal{C}|$ to Server, send **`abort`** *to Client if $\mathcal{C}$ includes (1) any unauthorized element, or (2) duplicated elements. Otherwise, compute the intersection of $\mathcal{C}$ and $\mathcal{S}$ and send $(|\mathcal{S}|, \mathcal{C} \cap \mathcal{S})$ to Client. It also sends b to Server, where b is the result(s) for verifying the existence of (1) (and (2) in stateless version) above with their cardinality(ies).*

For clear notation, we denote the functionalities above as $\mathcal{F}_{\mathsf{Auth}}$ and $\mathcal{F}_{\cap}$.

4.7.2 AD-APSI Construction

The main idea is from the double spending detection in [84]. i.e., TTP first divides each input value into two factors, where these factors are not revealed to anyone except *Client*. For the stateful TTP, the factors can be computed by choosing a random value in $\mathbb{Z}_p^*$ as the first factor and calculating the rest. For the stateless TTP, the first factor is computed so that it is unique per element value, e.g., with a pseudo-random function PRF (under TTP's secret key) for each element in $\mathcal{C}$, and the second factor is calculated by dividing the element with the first factor. Then, the TTP signs a message such that it can be easily re-computed by a third party while not revealing each factor so that anyone with the message can verify the signature with the TTP's public key.

In the online phase, *Client* sends $G_{\mathcal{C}}$, the pre-computed values that effectively hide two factors for each input value, and Σ, all the signatures given by TTP. Then, *Server* first verifies each signature with a newly-computed message with $G_{\mathcal{C}}$ and aborts if any signature verification fails. In the stateless TTP version, *Server* additionally checks if there are any same elements in $G_{\mathcal{C}}$ and aborts if so. If all passed, *Server* now proceeds to the intersection computation phase, similar to the other PSI protocols. i.e., It first chooses a random number R to hide its elements, and computes t_j, which can be also pre-computed. Then, *Server* exponentiates each $g^{e_{i,1}}$ to the same R so that *Client* can compute the same form, compare, and obtain the intersection result. Figure 4.11 shows the aforementioned offline and online phases with stateful and stateless TTP options, with an example form of message, $m_i := H(g^{e_{i,1}}, g^{e_{i,2}})$ for each $c_i = e_{i,1} * e_{i,2} \pmod{p}$ in $\mathcal{C}$. In the offline phase, *Client* can pre-compute $G_{\mathcal{C}}$ once it receives all the factors from TTP, or TTP can also send $G_{\mathcal{C}}$ along with the others, which is the trade-off between communication cost and *Client*'s computation cost.



AD-APSI Offline Phase with v1) Stateful and v2) Stateless TTP

Public: (p, g, G) where $G = \langle g \rangle$, a subgroup of $\mathbb{Z}_p^*$ of order q, PK : TTP's public key
Private: K : TTP's secret key, SK : TTP's private key, paired with PK

TTP | *Client* $(\mathcal{C} = [c_1, ..., c_n])$

$\xleftarrow{\mathcal{C}}$

for $\forall i$:
v1) *Abort* if $\exists c_j \in \mathcal{C}$, s.t. $c_j = c_i, j \neq i$.
Otherwise, $e_{i,1} \in_R \mathbb{Z}_p^*$,
v2) $e_{i,1} = PRF_K(c_i, \text{'}Client\text{'})$
$e_{i,2} = c_i / e_{i,1} \pmod{p}$
$\sigma_i = Sign_{SK}(H(g^{e_{i,1}}, g^{e_{i,2}}))$

$\xrightarrow{\{(e_{i,1}, e_{i,2}, \sigma_i)\}_{i=1}^n}$

Compute
$G_{\mathcal{C}} := \{(g^{e_{i,1}}, g^{e_{i,2}})\}_{i=1}^n$

AD-APSI Online Phase with v1) Stateful and v2) Stateless TTP

Public: (p, g, G) where $G = \langle g \rangle$, a subgroup of $\mathbb{Z}_p^*$ of order q, PK : TTP's public key

Client $(\mathcal{C} = [c_i]_i, E_{\mathcal{C}} = [(e_{i,1}, e_{i,2})]_i$
$\Sigma = \{\sigma_i\}_i, G_{\mathcal{C}} = [(g^{e_{i,1}}, g^{e_{i,2}})]_i)$ | *Server* $(\mathcal{S} = [s_1, ..., s_m])$

$\xrightarrow{G_{\mathcal{C}}, \Sigma}$

For $\forall i$:
$Verf_{PK}(H(g^{e_{i,1}}, g^{e_{i,2}}), \sigma_i) \overset{?}{=} 1$
Abort, if not
v2) *Abort* if $\exists g_i = g_j \in G_{\mathcal{C}}$ for $i \neq j$
$R \in_R \mathbb{Z}_p^*$
$t_j := H'(g^{s_j R}), j = 1, ..., m$
$\hat{e_i} := (g^{e_{i,1}})^R, i = 1, ..., n$

$\xleftarrow{\{t_j\}_j, \{\hat{e_i}\}_i}$

$d_i := H'(\hat{e_i}^{e_{i,2}})$
return $\{c_i | d_i \in \{t_j\}_j\}$

Figure 4.11: All-Distinct Authorized PSI (AD-APSI) scheme.

4.7.3 Security Analysis

Theorem 4.3. *The protocol described in Section 4.7.2 is a secure AD-APSI scheme, satisfying Definition 4.2 in ROM.*

Proof. **Correctness:** For an honest *Client* with all authorized and distinct elements, the stateful TTP generates authentic signatures for each element so that *Server* can verify the signatures correctly. For the stateless TTP, instead of tracking all the input values of *Client*, TTP generates unique and deterministic factors of the input. Thus, *Server* can tell when *Client* uses duplicated elements as the corresponding elements in $G_{\mathcal{C}}$ are the same. When *Server* replies, *Client* outputs the exact intersection of $\mathcal{C}$ and $\mathcal{S}$ because, for $c_i = s_j$, $d_i := H'(\hat{e}_i{}^{e_{i,2}}) = H'((g^{e_{i,1}R})^{e_{i,2}}) = H'(g^{c_i R}) = H'(g^{s_j R}) = t_j$. Therefore, duplicated elements in $\mathcal{C}$ are caught by either the stateful TTP or *Server* (when TTP is stateless), unauthorized (i.e., not signed by TTP) elements are caught by *Server*, and honest *Client* obtains the exact intersection of the two input sets.

For server and client privacy, we show that the distribution of protocol execution in the real world is computationally indistinguishable from the output from interaction with $\mathcal{F}$ in the ideal world, assuming the same corrupted party (either *Client* or *Server*). Since the interaction between *Server* and *Client* is during the online phase for `Interaction`, it is compared with $\mathcal{F}_{\cap}$ (recall Definition 4.2), assuming $\mathcal{C}$ is authorized with $\mathcal{F}_{\mathtt{Auth}}$.

Server Privacy: Assume that *Client* is corrupted, denoted by *Client**. We show that the distribution of *Client** outputs in the real world can be efficiently simulated by a PPT $\mathtt{Sim}_{\mathcal{C}}$ constructed as below.

1. $\mathtt{Sim}_{\mathcal{C}}$ builds two tables $T_1 = ((m_1, m_2), h)$ and $T_2 = (m, h')$ to answer the H and H' queries, respectively.
2. After getting the messages $G_{\mathcal{C}} := \{g_{i,1}, g_{i,2}\}_i$ and Σ of a corrupted real-world client, *Client**, $\mathtt{Sim}_{\mathcal{C}}$ verifies the received signatures with respect to each $H(g_{i,1}, g_{i,2})$ via *Verf* and TTP's public key. If any of those fails, it aborts. (Likewise, for the stateless version, $\mathtt{Sim}_{\mathcal{C}}$ also checks the duplicates in $G_{\mathcal{C}}$ and aborts if any.

3. Otherwise, $\mathtt{Sim}_{\mathcal{C}}$ picks m random elements, $u_1, ..., u_m$, in G and computes $t_j := H'(u_j)$ for $j = 1, ..., m$. It also picks a random R, computes $\{\hat{e}_i = g_{i,1}^R\}_i$, and replies $\{\hat{e}_i\}_i$ and $\{t_j\}_m$ to *Client**.
4. For each query to H and H', $\mathtt{Sim}_{\mathcal{C}}$ answers as follows:
    - For each query (m_1, m_2) to H, $\mathtt{Sim}_{\mathcal{C}}$ checks if $exists((m_1, m_2), h) \in T_1$ and returns h if so. Otherwise, $\mathtt{Sim}_{\mathcal{C}}$ picks a random h (from the same space as other values) and checks if $exists((\tilde{m_1}, \tilde{m_2}), \tilde{h}) \in T_1$ such that $h = \tilde{h}$. If so, output $\mathtt{fail}_1$ and abort. Otherwise, it adds $((m_1, m_2), h)$ to T_1 and returns h to *Client** as $H((m_1, m_2))$.
    - For each query m to H', $\mathtt{Sim}_{\mathcal{C}}$ checks if $exists(m, h') \in T_2$ and returns h' if so. Otherwise, $\mathtt{Sim}_{\mathcal{C}}$ picks a random h' (from the same space as other values) and checks if $exists(\tilde{m}, \tilde{h}) \in T_2$ such that $h' = \tilde{h}$. If so, output $\mathtt{fail}_2$ and abort. Otherwise, it adds (m, h') to T_2 and returns h' to *Client** as $H'(m)$.

This finishes the $\mathtt{Sim}_{\mathcal{C}}$ construction. The *Client**'s view in the interaction with $\mathtt{Sim}_{\mathcal{C}}$ above is different from the view in the real-world interaction with the real server, *Server*, only if $\mathtt{fail}_1$ or $\mathtt{fail}_2$ happen. However, due to the collision resistance property of cryptographic hash functions H, H', they occur with negligible probability. Thus, *Client**'s view when interacting with *Server* can be efficiently simulated by $\mathtt{Sim}_{\mathcal{C}}$ in the ideal world. For the outputs, the ideal-world server $\overline{Server}$ that interacts with $\mathcal{F}_{\cap}$, which answers the queries from $\mathtt{Sim}_{\mathcal{C}}$ as the ideal-world *Client*, $\overline{Client}$, receives $(|\mathcal{C}, b)$ from $\mathcal{F}_{\cap}$. On the other hand, the real-world (honest) server *Server* that interacts with *Client** in the real protocol also outputs (learns) $(|\mathcal{C}, b)$. i.e., $\overline{Server}$ interacting with $\mathtt{Sim}_{\mathcal{C}}$ and *Server* interacting with *Client** yield the identical outputs.

Client Privacy: Similarly, we assume a corrupted server, *Server**, and show that *Server**'s view in the real world can be efficiently simulated by a PPT simulator, $\mathtt{Sim}_{\mathcal{S}}$, constructed as below. Intuitively, $\mathsf{Sim}_{\mathcal{S}}$ sits between $\mathcal{F}_{\cap}$ and $Server^*$, and interacts with both in such a way that $Server^*$ is unable to distinguish protocol runs with $\mathsf{Sim}_{\mathcal{S}}$ from real-world protocol runs with $Client$. First, $\mathsf{Sim}_{\mathcal{S}}$ builds tables T_1 and T_2, and answers similarly to $\mathsf{Sim}_{\mathcal{C}}$ above for H and H' queries. Then, for inputs, since $Client$ and TTP communicate offline before the online phase, the authorized elements for $Client$ are made available to $\mathsf{Sim}_{\mathcal{S}}$. $\mathsf{Sim}_{\mathcal{S}}$ uses a subset of authorized elements during the simulation to emulate $Client$'s behavior. If $Server^*$ does not abort and reply $(\{t_j\}_j, \{\hat{e}_i\}_i)$, $\mathsf{Sim}_{\mathcal{S}}$ checks if $\hat{e}_i^{e_{i,2}} \in \{t_j\}_j$. If so, $\mathsf{Sim}_{\mathcal{S}}$ adds $s_i := e_{i,1}e_{i,2} \pmod{p}$ in $\mathcal{S}$, and otherwise, adds a dummy element in $\mathbb{Z}_p^*$ in $\mathcal{S}$. Then, $\mathsf{Sim}_{\mathcal{S}}$ plays the role of the ideal-world server, $\overline{Server}$, using $\mathcal{S}$ to respond to the queries from the ideal client ($\overline{Client}$). Since $\mathsf{Sim}_{\mathcal{S}}$ uses the authorized inputs, $Server^*$'s view in the interaction with $\mathsf{Sim}_{\mathcal{S}}$ is identical to the view in the interaction with honest $Client$ in the real world. Also, the output of the ideal-world client $\overline{Client}$ that interacts with $\mathcal{F}_{\cap}$, which answers the queries from $\mathsf{Sim}_{\mathcal{S}}$ as the ideal-world $Server$, $\overline{Server}$, is identical to the output of the real-world $Client$ interacting with $Server^*$ as $(|\mathcal{S}, \mathcal{C} \cap \mathcal{S})$, only when all inputs in $\mathcal{C}$ are authorized and distinct. □

4.8 Summary

This chapter suggested checking the input validity in PSI with input size limits. We identified two malicious behaviors to bypass the lower-bound limit: using duplicated and spurious elements. To prevent these, we proposed to prove the element distinctness of private input and applied it to a new PSI variant, AD-PSI, that additionally checks the “set-ness" of private input with potential duplicates. We also discussed AD-PSI variants, AD-PSI-CA, AD-PSI-X, and AD-PSI-DT, where duplicates cause more information leakage without PoED, and proposed a B-SH-PSI scheme with both upper and lower bounds on the client input size. Lastly, we presented AD-APSI that assesses both misbehaviors, involving two types of TTP.

Chapter 5

Communication-Efficient (Proactive) Secure Computation for Dynamic General Adversary Structures and Dynamic Groups

This chapter suggests enhancing system and adversary models in MPC for today's complex large distributed systems. We first explore current MPC protocols secure against adversaries with general corruption capabilities, and extend them to be secure against mobile adversaries, i.e., proactively secure. We then add protocols to handle dynamically changing computation groups and dynamic corruption scenarios, to adapt to the system changes in MTD.

5.1 Introduction

Secure Multiparty Computation (MPC) is a general primitive consisting of several protocols executed among a set of parties, and has motivated the study of different adversary models and various new settings in cryptography [154, 83, 259, 241, 99, 52, 98, 56, 169, 48]. For groups with more than two parties, i.e., the multiparty setting, secret sharing (SS) is often an underlying primitive used in constructing MPC; SS also has other applications in secure distributed systems and protocols used therein [76, 168, 143, 79, 46, 126, 127, 128].

In typical arithmetic MPC, the underlying SS [276, 58] is of the threshold type scheme, i.e., a dealer shares a secret s among n parties such that an adversary that corrupts no more than a threshold t of the parties (called corruption threshold) does not learn anything about s, while any $t+1$ parties can efficiently recover it. MPC protocols built on top of SS allow a set of distrusting parties $P_1, \ldots, P_n$, with private inputs $x_1, \ldots, x_n$, to jointly compute (in a secure distributed manner) a function $f(x_1, x_2, \ldots, x_n)$ while guaranteeing the correctness of its evaluation and the privacy of inputs for honest parties. The study of secure computation was initiated by [301] for two parties and [154] for three or more parties. Constructing efficient MPC protocols withstanding stronger adversaries has been an important problem in cryptography and witnessed significant progress since its inception, e.g., [55, 83, 259, 183, 99, 98, 56, 241, 48, 49].

Enforcing a bound on adversary's corruption limit is often criticized as being arbitrary for protocols with long execution times, especially when considering the so-called “reactive" functionalities that continuously run a control loop. Such reactive functionalities become increasingly important, as MPC is adopted to resiliently implement privacy-sensitive control functions in critical infrastructures such as power grids or command-and-control in distributed network monitoring and defense infrastructure. In those two cases, one should expect resourceful adversaries to continuously attack parties/servers involved in such an MPC, and given enough time, vulnerabilities in underlying software will eventually be found.

An approach to deal with the ability of adversaries to eventually corrupt all parties is the *proactive security model* [241]. This model introduced the notion of a mobile adversary motivated by the persistent corruption of parties in an MPC protocol. A mobile adversary can corrupt all parties in a distributed protocol during the execution of said protocol, but with the following limitations: (i) only a constant fraction (in the threshold setting) of parties can be corrupted during any round of the protocol; (ii) parties periodically get rebooted to a clean initial state, guaranteeing a small fraction of corrupted parties, assuming that the corruption rate is not more than the reboot rate[1]. The model also assumes that an adversary cannot predict or reconstruct the randomness used by parties in any uncorrupted period of time, as demarcated by rebooting.

In most (standard and proactive) MPC literature, the adversary's corruption capability is characterized by a threshold t (out of the n parties). More generally, however, the adversary's corruption capability could be specified by a so-called *general adversary structure (GAS)*, i.e., a set of potentially corruptible subsets of parties. Even more generally, it can be specified by a set of corruption scenarios, one of which the adversary can choose (secretly). For instance, each scenario can specify a set of parties that can be passively corrupted and a subset that can even be actively corrupted. Furthermore, such scenarios may change over time, thus effectively rendering the GAS describing them to itself be dynamically evolving. There are currently no proactive MPC protocols efficiently handling such dynamic general specifications of adversaries, especially when the group of parties performing MPC is dynamic.

Our main objective is to address a setting that is as close as possible to the complex dynamic reality of today's distributed systems. We accomplish this by answering the following question: *Can we design a communication-efficient proactively secure MPC (PMPC) protocol for*

[1]In our model, rebooting to a clean state includes global information, e.g., circuits to be computed via MPC, identities of parties in the computation, and access to secure point-to-point channels and a broadcast channel.

dynamic groups with security against dynamic general adversary structures?

Contributions: We answer the above question in the affirmative. One of our main contributions is to build a set of protocols to efficiently convert back and forth between two different MPC schemes for GAS; this process is often called *share conversion*. Specifically, we consider an MPC scheme based on additive secret sharing and another MPC scheme based on Monotone Span Programs (MSP). The ability to efficiently and securely convert between these MPCs enables us to construct *the first communication-efficient structure-adaptive proactive MPC (PMPC) protocol for dynamic GAS settings.* We note that *all* existing proactive secret sharing and PMPC protocols can only handle (threshold) adversary structures that describe sets of parties with cardinality less than a fraction of the total number of parties.

Given the large number of “moving parts" and complexity of PMPC protocols and the additional complexity for specifying them for GAS, we start from a standard (i.e., non-proactive) MPC protocol with GAS and extend it to the proactive setting for static groups and then dynamic groups. Note that MPC protocols typically extend secret sharing and perform computations on secret shared inputs, we thus focus the discussion in this work on MPC with the understanding that results also apply to secret sharing.

As part of the proactive protocols, we support the following three functionalities: refreshing shares that reconstruct the same secret, recovering shares of parties who lost them or were rebooted from a clear state, and redistributing new shares of the same secret to another group of parties. This implies that we can also deal with dynamic sets of parties, where parties can be eliminated and added (i.e., start with a recovery of their shares in a refresh phase). Also, we can deal with settings where the entire set of parties changes, and existing secret shared data has to be moved to a new set of parties with a possibly new specification of the GAS they should protect against. This original set of parties then redistributes the shared secrets to the new set (which may or may not have some overlap with the original set).

Organization: First, we emphasize why we need secure computation for dynamic groups and GAS. In Section 5.3, we overview the typical blueprint of PSS and PMPC and briefly discuss the roadblocks/challenges facing constructing communication-efficient structure-adaptive PMPC protocols for these settings. Section 5.4 contains necessary preliminaries and specifications of underlying network/communication models, adversary models, and some other basic building blocks required in the rest of the chapter. We then describe the details of the new protocols developed in this work and their security proofs in Section 5.5. Finally, we compare our work to related work in Section 5.6 and conclude this chapter.

5.2 The Need for Secure Computation for Dynamic Groups with Changing Specifications of the General Adversary Structures

Large networked systems, such as public clouds, private clouds and companies' computer infrastructure, are managed for security and reliability by specialists who employ tools, measurements, and reporting systems (including AI tools nowadays). These specialists maintain such large systems while facing changes and failures. This methodology of managing large systems is known as *DevOps* which is a set of practices that combines software development (Dev) and information-technology operations (Ops) that aims to shorten the systems development life cycle and provide continuous delivery with high software quality and system reliability [215]. In particular (starting with Google), the profession of such people performing these tasks is called *Site Reliability Engineering (SRE)*. Some of the responsibilities of SRE include: (1) Reducing organizational silos (separate sections of engineers to create joint coherent responsibilities in large systems with various elements cooperating); (2) Accepting failure as normal (and reacting to failures such as security breaches, overloading of subsystems, etc., managing system configuration with responsiveness and agility); (3) Implementing gradual changes (long-term maintenance based on past issues and future needs as they come or envisioned); (4) Leveraging tooling and automation (as the large system needs to be controlled remotely, effectively this cannot be performed manually, and control tools are needed); and (5) Measuring everything (constantly monitoring the needs and acting according to the data while keeping statistics on the performance of systems).

A modern information security concept in managing large systems and defending against threats is *moving target defense (MTD)*, which is the method of controlling change across multiple system dimensions to increase uncertainty and apparent complexity for attackers, reduce their window of opportunity, and increase the costs of their probing and attack efforts. MTD assumes that perfect security is unattainable and adds system changes as increased challenges to the potential attacker.

One can view an SRE team getting information about and reacting to a system's suspicious behavior (at some parts of the network) and employing analysis that dictates configuration change. One can also view the team as occasionally and proactively, for the sake of implementing an MTD strategy, calling the network of servers to rearrange itself differently than the current setting in the (general) scenarios we consider. This would correspond to changing the specification of the general adversary structure being protected against. When the team manages the configuration, they employ a secure and authenticated control and command system over servers, they can notify servers to reconfigure and organize their distributed data according to some protocol and dictated parameters, and certain servers do reboot from clean state. In our treatment, we assume that such a system is available in our underlying secure computation system and we augment existing configuration tools with the ability to manage and dynamically change the underlying "secret sharing" settings among the network's servers.

5.3 Overview of Proactive MPC and Design Roadblocks

5.3.1 Blueprint of Proactive Secret Sharing (PSS) and Proactive MPC (PMPC)

PMPC protocols [241, 48] are usually constructed on top of (linear) secret sharing schemes, and involve alternating compute and refresh (and reboot/reset) phases. The refresh phases involve distributed re-randomization of the secret shares and deleting old ones to ensure that a mobile adversary does not obtain enough shares (from the same phase) that can allow them to violate secrecy of the shared inputs and intermediate compute results. A PMPC protocol usually consists of the following six sub-protocols:

1. `Share`: allows a dealer (typically one of the parties) to share a secret s among n parties;
2. `Reconstruct`: allows parties to reconstruct a shared secret s using the shares they collectively hold;
3. `Refresh`: is executed between two consecutive phases, w and $w+1$, and generates new shares for phase $w+1$ that encode the same secret as, but are independent of the shares in phase w, and erases the old shares;
4. `Recover`: allows parties that lost their shares (due to rebooting/resetting or other reasons) to obtain new shares encoding the same secret s with the help of other online parties;
5. `Add`: allows parties holding shares of two secrets, s and t, to obtain shares encoding the sum, $s+t$; and
6. `Multiply`: allows parties holding shares of two secrets, s and t, to obtain shares encoding the product, $s \cdot t$.

The overall operation of a standard PMPC protocol is as follows: First, each party uses the `Share` sub-protocol to securely distribute its private inputs among the n parties (including itself). The function to be computed on parties' inputs is transformed into a public arithmetic circuit. The circuit is composed of multiple layers (the depth of the circuit), where each consists of a set of `Add` and `Multiply` gates computed via the corresponding sub-protocols one layer at a time. At the end of each circuit layer[2], shares of all nodes can be refreshed via the `Refresh` protocol and old shares are deleted; refreshing and deleting old shares ensure that different shares collected by the adversary at different phases can not be used together to reconstruct the secret shared inputs and intermediate and final results of the computation. In addition, during refresh phases, some nodes are randomly reset/rebooted, these then use the `Recover` protocol to obtain new shares encoding the same shared secrets corresponding to the current state of the PMPC computation, i.e., the output of the current circuit layer and any shared values that will be needed in future layers. When the (secret shared) output of the final layer of the computation is produced, parties use `Reconstruct` protocol to compute the final output in the clear (or towards whichever nodes are supposed to obtain it).

To deal with dynamic groups, where parties can leave, or new parties can join the group, the following additional sub-protocol `Redistribute` is required:

7. `Redistribute`: is executed between two consecutive protocol phases, w and $w+1$, and allows parties in a new group (in phase $w+1$) to obtain new shares that encode the same secret as the shares in phase w.

In addition, we observe that the specifics of the secret sharing-based encoding underlying the PMPC protocol largely dictate the communication-efficiency. This is an issue that is often overlooked and that does not appear when one only considers the threshold adversary structure as opposed to GAS. For example, if one considers an additive secret sharing

[2]Or after several layers, or at the end of one execution of a circuit of reactive functionalities executing in a loop. In this work, we do not specify when parties should refresh shares, we just develop the protocol to accomplish this.

scheme similar to the one used in the MPC protocol in [224], and if the adversary structure one should protect against is the threshold one, then there is an exponential blowup in the share size compared to a monotone span program (MSP) based scheme. Thus, any protocols that require transmitting such shares encoded additively, e.g., multiplication, recovery, or redistribution of shares, is going to be inefficient compared to an MSP-based one. A communication-efficient protocol should thus be structure-adaptive when considering evolving GAS, this means that if the set of parties performing the MPC receives (from an administrator) a request to adapt to a new GAS, for which it is known that another (secret sharing) encoding scheme is more efficient, they need to convert. We stress that this is different than the `Redistribute` protocol, which re-shares a shared secret but *with the same secret sharing scheme.* We require a non-trivial additional protocol to perform such a conversion:

8. `Convert`: is executed between two consecutive protocol phases, w and $w+1$, and allows parties in a new group defined by a new GAS (in phase $w + 1$) to obtain new shares under a different secret sharing scheme that encodes the same secret as the shares in phase w (under the old secret sharing scheme and the old GAS).

5.3.2 Roadblocks Facing PMPC for Dynamic General Adversary Structures and Dynamic Groups

Starting with appropriate SS and MPC protocols for GAS, the following is to be addressed to develop a communication-efficient PMPC scheme for dynamic groups and GAS:

1. *Design convert protocols to be structure-adaptive:* Given that we are considering settings with changing GAS, and given that some (secret sharing) encoding schemes underlying MPC results in different communication complexities, we design new efficient protocols (secure against GAS) to convert between different secret-sharing schemes.

We consider converting from an additive sharing to an MSP-based sharing, and in the opposite direction. Such conversion protocols may be of independent interest.

2. *Design refresh and recover protocols to "proactivize" the underlying SS scheme:* This enables parties to re-randomize the shares in a secure, distributed manner. To enable rebooted/reset parties to recover their shares and not to lose shared inputs or intermediate results of the computation over time, rebooted parties have to be able to recover shares with the help of the rest of the parties.
3. *Design a redistribute protocol for settings with dynamic groups:* In such settings, parties can leave and newly join the group performing the computation, which results in the GAS, as well as the number of parties in the group, changing over time. One has to redistribute new shares to parties in the new group which encode the same secret as the shares in the previous group, but also needs to prevent departing parties from using their shares to obtain any information about the secret.
4. *Efficient communication in all protocols:* All the involved protocols should be efficient, e.g., ideally have a linear dependence on the specification of adversary structures and the number of parties, or at least (a low) polynomial. We note that in this work, we do not attempt to minimize the descriptions of the adversary structures, i.e., the size of specifications of some structures may be exponential in the number of parties n.

5.4 Preliminaries

This section provides the preliminaries required for the rest of this chapter. We first provide terminology used in proactive security and types of security, communication, and adversary models in MPC literature. Then we discuss the underlying models and security guarantees we consider in our work and review the information checking and dispute control schemes used in the MPC protocols [170, 204] on which we build our protocols.

5.4.1 Terms in Proactive Security

Let $\mathcal{P} = \{P_1, ..., P_n\}$ be a set of n participating parties to compute a function f over a finite field $\mathbb{F}$. We adopt the previous formalization of the proactive security model from [30, 48, 134]. Briefly speaking, a proactive protocol proceeds in *phases*, and a phase consists of a number of consecutive rounds. Two types of phases, *refresh* and *operational*, alternate, where a refresh phase re-randomizes the shares, and an operational phase performs the computations. Finally, a *stage* is a larger notion than a phase, consisting of three phases: an *opening* refresh phase, an operational phase, and a *closing* refresh phase – i.e., each refresh phase is not only the closing of one stage, but also the opening of another stage. If an adversary corrupts a party during an operational phase, the adversary is given the view of the party starting from its state at the beginning of the current operational phase. Otherwise, if the corruption is made during a refresh phase, the adversary gets the view in both stages, u and $u+1$, that include the refresh phase as the closing and the opening and the party is assumed to be corrupted for the stage $u+1$. Detailed definitions are presented below.

Phases The rounds of a proactive protocol are grouped into *phases* $\phi_1, \phi_2, \ldots$: a phase ϕ consists of a sequence of consecutive rounds, and every round belongs to exactly one phase. There are two types of phases: *refresh* phases and *operation* phases. The phases alternate between refresh and operation phases; the first and the last phase of the protocol are both operation phases. Each refresh phase is furthermore subdivided into a *closing period* consisting of the first k rounds of the phase, followed by an *opening period* consisting of the final $\ell - k$ rounds of the phase, where ℓ is the total number of rounds in the phase.

In non-reactive MPC, the number of operation phases can be thought to correspond to the depth of the circuit to be computed in the MPC. Intuitively, each operation phase serves to compute a layer of the circuit to be computed, and each refresh phase serves to re-randomize the data held by parties such that combining the data of corrupt parties across different phases will not be helpful to an adversary.

Stages A *stage* σ of the protocol consists of an *opening period* of a refresh phase, followed by the subsequent *operation phase*, followed by the *closing period* of the subsequent refresh phase. In the case of the first and last stages of a protocol, there is an exception to the alternating "refresh-operation-refresh" format: the first stage starts with the first operation phase and the last stage ends with the last operation phase. Thus, a stage spans (but does not cover) three consecutive phases, and the number of stages in a protocol is equal to its number of operation phases. ($\because$ For a protocol Π, if the number of phases in Π is $\#(\phi) = m$, then the number of operation phases in Π is $\#(\text{op}) = \lceil \frac{m}{2} \rceil$ and the number of stages in Π is $\#(\sigma) = \frac{m-3}{2} + 2 = \frac{m+1}{2}$. Since $m = 2m' + 1$ for some $m' \in \mathbb{N}$, $\#(\text{op}) = \#(\sigma) = m' + 1$.)

Stage Changes The adversary $\mathcal{A}$ can trigger a new stage at any point during an operation phase by sending a special message `newstage` to all parties. Upon receiving the `newstage` message, the parties initiate a refresh phase.

Corruptions If a party P_i is corrupted by the adversary $\mathcal{A}$ during an operation phase of a stage σ_j, then $\mathcal{A}$ learns the view of P_i starting from his state at the beginning of stage σ_j. If the corruption is made during a refresh phase between consecutive stages σ_j and σ_{j+1}, then $\mathcal{A}$ learns P_i's view starting from the beginning of stage σ_j. Moreover, in the case of corruption during a refresh phase, P_i is considered corrupt in both stages σ_j and σ_{j+1}.

Finally, if P_i is corrupted during the closing period of a refresh phase in stage σ_j, $\mathcal{A}$ may decide to *decorrupt* him. In this case, P_i is considered to be no longer corrupted in stage σ_{j+1} (unless $\mathcal{A}$ corrupts him again before the end of the next closing period). A de-corrupted party P_i immediately rejoins the protocol as an honest party: if P_i was passively corrupted, then it rejoins with the correct state according to the protocol up to this point; or if P_i was actively corrupted, then it is restored to a clean default state (which may be a function of the current round). Note that in restoring a party to the default state, its randomness tapes are overwritten with fresh randomness: this is important since, otherwise, any once-corrupted party would be deterministic to the adversary.

Erasing State In our model, parties erase their internal state (i.e., the content of their tapes) between phases. The capability of erasing state is necessary in the proactive model: if an adversary could learn all previous states of a party upon corruption, then achieving security would be impossible, since throughout a protocol execution a proactive adversary would be able to learn the state of *all* parties in certain rounds.

5.4.2 Adversary Models

An adversary's capability can be described by a corruption type and an adversary structure. The adversary structure, denoted by Δ (details in Section 5.4.4), is a set of subsets of parties that are potentially corruptible by an adversary. i.e., The adversary can choose a set of parties in Δ and corrupt all the parties listed in the set. The corruption types are classified as passive corruption, active corruption, and both, where each type means as follows:

Passive Corruption Passive adversaries (also called *honest-but-curious* (HBC) adversaries) can eavesdrop on all the views of corrupted parties, whereas they cannot forge the process that parties should follow. i.e., The corrupted parties should follow the protocol as described.

Active Corruption Active adversaries can take full control of the corrupted parties and make them behave arbitrarily from the protocol. i.e., They can forge the messages of corrupted parties as well as eavesdrop on all of their views.

From the MPC literature, the adversaries' capabilities can be categorized as follows.

Threshold Adversary Models In the classic t-threshold MPC, adversaries are assumed to be able to either passively or actively corrupt up to t parties. i.e., The adversary structure in the t-threshold model is the set of all subsets of $\mathcal{P}$ which size is at most t.

General Adversary Models General adversary structures (GAS) extend this threshold setting to non-threshold models. The adversary can actively corrupt a subset of parties and passively corrupt another subset of parties. Sometimes, it is classified as general adversary models and mixed general adversary models as follows: The former has adversaries who can either passively corrupt or actively corrupt the parties, while the latter has adversaries who can do both passive corruption and active corruption. In this work, we collectively describe both models as general adversary models.

This work considers GAS, which is more general and flexible compared to the threshold models and applicable to various cases, e.g., when a special combination of parties is needed for computation, or when some of the parties are authorized, et cetera.

5.4.3 Types of Security and Communication Models

MPC literature distinguishes between two types of security, perfect (or information-theoretic) security and cryptographic security. Protocols with information-theoretic security can withstand an adversary with unrestricted computing power, while protocols with cryptographic security restrict an adversary's computing power and assume certain assumptions about the hardness of some computational problems, e.g., factoring large integers or computing discrete logarithms. In this work, we consider protocols with unconditional security for both passive and active adversaries.

For communication models, the literature considers either synchronous or asynchronous models. In asynchronous communication models, there are no guarantees about data transmission between sender and receiver. In contrast, synchronous communication models guarantee that any pair of parties can communicate over a bilateral secure channel. That is, when a sender sends data to a receiver, the receiver is guaranteed to get data in certain times. The synchronous communication models sometimes include a broadcast channel which guarantees the consistency of received values. We consider a synchronous network of n parties connected by an authenticated broadcast channel and point-to-point channels. Note that without this setting, we do not guarantee the information-theoretic security.

5.4.4 Definitions in General Adversary Structures

Let $2^{\mathcal{P}}$ denote the set of all the subsets of $\mathcal{P}$. A subset of $2^{\mathcal{P}}$ is called *qualified* if parties in the subset can reconstruct/access the secret, while a subset of $2^{\mathcal{P}}$ that parties in the set obtain no information about the secret is called *ignorant*. Every subset of $\mathcal{P}$ is either qualified or ignorant. The secrecy condition is stronger: even if any ignorant set of parties holds any kind of partial information about the shared value, they must not obtain any additional information about the shared value.

The *access structure* Γ is the set of all qualified subsets of $\mathcal{P}$ and the *secrecy structure* Σ is the set of all ignorant subsets of $\mathcal{P}$. Naturally, Γ includes all supersets of each element in it (so often called *monotone access structure*), while Σ includes all subsets of each element in it. We call such minimum or maximum sets as *basis structure* and denote it with $\widetilde{\cdot}$. i.e., the basis access structure $\widetilde{\Gamma}$ is the set of all minimal subsets in Γ, and the basis secrecy structure $\widetilde{\Sigma}$ is the set of all maximal subsets in Σ.

As a generalization of specifying threshold adversaries' capabilities by a corruption type (passive or active) and a threshold t, an adversary can be described by a corruption type and an *adversary structure* Δ, where $\Delta \subseteq \Sigma$ is a set of subsets of parties that can be potentially corrupted. Note that the adversary structure in t-threshold SS is the set of all subsets of $\mathcal{P}$ of at most t parties and GAS extends this to non-threshold models. A GAS includes all of these structures, (Γ, Σ, Δ), and the adversary is specified by Σ and Δ. The (Σ, Δ)-adversary denotes the adversary that can passively corrupt some parties in a set A and actively corrupt some parties in a set B, where $A \in \Delta$ and $(A \cup B) \in \Sigma$.

5.4.5 Information Checking (IC) and Dispute Control

Information checking (IC) is used in some MPC literature to prevent active adversaries from announcing wrong values through corrupted parties. It is a three-party protocol among a sender P_s, a receiver P_r, and a verifier P_v. When P_s sends a message m to P_r, P_s also encloses an authentication tag to P_r while giving a verification tag to P_v through private channels. Whenever any disagreement about what P_s sent to P_r occurs, P_k acts as an objective third party and verifies the authenticity of m to P_r. The MPC protocols in this work use different variants of IC, but the common idea is to check if all the points that P_r and P_v have lie on the polynomial of degree 1. Note that this can be naturally extended to the polynomial of degree l, where l is the number of secrets in a batch of sharing, as in [204]. An IC scheme consists of two protocols, called `Authenticate` and `Verify`, where `Authenticate` generates valid tags for participating parties with respect to the input value, and `Verify` verifies the input value with the input tags.

MPC protocols also use the *dispute control* to deal with the detected cheaters. Each party P_i locally maintains two lists: a list $\mathcal{D}_i$ of parties that P_i distrusts, and the list $\mathcal{D}$ of pairs of parties in dispute with each other. These lists are empty when the protocol begins, and whenever any dispute arises between two parties P_i and P_j (for example, P_i insists that P_j is lying), the pair $\{P_i, P_j\}$ is added to the dispute list $\mathcal{D}$. Since all disputes are broadcasted, each party P_i has the same list $\mathcal{D}$ while maintaining its own list $\mathcal{D}_i$. After P_j is added to $\mathcal{D}_i$, P_i behaves in all future invocations of the protocol for authentication and verification with

P_j as if it fails, whether this is the case or not. Some MPC schemes also maintain a list $\mathcal{C}$ of parties that everyone agrees are corrupted.

5.5 Proactivizing MPC Protocols for Dynamic General Adversary Structure and Dynamic Groups

As mentioned in Section 5.3, our PMPC protocols build on two MPC protocols with different underlying secret sharing schemes. One is an MPC protocol [170] based on additive secret sharing, and the other [204] is based on a monotone span program (MSP) with multiplication. For convenience, in the rest of the chapter, we call the former as *additive MPC* and the latter as *MSP-based MPC*. Both guarantee unconditional security against active $Q2$ adversaries. $Q2$ means no two sets in Δ cover the entire set of parties; i.e., for $\forall A, B \in \Delta$, $\mathcal{P} \nsubseteq A \cup B$. Table 5.1 summarizes the notations we use in this chapter.

Notation	**Explanation**
$\mathcal{P} = \{P_1, ..., P_n\}$	a set of participating parties in a protocol
(Γ, Σ, Δ)	the access/secrecy/adversary structures in a GAS
$\mathbb{S}$	a sharing specification describing how shares are distributed
$w, w+1$	a phase (number)
$[s]^w$	a sharing of a secret s in phase w, i.e., a set of shares of s
$\mathbb{F}$	a finite field
$\mathcal{D}$	the (public) list of pairs of parties who are in dispute with each other
$\mathcal{D}_i$	a (local) list of parties that P_i distrusts
$\mathcal{C}$	the (public) list of parties that everyone agrees to their corruptness
M	a matrix from a MSP $\widehat{M} = (\mathbb{F}, M, \rho, \mathbf{r})$[3]
$\mathbf{a}$	a vector (with bold texts)
M_i	a matrix of rows of M assigned to P_i according to an indexing function
M_A	a matrix of rows of M assigned to all $P_i \in A$ according to an indexing function
$\langle \, , \rangle$	the inner product
$A \setminus B$	the set difference of A and B, i.e., the set of elements in A but not in B
$a \overset{\$}{\leftarrow} \mathbb{F}$	randomly chosen element a from the finite field $\mathbb{F}$

Table 5.1: Notations used in Chapter 5

This section first presents the *additive PMPC* and *MSP-based PMPC* schemes with our new additional protocols to "proactivize" each MPC scheme. We formalize the base protocols of

[3]Detailed definitions of components of MSP are provided in Section 5.5.4.

[170] and [204] in Sections 5.5.1 and 5.5.3, and provide our new protocols in Sections 5.5.2 and 5.5.4. For proactivizing an MPC scheme, we develop two new main protocols, called `Refresh` and `Recover`, and add one more protocol, called `Redistribute`, for dynamic groups. The resulting PMPC is composed of 6 protocols, `Share`, `Reconstruct`, `Add`, `Multiply`, `Refresh`, and `Recover`, or 7 in the dynamic groups case when including `Redistribute`. For clarification, we denote each protocol with superscripts, `A` or `M`, for additive PMPC and MSP-based PMPC, respectively. Note that the complexity of additive PMPC protocols depends on $|\widetilde{\Sigma}|$ and n, while that of MSP-based PMPC protocols depends on d and n, where n is the number of participating parties, $|\widetilde{\Sigma}|$ is the size of the set of all maximal subsets in the secrecy structure, and d is the number of rows of the MSP matrix.

Then, in Section 5.5.5, we develop *share conversion* protocols between those schemes to enable one to adapt/change the utilized protocols according to the dynamic GAS. As we mentioned in Section 5.2, this is necessary and important because one can become more communication-efficient than the other depending on the circumstances. For instance, considering the upper bound on d is about $|\widetilde{\Sigma}|^{2.7}$ [204], the MSP-based MPC is more expensive than the additive MPC, but d can also be low as $n = |\mathcal{P}|$ in some cases, which makes the MSP-based MPC more communication-efficient.

5.5.1 Protocols for Additive MPC Scheme [170]

We build our additive PMPC protocol on top of Hirt and Tschudi's unconditional MPC [170] based on additive secret sharing.

Assuming n participating parties $\mathcal{P} = \{P_1, ..., P_n\}$, parties want to share a sharing of a secret s according to the sharing specification $\mathbb{S}$. Any Δ-*private* sharing specification, which means for every $Z \in \Delta$, $\exists S \in \mathbb{S}$ such that $S \cup Z = \emptyset$, can be used to securely share a secret, and we adopt one from [224], $\mathbb{S} = (S_1, ..., S_k)$, where $S_i = \mathcal{P} \setminus T_i$ for $\widetilde{\Sigma} = \{T_1, ..., T_k\}$, the set of all maximal subsets in Σ.

Information Checking

In [170], they use an IC scheme for dealing with active adversaries, consisting of $\texttt{Authenticate}^{\texttt{A}}$ and $\texttt{Verify}^{\texttt{A}}$. $\texttt{Authenticate}^{\texttt{A}}(P_s, P_r, P_v, m)$ is for a sender P_s to distribute the authentication tag of m to a receiver P_r and the verification tag of m to a verifier P_v, and $\texttt{Verify}^{\texttt{A}}(P_s, P_r, P_v, m', tags)$ is for P_j to request P_v to verify the value m' with an authentication tag and a verification tag.

For any pair of two parties (P_s, P_v), it is assumed that a fixed secret value, denoted by $\alpha_{s,v} \in \mathbb{F} \setminus \{0, 1\}$, is known by the parties. A vector (s, y, z, α) is *1-consistent* if there exists a polynomial f of degree 1 over $\mathbb{F}$ such that $f(0) = s, f(1) = y, f(\alpha) = z$. And a value s is called (P_s, P_r, P_v)-*authenticated*, if P_r knows s and some authentication tag y and P_v knows a verification tag z such that $(s, y, z, \alpha_{s,v})$ is 1-consistent. The vector $(y, z, \alpha_{s,v})$ is denoted by $A_{s,r,v}(s)$. The protocol $\texttt{Authenticate}^{\texttt{A}}$ allows P_i to securely (P_i, P_j, P_k)-authenticate the value s. If P_k is honest and s is known to the honest parties $\{P_i, P_j\}$, then $\texttt{Authenticate}^{\texttt{A}}(P_i, P_j, P_k, s)$ either securely (P_i, P_j, P_k)-authenticate s or aborts with error probability at most $1/|\mathbb{F}|$.

---

[170] Protocol $\texttt{Authenticate}^{\texttt{A}}(P_i, P_j, P_k, w, s) \longrightarrow (y, z)$ or $\perp$

Input: P_i and P_j holding s, $P_k \in \mathcal{P}$, a phase w, and a value s
Output: a pair of authentication tag y and verification tag z, or aborted

1. P_i randomly chooses $(y, z) \in \mathbb{F}^2$ and $(s', y', z') \in \mathbb{F}^3$ such that $(s, y, z, \alpha_{i,k})$ and $(s', y', z', \alpha_{i,k})$ are 1-consistent, and sends (s', y, y') to P_j and (z, z') to P_k.
2. P_k broadcasts random value $r \in \mathbb{F}$.
3. P_i broadcasts $s'' := rs + s'$ and $y'' := ry + y'$.
4. P_j checks if the forwarded values s'', y'' are correct by comparing $s'' \overset{?}{=} rs + s'$ and $y'' \overset{?}{=} ry + y'$ and broadcasts OK/NOK. If NOK is broadcasted, then P_j adds P_i to the list $\mathcal{D}_j$, and the

protocol is aborted, which outputs $\perp$.

5. P_k checks if $(s'', y'', rz + z', \alpha_{i,k})$ is 1-consistent. If it is, then P_k sends OK to P_j. Otherwise, P_k sends $(\alpha_{i,k}, z)$ to P_j and adds P_i to the list $\mathcal{D}_k$. When P_j receives $(\alpha_{i,k}, z)$, P_j adjusts y such that $(s, y, z, \alpha_{i,k})$ is 1-consistent.
6. P_j outputs y as the authentication tag and P_k outputs z as the verification tag.

Assuming that P_k knows a candidate s' or a (P_i, P_j, P_k)-authenticates value s and P_j wants to prove the authenticity of s', $\texttt{Verify}^{\texttt{A}}$ allows the parties to authenticate s' with their tags. If P_k and P_j are honest parties knowing $s' = s$, P_k will output s in $\texttt{Verify}^{\texttt{A}}$, or output $\perp$ otherwise, except with error probability at most $1/(\mathbb{F} - 2)$.

---

[170] Protocol $\texttt{Verify}^{\texttt{A}}(P_i, P_j, P_k, w, s', A_{i,j,k}(s)) \longrightarrow s$ or $\perp$

Input: a candidate value s' known to P_j and P_k for a (P_i, P_j, P_k)-authenticated value s, a phase w, and the authentication for s, $A_{i,j,k}(s) = (y, z, \alpha_{i,k})$, where P_j has the authentication tag y, and P_k has the verification tag z

Output: s or $\perp$

1. P_j sends y to P_k.
2. P_k checks if $(s', y, z, \alpha_{i,k})$ is 1-consistent and outputs s' if it is.
   Otherwise, P_k adds P_j to the list $\mathcal{D}_k$ and outputs $\perp$.

As the parties use local dispute control, even though the adversary has at most n^2 attempts to cheat, the total error probability of arbitrarily many instances of each protocol is at most $O(n^2/|\mathbb{F}|)$, which is independent of secrecy structure.

Secret Sharing

Secret sharing consists of $\texttt{Share}$ and $\texttt{Reconstruct}$ protocols, where $\texttt{Share}$ generates a sharing of an input secret and $\texttt{Reconstruct}$ collectively reconstructs the secret from the input sharing. i.e., A secret value s is shared among $\mathcal{P}$ through $\texttt{Share}^{\texttt{A}}$ and any qualified subgroup

B of $\mathcal{P}$ can reconstruct the secret by $\texttt{Reconstruct}^{\texttt{A}}$.

In $\texttt{Share}^{\texttt{A}}$ protocol, a dealing party randomly chooses $k-1$ values in $\mathbb{F}$, sets the k-th value as $s - \sum_{i=1}^{k-1} s_i$, and sends each i-th value to every player in S_i. Then multiple $\texttt{Authenticate}^{\texttt{A}}$ are invoked to generate the IC tags. Note that the sharing of s is linear and does not leak any information about s without the whole set of sharing.

---

[170] Protocol $\texttt{Share}^{\texttt{A}}(w, s, \mathcal{P}) \longrightarrow [s]^w$

Input: a phase w, a secret value s, and a set $\mathcal{P}$ of parties who receives the shares
Output: k shares of s in phase w

```
1. A dealing party $P_D$ chooses $k-1$ random integers, $s_1, ..., s_{k-1} \xleftarrow{\$} \mathbb{F}$, and sets the $k$-th share as
   $s_k := s - \sum_{i=1}^{k-1} s_i$.
2. For all $i \in \{1, ..., k\}$, do the following:
3.    $P_D$ sends $s_i$ to every party in $S_i$.
4.    $\forall P_a, P_b \in S_i$ and $\forall P_c \in \mathcal{P}$ invoke $\texttt{Authenticate}^{\texttt{A}}(P_a, P_b, P_c, s_i)$. If any result was aborted,
      $P_D$ broadcasts $s_i$, the parties in $S_i$ replace their share, and $\forall P_a, P_b \in S_i$ and $\forall P_c \in \mathcal{P}$ set the
      forwarded value $s_i$ as the authentication and verification tags.
5. The parties in $\mathcal{P}$ collectively output $[s]$.
```

On the other hand, In $\texttt{Reconstruct}^{\texttt{A}}$, parties in B verify the forwarded values of each share from the others using $\texttt{Verify}^{\texttt{A}}$ and reconstruct the secret value by locally adding all the verified share values. The value for s_q forwarded from P_j to P_k is denoted by $s_q^{(j,k)}$. It is shown in [170] that the following protocol securely reconstructs s to the parties in B if $Q^2(\mathbb{S}, \Delta)$ is met. i.e., For $\forall Z_1, Z_2 \in \Delta$, $\forall S \in \mathbb{S}$, $S \not\subseteq Z_1 \cup Z_2$.

---

[170] Protocol $\texttt{Reconstruct}^{\texttt{A}}(w, [s], B) \longrightarrow s$ or $\perp$

Input: a phase w, a sharing of s (collectively), and a set B of participants in reconstruction

```
    Output: $s$ or aborted
1.  For all $q \in \{1, ..., k\}$, do the following:
2.      Every party in $S_q$ sends $s_q$ to each party in $B$.
3.      For all $P_j \in S_q$ and $P_k \in B$,
4.          $\texttt{Verify}^{\texttt{A}}(P_i, P_j, P_k, w, s_q^{(j,k)}, A_{i,j,k}(s_q))$ is invoked for $\forall P_i \in S_q$. If $P_k$ outputs $s_q^{(j,k)}$ in each
            invocation, $P_k$ accepts it as value for $s_q$.
5.      Each $P_k \in B$ outputs $\perp$ if he never accepted in Step 4.
6.  Each party in $B$ locally adds up the accepted shares and outputs the sum.
```

Addition and Multiplication

Assuming the shares for the values s and t are shared among $\mathcal{P}$, adding s and t can be done naturally without any interaction among n parties. Due to the linearity, each party can locally add two shares and set it as the new share for $s+t$. i.e., $[s+t] = \{(s+t)_1, ..., (s+t)_k\}$ where $(s+t)_i; = s_i + t_i$.

[170] Protocol $\texttt{Add}^{\texttt{A}}(w, [s], [t]) \longrightarrow [s+t]$

```
    Input: phase $w$, shares of $s$, and shares of $t$
    Output: new shares of $s+t$
    Precondition: Two values $s = \sum_{i=1}^{k} s_i$ and $t = \sum_{i=1}^{k} t_i$ are shared
    Postcondition: $s+t$ is shared independently
1.  Each party $P_h$ locally adds each share of $s$ to the share of $t$ and keep the result as a share of
    $s+t$. i.e., $(s+t)_i; = s_i + t_i$ for each $i \in \{i \in \{1, ..., k\} | P_h \in S_i\}$.
```

On the other hand, it is quite tricky and requires a lot of communication to securely form the share of $(s * t)$ among n parties, as $s \cdot t = \sum_{i=1}^{k} \sum_{j=1}^{k} (s_i \cdot t_i)$. To securely form the share of $(s * t)$ among n parties, where s and t are pre-shared through $\texttt{Share}^{\texttt{A}}$ protocol, participating parties need to perform the protocol $\texttt{Multiply}^{\texttt{A}}$ below. Each party computes the local product $(s_p * s_q)$ for all s_p and s_q that the party holds, and shares it. Then, they perform a probabilistic check in each loop to identify corrupted parties. For privacy, the multiplication of random values is used instead of actual multiplying values.

---

[170] Protocol $\texttt{Multiply}^{\texttt{A}}(w, [x], [y]) \longrightarrow [xy]$

Input: a phase w, a sharing of x, and a sharing of y, collectively
Output: a sharing of xy

```
1. Set $M = \emptyset$ and invoke $\texttt{RandomTriple}^{\texttt{A}}(w, M)$.
2. If the protocol outputs $M'$, then repeat Step 1 with $M'$. Otherwise, use the output as random
   multiplication triple $([a], [b], [c])$ such that $c = ab$.
3. Each party locally computes $[d_x] := [x] - [a]$ and $[d_y] := [y] - [b]$.
4. parties invoke $\texttt{Reconstruct}^{\texttt{A}}(w, [d_x], \mathcal{P})$ and $\texttt{Reconstruct}^{\texttt{A}}(w, [d_y], \mathcal{P})$ to get $d_x$ and $d_y$, and
   locally compute $d_x d_y + d_x[b] + d_y[a] + [c]$ and set it as the share of $xy$.
```

The protocol $\texttt{Multiply}^{\texttt{A}}$ uses $\texttt{RandomTriple}^{\texttt{A}}$ as a subprotocol to obtain a random multiplication triple $([a], [b], [c])$ such that $c = ab$, and compute the sharing of xy by computing $xy = ((x-a)+a)((y-b)+b) = (d_x+a)(d_y+b) = d_x d_y + d_x b + d_y a + ab = d_x d_y + d_x b + d_y a + c$. In $\texttt{RandomTriple}^{\texttt{A}}$, $I_Z(i)$ denotes the set of pairs of shares assigned to P_i, i.e., $I_Z(i) := \{(p, q) | P_i = min_{\mathcal{P}}(P \in (S_p \cap S_q) \setminus Z)\}$, for some $Z \in \Delta$.

---

[170] Protocol $\texttt{RandomTriple}^{\texttt{A}}(w, M) \longrightarrow ([a], [b], [c])$ or M'

Input: a phase w, and a set of (identified) malicious parties M
Output: a random multiplication triple $([a], [b], [c])$ or a set M' such that $M \subsetneq M'$

```
1. Parties generate random shared values $[a], [b], [b'], [r]$ by summing up shared random values (one
   from each party) for each value.
2. $\texttt{BasicMultiply}^{\texttt{A}}([a], [b], M)$ is invoked to compute $(([c_1], ..., [c_n]), [c])$ and $\texttt{BasicMultiply}^{\texttt{A}}([a], [b'], M)$
```

is invoked to compute $(([c'_1], ..., [c'_n]), [c'])$.

3. $\texttt{Reconstruct}^{\texttt{A}}(w, [r], \mathcal{P})$ is invoked and each party gets the value r.
4. Each party locally computes $[e] := r[b] + [b']$.
5. $\texttt{Reconstruct}^{\texttt{A}}(w, [e], \mathcal{P})$ is invoked and each party gets the value e.
6. Each party locally computes $[d] := e[a] - r[c] - [c']$.
7. $\texttt{Reconstruct}^{\texttt{A}}(w, [d], \mathcal{P})$ is invoked and each party gets the value d.
8. If $d = 0$, each party collectively outputs $([a], [b], [c])$. Otherwise, reconstruct the sharings $[a], [b], [b'], [c_1], ..., [c_n], [c'_1], ..., [c'_n]$ and output $M' := M \cup \{P_i : rc_i + c'_i \neq \sum_{(p,q)\in I(i)} r(a_p b_q) + (a_p b'_q)\}$.

The protocol $\texttt{BasicMultiply}^{\texttt{A}}$ in $\texttt{RandomTriple}^{\texttt{A}}$ computes the sharing of $c = ab$, where the inputs are the sharings of two values, $[a]$ and $[b]$ and a set of malicious parties M. It also outputs the sharing of the shares c_i's of c such that $[c] = \sum_{i=1}^{n}[c_i]$, if no more actively corrupted parties exist in $\mathcal{P} \setminus M$ as below.

---

[170] Protocol $\texttt{BasicMultiply}^{\texttt{A}}(w, [a], [b], M) \longrightarrow ([c_1], ..., [c_n]), [c])$ or $\perp$

Input: a phase w, sharings of a and b, and a set of (identified) malicious parties M
Output: $([c_1], ..., [c_n])$ and $[c] = \sum_{i=1}^{n}[c_i]$, if no party in $\mathcal{P} \setminus M$ actively cheats

```
1. For all $S_q$ such that $S_q \cap M \neq \emptyset$,
2.   Every party in $S_q$ sends their holding values for $a_q$ and $b_q$ to each other.
3.   For all $P_j, P_k \in S_q$, $\texttt{Verify}^{\texttt{A}}(P_i, P_j, P_k, w, a_q^{(j,k)}, A_{i,j,k}(a_q))$ and $\texttt{Verify}^{\texttt{A}}(P_i, P_j, P_k, w, b_q^{(j,k)}, A_{i,j,k}(b_q))$
     are invoked for $\forall P_i \in S_q$. If $P_k$ outputs $a_q^{(j,k)}$(or $b_q^{(j,k)}$, respectively) in each invocation, $P_k$
     accepts it as value for $a_q$ (or $b_q$).If all output $\perp$, the protocol is aborted.
4. a) Each party $P_i \in \mathcal{P} \setminus M$ locally computes and shares $c_i = \sum_{(p,q)\in I(i)} a_p b_q$, where $I(i) :=$
     $\{p, q)|P_i = min_P(P \in S_p \cap S_q)\}$.
   b) Each party $P_i \in M$ sets the sharing of $c_i$ as $(c_i, 0, ..., 0)$ where $c_i = \sum_{(p,q)\in I(i)} a_p b_q$, and
     $\forall P_j, P_k$ set corresponding tags as $y_j = [c_i]_j, z_j = [c_i]_j$, for $j = 1, ..., k$.
     i.e., The tags are $(c_i, c_i)$ only for $[c_i]_1$ and the rest is $(0, 0)$ for all $[c_i]_j$.
5. Parties in $\mathcal{P}$ collectively output $([c_1], ..., [c_n])$ and $[c] = \sum_{i=1}^{n}[c_i]$.
```

5.5.2 Additive PMPC Scheme for Dynamic GAS and Dynamic Groups

To make this additive MPC scheme to be a PMPC that can also handle dynamic groups, we build three protocols: $\texttt{Refresh}^{\texttt{A}}$, $\texttt{Recover}^{\texttt{A}}$, and $\texttt{Redistribute}^{\texttt{A}}$.

$\texttt{Refresh}^{\texttt{A}}$ protocol periodically refreshes or re-randomizes the shares in a distributed manner. This can be done naturally by every party's sharing zero and locally adding all the received shares to the current holding share. The execution of this protocol does not reveal any additional information about the secret, as only the shares of zeros are communicated.

Protocol $\texttt{Refresh}^{\texttt{A}}(w, [s]) \longrightarrow [s]^{w+1}$

Input: a phase w and a sharing of s
Output: new sharing of s in phase $w+1$, $[s]^{w+1}$

1. Every party P_i in $\mathcal{P}$ invokes $\texttt{Share}^{\texttt{A}}(w, 0, \mathcal{P})$. (in parallel)
2. Each party adds all shares received in Step 1 to shares of s and sets the result as the new share of s in phase $w+1$.
3. parties in $\mathcal{P}$ collectively output $[s]^{w+1}$.

Theorem 5.1. *(Correctness and Secrecy of* $\texttt{Refresh}^{\texttt{A}}$*) When* $\texttt{Refresh}^{\texttt{A}}$ *terminates, all parties receive new shares encoding the same secret as old shares with error probability* $n^4|\mathbb{S}|/|\mathbb{F}|$*, and cannot obtain any information about the secret by the protocol execution. It requires* $|\mathbb{S}|(7n^4 + n^2)\log|\mathbb{F}|$ *bits of communication and broadcasts* $|\mathbb{S}|((3n^4 + n)\log|\mathbb{F}| + n^3)$ *bits.*

Proof. Because of the linearity of the authentication, each party can locally set up the corresponding authentication tag y and verification tag z. Let $r_{i,q}$ be the q-th share of zero from P_i. Then, for $[s]_q^w := (s_q, A_{i,j,k}(s_q))$, the new q-th share of s in phase $w+1$ is $[s]_q^{w+1} = (s'_q, A_{i,j,k}(s'_q))$, where $s'_q = s_q + \sum_{i=1}^n r_{i,q}$ and $A_{i,j,k}(s'_q) = (y_{s_q} + \sum_{i=1}^n y_{r_{i,q}}, z_{s_q} + \sum_{i=1}^n z_{r_{i,q}}, \alpha_{i,k})$. Also, since every party shares the sharing of zero, the new sharing of s also reconstructs the same value s as follows:

$$\sum_{q=1}^{h} s'_q = \sum_{q=1}^{h}(s_q + \sum_{i=1}^{n} r_{i,q}) = \sum_{q=1}^{h} s_q + \sum_{q=1}^{h}\sum_{i=1}^{n} r_{i,q} = \sum_{q=1}^{h} s_q + \sum_{i=1}^{n}\sum_{q=1}^{h} r_{i,q} = s + \sum_{i=1}^{n} 0 = s$$

In addition, each q-th share is verified with $A_{i,j,k}(s'_q) = (y', z', \alpha_{i,k})$ because $(s'_q, y', z', \alpha_{i,k})$ is 1-consistent for $\forall P_i, P_j \in S_q$ and $\forall P_k \in B$ for $\forall B \in \Gamma$. i.e.,

$$(f+F)(0) = f(0) + F(0) = s_q + \sum_{q=1}^{h} r_{i,q} = s'_q,$$
$$(f+F)(1) = f(1) + F(1) = y_{s_q} + \sum_{i=1}^{n} y_{r_{i,q}} = y', \text{ and}$$
$$(f+F)(\alpha_{i,k}) = f(\alpha_{i,k}) + F(\alpha_{i,k}) = z_{s_q} + \sum_{i=1}^{n} z_{r_{i,q}} = z'$$

As every party in $\mathcal{P}$ invokes $\texttt{Share}^{\texttt{A}}$, they communicate $n * Cost(\texttt{Share}^{\texttt{A}})$ bits. □

For $\texttt{Recover}^{\texttt{A}}$ protocol, we construct two sub-protocols, $\texttt{ShareRandom}^{\texttt{A}}$ and $\texttt{RobustReshare}^{\texttt{A}}$. $\texttt{ShareRandom}^{\texttt{A}}$ generates a sharing of a random element r in $\mathbb{F}$ and parties in the same S_i receive the i-th share of r for each i, but the value of r is not revealed to anyone. Since each iteration requires $O(|S_q|^2 \log |\mathbb{F}|)$ broadcast bits for each q and each $|S_q|$ is less than n, it broadcasts at most $O(|\mathbb{S}|n^2 \log |\mathbb{F}|)$ bits among parties and no communications is required.

Protocol $\texttt{ShareRandom}^{\texttt{A}}(w, \mathcal{P}) \longrightarrow [r]^w$

Input: a phase w and a set of participating parties $\mathcal{P}$

```
   **Output:** a sharing $[r]$ of a random number $r$, shared among $\mathcal{P}$
1. For each $S_q \in \mathbb{S} = \{S_1, ..., S_k\}$ :
2.    Each party $P_i \in S_q$ generates a random number $r_{qi}$ and broadcast it among all parties in $S_q$.
3.    Each $P_i \in S_q$ locally adds up all values received in Step 2 and sets it as $r_q$.
4. The parties in $\mathcal{P}$ collectively output $[r]$, where $r = \sum_{q=1}^{k} r_q$.
```

RobustReshare$^{\mathtt{A}}$ allows parties in $\mathcal{P}_R \in \Gamma$ to receive a sharing of an input random number r (with the value of r) from the parties in $\mathcal{P}_S$, where everyone in $\mathcal{P}_S$ knows the value of r. Distributing one sharing of r is non-trivial in the active adversary model because we cannot trust one party who might be corrupted. Let $Honest := \{\mathcal{P} \setminus A \mid A \in \overline{\Delta}\}$, where $\overline{\Delta}$ is the set of all maximal subsets in Δ. Since the adversary can corrupt one set of parties in Δ in each phase, there exists at least one set in $Honest$ which includes the honest parties only in that phase. The main idea is to find such a set by repeating to share and reconstruct for each party's holding value for r. At the end of the protocol, parties in $\mathcal{P}_R$ can set a sharing of r and also know the value of the random number r.

Protocol RobustReshare$^{\mathtt{A}}(w, r, \mathcal{P}_S, \mathcal{P}_R) \longrightarrow [r]^w$

```
   **Input:** a phase $w$, a random number $r$, a set $\mathcal{P}_S$ of parties sending $r$, and a set $\mathcal{P}_R$ of receiving
         parties, where $\mathcal{P}_R \in \Gamma$
   **Output:** a sharing of $r$ in phase $w$, $[r]^w$
1. Every party in $\mathcal{P}_S$ executes Share$^{\mathtt{A}}(w, r, \mathcal{P}_R)$ according to the sharing specification $\mathbb{S}_R$ on $\mathcal{P}_R$.
   Let $[r]^{(i)}$ be the sharing of $r$ that $P_{k_i} \in \mathcal{P}_S$ shares.
2. Parties in $\mathcal{P}_R$ invoke Reconstruct$^{\mathtt{A}}([r]^{(i)}, \mathcal{P}_R)$, for each $i = 1, 2, ..., |\mathcal{P}_S|$.
   Let $r^{(i)}$ be the output of each invocation.
3. Each party chooses a set $H \in Honest$ such that $\exists v, v = r^{(i)}$ for all $P_{k_i} \in H$. If multiple such
   sets exist, choose the minimal set including $P_i$ with lower id, $i$.
4. Output the sharing of $r$ from the party $P_i$ in $H$ with the minimum id, $i$.
   i.e. Output $[r] \leftarrow [r]^{(min)}$, where $min := \min_{P_i \in H}\{i\}$.
```

The security of RobustReshare$^{\mathtt{A}}$ relies on the security of Share$^{\mathtt{A}}$ and Reconstruct$^{\mathtt{A}}$, as the rest is executed locally. For complexities, as both protocols $\texttt{Share}^{\texttt{A}}$ and $\texttt{Reconstruct}^{\texttt{A}}$ are invoked for each party in $\mathcal{P}_S$ and $|Honest| = |\overline{\Delta}| \leq |\overline{\Sigma}| = |\mathbb{S}|$, the total communication and broadcast complexities of $\texttt{RobustReshare}^{\texttt{A}}$ is $O(|\mathbb{S}|n^3 + |\mathcal{P}_S||\mathbb{S}|n^3 + |Honest|) = O(|\mathcal{P}_S||\mathbb{S}|n^3)$. The total analysis of all additive PMPC protocols is shown in Table 5.2, with $\mathcal{P}_S$ denoting a set of sending parties, which is less than n.

Using these, $\texttt{Recover}^{\texttt{A}}$ allows rebooted/reset parties to obtain new shares for the same secret s with the assistance of other parties. Let $R \subset \mathcal{P}$ be a set of parties who need to recover their shares. Note that $\mathcal{P} \setminus R$ must still be in Γ to output a new sharing of s because, otherwise, it contradicts the definition of the access structure. It needs the condition $\mathcal{Q}^1(S_q, \mathcal{Z})$, which is already a necessary condition for the protocol $\texttt{Reconstruct}^{\texttt{A}}$. The main idea is as follows: a sharing of unknown random value r is generated among entire parties in $\mathcal{P}$ by $\texttt{ShareRandom}^{\texttt{A}}$ and the parties in $\mathcal{P} \setminus R$ holding the shares of s re-share the value $r' = r + s$ and a sharing of r' to entire parties. Then, all parties, including R, can compute the new shares of s by computing $[r'] - [r]$.

Protocol $\texttt{Recover}^{\texttt{A}}(w, [s], R) \longrightarrow [s]^{w+1}$ or $\perp$

Input: a phase w, a sharing of s, and a set of rebooted parties R
Output: new sharing of s in phase $w+1$, $[s]^{w+1}$, or aborted

```
1.  Parties in $\mathcal{P}$ invoke $\texttt{ShareRandom}^{\texttt{A}}(w, \mathcal{P})$ to generate a sharing $[r]$ of $r$, where $r$ is a random in $\mathbb{F}$.
2.  Each party in $\mathcal{P} \setminus R$ invokes $\texttt{Add}^{\texttt{A}}(w, [r], [s])$ to share the sharing of $r + s$.
3.  $\texttt{Reconstruct}^{\texttt{A}}(w, [r+s], \mathcal{P} \setminus R)$ is invoked and every party in $\mathcal{P} \setminus R$ gets $r' := r + s$.
4.  $\texttt{RobustReshare}^{\texttt{A}}(w, r', \mathcal{P} \setminus R, \mathcal{P})$ is invoked, and each party in $\mathcal{P}$ gets $[r']$.
5.  Each party computes $[r'] - [r]$ by executing $\texttt{Add}^{\texttt{A}}(w, [r'], -[r])$, where $-[r]$ is the additive inverses
    of the shares in $\mathbb{F}$.
```

Theorem 5.2. *(Correctness and Secrecy of* $\texttt{Recover}^{\texttt{A}}$*) If $\mathbb{S}$ and $\mathcal{Z}$ satisfy $\mathcal{Q}^1(\mathbb{S}, \mathcal{Z})$, the protocol* $\texttt{Recover}^{\texttt{A}}$ *allows a set $\forall R \in \Delta$ of rebooted parties to recover their shares encoding the same secret with error probability $O((n-|R|)|\mathbb{S}|n^3/|\mathbb{F}| + (n-|R|)|\mathbb{S}|n^2/(|\mathbb{F}|-2))$, and does not* *reveal any additional information about the secret. It communicates* $O((n-|R|)|\mathbb{S}|n^3 \log|\mathbb{F}|)$ *bits and broadcasts* $O((n-|R|)|\mathbb{S}|n^3 \log|\mathbb{F}|)$ *bits.*

Proof. **Correctness:** Since all parties in $\mathcal{P}$ hold both sharings of r and r' and by the linearity of additive sharing, each party's locally computing value is $[r'] - [r] = [r+s] + [-r] = [(r+s)-r] = [s]$. As $\texttt{Reconstruct}^{\texttt{A}}$ terminates with error probability $n^2|\mathbb{S}|/(|\mathbb{F}|-2)$ and $\texttt{RobustReshare}^{\texttt{A}}$ has error probability $O(|\mathcal{P}_S||\mathbb{S}|n^3/|\mathbb{F}| + |\mathcal{P}_S||\mathbb{S}|n^2/(|\mathbb{F}|-2))$, the protocol $\texttt{Recover}^{\texttt{A}}$ successfully ends with error probability $O((n-|R|)|\mathbb{S}|n^3/|\mathbb{F}| + (n-|R|)|\mathbb{S}|n^2/(|\mathbb{F}|-2))$. **Secrecy:** As each party locally adds its holding share of r and the share of s without reconstructing r or s, all they can see is each sharing of r' and the reconstructed value r'. Since r is a random shared element in $\mathbb{F}$, $r' = r+s$ is also random in $\mathbb{F}$, and it does not reveal any information about s without reconstructing r. Each party can sync the sharing of r' by $\texttt{RobustReshare}^{\texttt{A}}$. **Communication:** Recall that the protocol $\texttt{ShareRandom}^{\texttt{A}}$ communicates $|\mathbb{S}|n$ values in $\mathbb{F}$ and also broadcasts $|\mathbb{S}|n$ values in $\mathbb{F}$, the protocol $\texttt{RobustReshare}^{\texttt{A}}$ communicates/broadcasts $O(|\mathcal{P}_S||\mathbb{S}|n^3)$ values in $\mathbb{F}$, and the protocol $\texttt{Reconstruct}^{\texttt{A}}$ communicates $|\mathbb{S}|(n^3+n^2)$ values in $\mathbb{F}$ without broadcasting. Therefore, the total communication complexity is $|\mathbb{S}|n^2 + |\mathbb{S}|(n^3+n^2) + O((n-|R|)|\mathbb{S}|n^3) = O((n-|R|)|\mathbb{S}|n^3)$, and the total broadcast complexity is $|\mathbb{S}|n^2 + O((n-|R|)|\mathbb{S}|n^3) = O((n-|R|)|\mathbb{S}|n^3)$. □

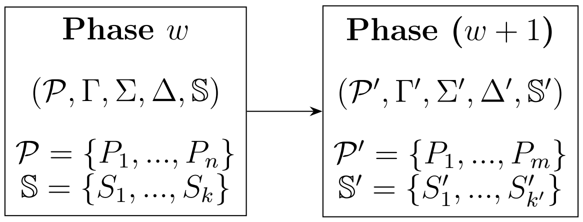

Phase w
$(\mathcal{P}, \Gamma, \Sigma, \Delta, \mathbb{S})$
$\mathcal{P} = \{P_1, ..., P_n\}$
$\mathbb{S} = \{S_1, ..., S_k\}$
Phase $(w+1)$
$(\mathcal{P}', \Gamma', \Sigma', \Delta', \mathbb{S}')$
$\mathcal{P}' = \{P_1, ..., P_m\}$
$\mathbb{S}' = \{S'_1, ..., S'_{k'}\}$

Figure 5.1: Dynamic groups and GAS in two consecutive phases, w and $w+1$

To handle dynamic groups and GASs, assume that the participating parties and structures are given as in Figure 5.1. As mentioned in Section 5.2, these phase information is specified by a trusted third party. $\texttt{Redistribute}^{\texttt{A}}$ allows new participating parties to obtain a sharing of the same secret as the previous phase according to the new GAS. The idea is quite intuitive because of the repetitive sharing properties, which is to double-share the sharing of a secret from the previous participating group to the new group. Note that the protocol $\texttt{Redistribute}^{\texttt{M}}$ we will show in the next section has different, non-trivial ideas and reduced complexities.

Protocol $\texttt{Redistribute}^{\texttt{A}}(w, s) \longrightarrow [s]^{w+1}$

```
   Input: phase $w$ and a secret $s$
   Output: shares of $s$ in phase $w+1$
   Precondition: parties in $P$ share $[s]^w$ for a secret $s$
   Postcondition: parties in $P'$ share $[s]^{w+1}$ encoding the same secret $s$
1. For each $S_i \in \mathbb{S}$:
2.    Each party $P_y$ in $S_i$ forwards its holding value $[s_i]_y$ for $s_i$ to every party in $S_i$ who is supposed
      to hold the same share (over the secure channel).
3.    $\texttt{Verify}^{\texttt{A}}(P_S, P_R, P_V, w, [s_i]_y, A_{S,R,V}(s_i))$ is invoked for all $P_R, P_V \in S_i$, $\forall P_S \in S_i$. If $P_V$
      outputs $[s_i]_y$ in each invocation, $P_V$ accepts it as value for $s_i$. Denote $v_i$ as the accepted
      value for $s_i$, for each $i$.
4.    Each party $P_y \in S_i$ runs $\texttt{Share}^{\texttt{A}}(w+1, v_i, \mathcal{P}')$ according to $\mathbb{S}'$.
5. For each $S'_j \in \mathbb{S}'$:
6.    Each party in $S'_j$ holds $\{v_{ij}\}_{i=1}^{k}$. For each $v_{ij}$, all $P_R, P_V \in S'_j$ invoke $\texttt{Verify}^{\texttt{A}}(P_S, P_R, P_V, w, v_{ij},$
      $A_{S,R,V}(v_{ij}))$ for $\forall P_S \in S'_j$ and accept the output value as $v_{ij}$.
7.    Each party in $S'_j$ sums up all $k$ values accepted in step 6 and sets it as new $j$-th share of $s$.
      i.e., $s'_j := \sum_{i=1}^{k} v_{ij}$.
```

THEOREM 5.3. *(Correctness and Secrecy of* $\texttt{Redistribute}^{\texttt{A}}$*) By executing* $\texttt{Redistribute}^{\texttt{A}}$*, new participating parties receive a sharing of the same secret as the old shares with error probability* $((|\mathbb{S}|n^3 + |\mathbb{S}'|m^3)/(|\mathbb{F}| - 2) + nm^3|\mathbb{S}'|/|\mathbb{F}|)$ *and it does not reveal any additional information about the secret. It communicates* $O(|\mathbb{S}||\mathbb{S}'|nm^3 \log |\mathbb{F}|)$ *bits and broadcasts* $O(|\mathbb{S}||\mathbb{S}'|nm^3 \log |\mathbb{F}|)$ *bits, where* $\mathbb{S}$ *and* $\mathbb{S}'$ *denote the sets for sharing specification in two consecutive phases and* n, m *are the number of parties in each participating group, i.e.,* $n = |\mathcal{P}|$ *and* $m = |\mathcal{P}'|$*. Assuming* $n = m$ *and* $|\mathbb{S}| = |\mathbb{S}'|$*, communication/broadcast complexities are* $O(|\mathbb{S}|^2 n^4 \log |\mathbb{F}|)$*.*

Proof. **Correctness:** The new sharing reconstructs the same secret s, as

$$\sum_{j=1}^{k'} s'_j = \sum_{j=1}^{k'} \sum_{i=1}^{k} v_{ij} = \sum_{i=1}^{k} \sum_{j=1}^{k'} v_{ij} = \sum_{i=1}^{k} v_i = \sum_{i=1}^{k} s_i = s.$$

For error probability, as the error probability of $\texttt{Verify}^{\texttt{A}}$ is $1/(|\mathbb{F}|-2)$ and the one of $\texttt{Share}^{\texttt{A}}$ is $n^3|\mathbb{S}|/|\mathbb{F}|$, the protocol $\texttt{Redistribute}^{\texttt{A}}$ outputs new sharing of s with error probability $|\mathbb{S}|n^3 Err(\texttt{Verify}^{\texttt{A}}) + max|S_i|Err(\texttt{Share}^{\texttt{A}}) + |\mathbb{S}'|m^3 Err(\texttt{Verify}^{\texttt{A}}) = (|\mathbb{S}|n^3 + |\mathbb{S}'|m^3)/(|\mathbb{F}| - 2) + nm^3|\mathbb{S}'|/|\mathbb{F}|$. **Secrecy:** Each party forwards their share to the parties who are supposed to have the same share, Step 1 does not reveal additional information about the share. Steps 3 to 6 reply on the secrecy of the protocols $\texttt{Verify}^{\texttt{A}}$ and $\texttt{Share}^{\texttt{A}}$, and Step 7 is local computation, which does not reveal any. **Communication:** In Step 1, each party in S_i sends their share value to each other, so they communicate $O(max|S_i|^2 \log|\mathbb{F}|)$ bits for each $i = 1, ..., |\mathbb{S}|$. Thus, the total communication complexity is $O(|\mathbb{S}|(n^2 \log|\mathbb{F}| + n^3 Cost(\texttt{Verify}^{\texttt{A}}) + nCost(\texttt{Share}^{\texttt{A}})) + |\mathbb{S}'|(n^3 Cost(\texttt{Verify}^{\texttt{A}}))) = O(|\mathbb{S}||\mathbb{S}'|nm^3 \log|\mathbb{F}|)$ bits and the total broadcast is $|\mathbb{S}|nCost(\texttt{Share}^{\texttt{A}}) = O(|\mathbb{S}||\mathbb{S}'|nm^3 \log|\mathbb{F}|)$ bits. When we assume $n = m$ and $|\mathbb{S}| = |\mathbb{S}'|$ for simplicity, the communication/broadcast complexities become $O(|\mathbb{S}|^2 n^4 \log|\mathbb{F}|)$. □

Note that the function of $\texttt{Recover}^{\texttt{A}}$ can be naturally substituted with $\texttt{Redistribute}^{\texttt{A}}$ with the same participating groups and the same sharing specification, but using our $\texttt{Recover}^{\texttt{A}}$ protocol is more efficient as it has linear complexity in $|\mathbb{S}|$, while $\texttt{Redistribute}^{\texttt{A}}$ has quadratic complexities in $|\mathbb{S}|$. Table 5.2 shows the total analysis of communication and broadcast complexities with error probability for each protocol in our additive PMPC scheme. In the table, R denotes the set of parties who need to recover their shares, and we assume $m = n$ and $|\mathbb{S}'| = |\mathbb{S}|$ for `Redistribute`. With that assumption, the total complexities for the static group (without `Redistribute`) are less than the complexities for `Multiply` and remain linear in $|\mathbb{S}|$, but for dynamic groups (with `Redistribute`), communication and broadcast complexities are quadratic in $|\mathbb{S}|$.

	Additive PMPC based on [170]			
	Protocol	**Communication Cost (bits)**	**Broadcast Cost (bits)**	**Error Probability for Protocol Failure**
Information Checking	`Authenticate`$^{\mathtt{A}}$ [170]	$7 \log \lvert\mathbb{F}\rvert$	$3 \log \lvert\mathbb{F}\rvert + 1$	$1/\lvert\mathbb{F}\rvert$
	`Verify`$^{\mathtt{A}}$ [170]	$\log \lvert\mathbb{F}\rvert$	-	$1/(\lvert\mathbb{F}\rvert-2)$
Secret Sharing / MPC	`Share`$^{\mathtt{A}}$ [170]	$O(\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	$n^3\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert$
	`Reconstruct`$^{\mathtt{A}}$ [170]	$O(\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	-	$n^2\lvert\mathbb{S}\rvert/(\lvert\mathbb{F}\rvert-2)$
	`Add`$^{\mathtt{A}}$ [170]	-	-	-
	`BasicMultiply`$^{\mathtt{A}}$ [170]	$O(\lvert\mathbb{S}\rvert n^4 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^4 \log \lvert\mathbb{F}\rvert)$	$O(n^4\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert)$
	`RandomTriple`$^{\mathtt{A}}$ [170]	$O(\lvert\mathbb{S}\rvert n^4 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^4 \log \lvert\mathbb{F}\rvert)$	$O(n^4\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert)$
	`Multiply`$^{\mathtt{A}}$ [170]	$O(\lvert\mathbb{S}\rvert n^5 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^5 \log \lvert\mathbb{F}\rvert)$	$O(n^5\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert)$
Our Additional Protocols	`Refresh`$^{\mathtt{A}}$	$O(\lvert\mathbb{S}\rvert n^4 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^4 \log \lvert\mathbb{F}\rvert)$	$n^4\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert$
	`ShareRandom`$^{\mathtt{A}}$	-	$O(\lvert\mathbb{S}\rvert n^2 \log \lvert\mathbb{F}\rvert)$	-
	`RobustReshare`$^{\mathtt{A}}$	$O(\lvert\mathcal{P}_S\rvert\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathcal{P}_S\rvert\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert(\lvert\mathcal{P}_S\rvert n^3/\lvert\mathbb{F}\rvert + \lvert\mathcal{P}_S\rvert n^2/(\lvert\mathbb{F}\rvert-2)))$
	`Recover`$^{\mathtt{A}}$	$O((n-\lvert R\rvert)\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	$O((n-\lvert R\rvert)\lvert\mathbb{S}\rvert n^3 \log \lvert\mathbb{F}\rvert)$	$O((n-\lvert R\rvert)n^3\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert + (n-\lvert R\rvert)n^2\lvert\mathbb{S}\rvert/(\lvert\mathbb{F}\rvert-2))$
	`Redistribute`$^{\mathtt{A}}$	$O(\lvert\mathbb{S}\rvert^2 n^4 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert^2 n^4 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^3/(\lvert\mathbb{F}\rvert-2) + n^4\lvert\mathbb{S}\rvert/\lvert\mathbb{F}\rvert)$
Total	`Additive PMPC` for Static groups	$O(\lvert\mathbb{S}\rvert n^5 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert n^5 \log \lvert\mathbb{F}\rvert)$	
	`Additive PMPC` for Dynamic groups	$O(\lvert\mathbb{S}\rvert^2 n^5 \log \lvert\mathbb{F}\rvert)$	$O(\lvert\mathbb{S}\rvert^2 n^5 \log \lvert\mathbb{F}\rvert)$	

Table 5.2: Total Analysis of Protocols in Additive PMPC based on [170]

5.5.3 Protocols for MSP-based MPC Scheme [204]

Lampkins and Ostrovsky [204] presented an unconditionally secure MPC protocol based on Monotone Span Program (MSP) secret sharing against any $Q2$-adversary, which has linear communication complexity in the size of multiplicative MSP. We build our MSP-based PMPC protocol on top of their MPC protocol, *without increasing the complexity in terms of the size of MSP, d.* As in Table 5.1, a vector is denoted with bold texts.

Definition 5.1. *$(\mathbb{F}, M, \rho, \boldsymbol{a})$ is called a monotone span program, if $\mathbb{F}$ is a finite field, M is a $d \times e$ matrix over $\mathbb{F}$, $\rho : \{1, 2, ..., d\} \rightarrow \{1, 2, ..., n\}$ is a surjective indexing function for each row of M, and $\boldsymbol{a} \in \mathbb{F}^e \backslash \boldsymbol{0}$ is a (fixed) target vector, where $\boldsymbol{0} = (0, ..., 0) \in \mathbb{F}^e$. $(\mathbb{F}, M, \rho, \boldsymbol{a}, \boldsymbol{r})$ is called a multiplicative MSP, if $(\mathbb{F}, M, \rho, \boldsymbol{a})$ is a MSP and $\boldsymbol{r}$ is a recombination vector, which means the vector $\boldsymbol{r}$ satisfies the property that $\langle \boldsymbol{r}, M\boldsymbol{b} * M\boldsymbol{b}' \rangle = \langle \boldsymbol{a}, \boldsymbol{b} \rangle \cdot \langle \boldsymbol{a}, \boldsymbol{b}' \rangle$, for all $\boldsymbol{b}, \boldsymbol{b}'$, where $*$ is the Hadamard product and $\cdot$ is the inner product.*

The target vector $\mathbf{a}$ can be any vector in $\mathbb{F}^e \backslash \mathbf{0}$; we use $\mathbf{a} = (1, 0, ..., 0)^t \in \mathbb{F}^e$ for convenience, as in [204]. Let $f : \{0, 1\}^n \rightarrow \{0, 1\}$ be a monotone function. A MSP $(\mathbb{F}, M, \rho, \mathbf{a})$ is said to *compute* f if for all nonempty set $A \subset \{1, ..., n\}$, $f(A) = 1 \Leftrightarrow \mathbf{a} \in \mathrm{Im} M_A^t$, i.e.,

$\exists\lambda_A$ such that $M_A^t\lambda_A = \mathbf{a}$. Also, a MSP $(\mathbb{F}, M, \rho, \mathbf{a})$ computing f is said to *accept* Γ if $B \in \Gamma \Leftrightarrow f(B) = 1$. Note that any given MSP computes a monotone Boolean function f, defined $f(x_1, ..., x_n) = 1 \Leftrightarrow \mathbf{a} \in \mathrm{Im}M_A^t$ where $A = \{1 \leq i \leq n | x_i = 1\}$, and it is well known that any monotone Boolean function can be computed by an MSP.

Information Checking

[204] uses a variant of [51] for information checking, which is described below. They use an extension field $\mathbb{G}$ over $\mathbb{F}$ such that the field $\mathbb{G}$ has minimal size satisfying $|\mathbb{G}| \geq d|\mathbb{F}|$, to allow the sender to produce tags for messages of length at most d. Note that κ is a security parameter, P_S is the sender, P_R is the receiver, and P_V is the verifier.

---

[204] Protocol $\texttt{Authenticate}^{\texttt{M}}(P_S, P_R, P_V, w, \mathbf{s}) \longrightarrow (y, z)$ or $\perp$

Input: P_S and P_R both knowing $\mathbf{s}$, P_V, a phase w, and a vector of secret values $\mathbf{s} = (s^{(1)}, ..., s^{(l)}) \in \mathbb{F}^l$ such that $l \leq d$

Output: a pair of tags (y, z), where $y = \{y_i\}_{i=1}^{\kappa}$ is a set of authentication tags and $z = \{z_i\}_{i=1}^{\kappa}$ is a set of verification tags, or aborted $(\perp)$

1. P_S picks 2κ random elements $y_1, ..., y_\kappa, u_1, ..., u_\kappa \in \mathbb{G}$.

2. For each $i = 1, ..., \kappa$, P_S determines v_i such that the $(l+2)$ points, $(0, y_i), (1, s^{(1)}), ..., (l, s^{(l)}), (u_i, v_i)$ lie on a polynomial of degree l over $\mathbb{G}$.

3. P_S sends $y_1, ..., y_\kappa$ to P_R and $z_1, ..., z_\kappa$ to P_V, where $z_i = (u_i, v_i)$ for each i.

4. P_V partitions the set $\{1, ..., \kappa\}$ into two sets I and $\overline{I}$ of almost equal size $(||I| - |\overline{I}|| \leq 1)$, and sends $\{z_i\}_{i \in I}$ to P_R.

5. P_R checks if the $(l+2)$ points, $(0, y_i), (1, s^{(1)}), ..., (l, s^{(l)}), (u_i, v_i)$ lie on a polynomial of degree l, for each z_i. P_R broadcasts NOK if any of these checks fails, or OK, otherwise.

6. Only if P_R broadcasts NOK in Step 5, the following are executed:

a) P_R picks one z_i that failed the check and broadcasts (i, z_i).

b) P_S and P_V broadcast z_i for i received in Step a).

c) Based on the values broadcasted in Step a) and b), a pair $\{P_i, P_j\}$ of parties is added to the dispute list $\mathcal{D}$, where P_i, P_j are two parties over P_S, P_R, P_V such that their broadcasted values are different. The protocol is aborted.

7. Output $\{y_i\}_{i=1}^{\kappa}$ as authentication tags and $\{z_i\}_{i=1}^{\kappa}$ as verification tags.

When $\texttt{Authenticate}^{\texttt{M}}$ succeeds, P_R receives the messages and authentication tags, and P_V receives verification tags, which give no information about the messages. On the other hand, $\texttt{Verify}^{\texttt{M}}$ allows P_V to verify the authenticity of the messages that P_R requested.

---

[204] Protocol $\texttt{Verify}^{\texttt{M}}(P_S, P_R, P_V, w, \mathbf{s}', (y, z)) \longrightarrow \mathbf{s}'$ or $\perp$

Input: a phase w, a candidate vector $\mathbf{s}'$ for $\mathbf{s} = (s^{(1)}, ..., s^{(l)}) \in \mathbb{F}^l$, the authentication tag y P_R that has, and the verification tag z that P_V has

Output: s or $\perp$

1. P_R sends $\mathbf{s}' = (s'^{(1)}, ..., s'^{(l)})$ and authentication tags $y = \{y_i\}_{i \in \overline{I}}$ to P_V.
2. P_V broadcasts OK and outputs $\mathbf{s}'$, if the points $(0, y_i), (1, s^{(1)}), ...,$ $(l, s^{(l)})$, (u_i, v_i) form a polynomial of degree l, for any $i \in \overline{I}$. Otherwise, P_V broadcasts NOK, and the protocol is aborted.

As shown in [204], the communication complexity of $\texttt{Authenticate}^{\texttt{M}}$ is $O(\kappa \log d)$, and the one of $\texttt{Verify}^{\texttt{M}}$ is $O(l + \kappa \log d)$, with negligible error probability less than $\kappa/(d(2^\kappa - 1) - 1)$.

Secret Sharing and Dispute Control

In [204], a dealer generates multiple secrets, and only one pair of authentication and verification tags is generated for the multiple secrets. To fairly compare with the additive SS in which a dealer shares one secret per protocol execution, we present a naturally reduced version of $\texttt{Share}^{\texttt{M}}$ protocol, where a dealer shares one secret per protocol execution. For clarifying, we call the original SS scheme in [204] as $\texttt{ShareMultiple}^{\texttt{M}}$ and $\texttt{ReconstructMultiple}^{\texttt{M}}$ and the reduced version as $\texttt{Share}^{\texttt{M}}$ and $\texttt{Reconstruct}^{\texttt{M}}$. We only present $\texttt{Share}^{\texttt{M}}$ and $\texttt{Reconstruct}^{\texttt{M}}$ protocols below.

The protocol $\texttt{Share}^{\texttt{M}}$ uses a subprotocol called $\texttt{BasicShare}^{\texttt{M}}$ to deal with active adversaries. $\texttt{BasicShare}^{\texttt{M}}$ [95] is a basic secret sharing protocol using an MSP with matrix M of size $d \times e$. After a protocol execution, each party not in dispute with dealing party P_D will get the shares of a secret s, while the parties in dispute with P_D will receive the all-zero vectors as their shares, called *Kudzu share* [51].

[95, 204] Protocol $\texttt{BasicShare}^{\texttt{M}}(w, s) \longrightarrow [s]^w$

Input: a phase w and a secret value $s \in \mathbb{F}$
Output: the sharing of s in phase w

1. A dealing party P_D constructs a vector $\mathbf{b} := (s, r_2, ..., r_e) \in \mathbb{F}^e$, where $r_i \overset{\$}{\leftarrow} \mathbb{F}$ such that all parties in $\mathcal{D}_D$ will receive the all-zero vectors as their shares.

2. P_D computes $\mathbf{s} = M\mathbf{b}$, where M is the MSP corresponding to Δ.

3. P_D sends $\mathbf{s}_j = M_j\mathbf{b}$ to each $P_j \notin \mathcal{D}_D$, where M_j denotes the matrix collecting all the rows assigned to P_j, i.e., all i's such that $\rho(i) = j$.

In $\texttt{Share}^{\texttt{M}}$, one more list $\mathcal{C}$ is used for dispute control. The list $\mathcal{C}$ is a set of parties known by all parties to be corrupted. i.e., The list $\mathcal{D}$ maintains the parties in each dispute list $\mathcal{D}_i$ for all i, and some of them move to the list $\mathcal{C}$ when all parties agree their being corrupted.

---

[204] Protocol $\texttt{Share}^{\texttt{M}}(w, s, \mathcal{P}) \longrightarrow [s]^w$

Input: a phase w, a secret value $s \in \mathbb{F}$, and a group $\mathcal{P}$ of parties receiving shares
Output: the sharing of s in phase w

1. A dealing party P_D chooses n extra random values, $u^{(1)}, ..., u^{(n)}$, then invokes $(n+1)$ $\texttt{BasicShare}^{\texttt{M}}$ (in parallel) for each $\{u^{(i)}\}_{i=1}^n$ and s.

2. For each pair $P_R, P_V \notin \mathcal{D}_D$ such that $\{P_R, P_V\} \notin \mathcal{D}$, $\texttt{Authenticate}^{\texttt{M}}(P_D, P_R, P_V, \mathbf{v}_R)$ is invoked, where $\mathbf{v}_R := (\mathbf{s}_R, \mathbf{u}_R^{(1)}, ..., \mathbf{u}_R^{(n)})$. Note that each $\mathbf{s}_R$ and $\mathbf{u}_R^{(i)}$ is a vector of length d_R so that the length of the vector $\mathbf{v}_R$ is $d_R * (n+1)$.

3. For each $P_V \notin \mathcal{D}_D$, the followings are performed (in parallel):

4. P_V chooses a random vector $r \in \mathbb{F}$ and broadcasts it.

5. Each party $P_i \notin \mathcal{D}_D$ sends his share of $r * \mathbf{s}_i + \mathbf{u}_i^{(V)}$ to P_V. Recall that $\mathbf{s}_i := M_i\mathbf{b}$, where $\mathbf{b}$ is a random vector in $\mathbb{F}^e$ with first component s, and $\mathbf{u}_i^{(V)}$ is similarly defined.

6. If the shares received in Step 5 form a consistent sharing, P_V broadcasts OK. Otherwise, P_V broadcasts NOK. i.e., P_V accepts if the sharing is a vector in the span of the matrix $M_{\mathcal{G}}$, where $\mathcal{G} = \mathcal{P} - \mathcal{C}$.

7. For the lowest P_V who broadcast NOK in Step 6, the following are executed:
a) P_D broadcasts each share of $r * \mathbf{s}_k + \mathbf{u}_k^{(V)}$ for $k = 1, ..., n$.

b-1) If this sharing is not in Span($M_{\mathcal{G}}$), then each party adds P_D to his list $\mathcal{D}_i$, i.e., P_D is added to $\mathcal{C}$, and the protocol is aborted.

b-2) Otherwise, there is a share of some party $P_i \notin \mathcal{D}_D$ which is different from the one broadcasted by P_D. P_V broadcasts ($\texttt{accuse}, P_i, P_D, v_i, v_D$), where v_i is the share sent by P_i and v_D is the value of the share sent by P_D for i-th share.

c) If P_i disagrees with the value v_i broadcasted by P_V, then P_i broadcasts ($\texttt{dispute}, P_i, P_V$) so that the pair $\{P_i, P_V\}$ is added to $\mathcal{D}$, and the protocol is aborted.

d) If P_D disagrees with the value v_D broadcasted by P_V, then P_D broadcasts ($\texttt{dispute}, P_D, P_V$), the pair $\{P_D, P_V\}$ is added to $\mathcal{D}$, and the protocol is aborted.

e) If neither P_i nor P_D complained in the previous steps, then $\{P_i, P_D\}$ is added to $\mathcal{D}$ and the protocol is aborted.

8. Otherwise, the parties in $\mathcal{P}$ collectively output $[s] := \{\mathbf{s}_1, ..., \mathbf{s}_n\}$ with $\{[u^{(i)}]\}_{i=1}^n$.

$\texttt{Reconstruct}^{\texttt{M}}$ allows parties in $\forall B \in \Gamma$ to reconstruct the secret shared using $\texttt{Share}^{\texttt{M}}$.

---

[204] Protocol $\texttt{Reconstruct}^{\texttt{M}}(w, [s]) \longrightarrow s$ or $\perp$

Input: a phase w and a sharing of s (collectively), shared by P_D
Output: s or aborted

1. Each party in $\mathcal{G} := \mathcal{P} \setminus \mathcal{C}$ holding a non-Kudzu share of $[s]$ broadcasts its share.

2. If the shares broadcast in Step 1 and Kudzu shares form a consistent sharing, i.e. they are in Span($M_{\mathcal{G}}$), then the protocol terminates with the output of $\langle \lambda_{\mathcal{G}}, [s]_{\mathcal{G}} \rangle$, where $\lambda_{\mathcal{G}}$ is a vector satisfying $M_{\mathcal{G}}^t \lambda_{\mathcal{G}} = \mathbf{a}$ and $[s]_{\mathcal{G}}$ is the recomposition vector with respect to the indexing function ρ with all the shares of parties in $\mathcal{G}$.

3. If the shares broadcast in Step 1 are inconsistent, i.e., not in Span($M_{\mathcal{G}}$), then P_D broadcasts the index i of each party he accuses of sending an incorrect share.

4. If P_D did not broadcast an index in Step 3, or if the remaining shares after removing the shares that P_D accused are still inconsistent, or if the set of parties in dispute with P_D is no longer in Δ, then P_D is added to $\mathcal{C}$.

5. If $P_D \notin \mathcal{C}$, do the following:

5.a) For each party P_i accused by P_D in Step 3, parties invoke $\texttt{Verify}(P_D, P_i, P_k, w, \mathbf{v}_i, tags)$ for each party $P_k \notin \mathcal{D}_i \cup \mathcal{D}_j$, where $\mathbf{v}_i = (\mathbf{s}_i, \mathbf{u}_i^{(1)}, ..., \mathbf{u}_i^{(n)})$ as defined in Step 2 of $\texttt{Share}^{\texttt{M}}$.

5.b) For any P_i who sent a share to P_k that was different than the share broadcast in Step 1, P_k broadcasts ($\texttt{accuse}, i$) and $\{P_k, P_i\}$ is added to the list $\mathcal{D}$.

5.c) If $P_k \notin \mathcal{D}_i$ rejects in Step 5.a, then $\{P_k, P_j\}$ is added to the list $\mathcal{D}$. Otherwise, $\{P_D, P_k\}$ is added to $\mathcal{D}$.

5.d) If the shares of parties not in $\mathcal{C}$ (after some parties are added to $\mathcal{C}$) and the Kudzu shares form a consistent sharing, then those shares are used to reconstruct s. Otherwise, P_D is added to $\mathcal{C}$ and proceed to Step 6.

6. If $P_D \in \mathcal{C}$, do the following:

6.a) For all P_j holding non-Kudzu shares and for all $P_k \notin \mathcal{D}_j$, the parties invoke $\texttt{Verify}^{\texttt{M}}(P_D, P_j, P_k,$

$w, \mathbf{v}_j, tags)$, where $\mathbf{v}_j = (\mathbf{s}_j, \mathbf{u}_j^{(1)}, ..., \mathbf{u}_j^{(n)})$.

6.b) For any P_j who sent a share to P_k that is different than the share broadcast in Step 1, P_k broadcasts $(\texttt{accuse}, j)$ and $\{P_k, P_j\}$ is added to the list $\mathcal{D}$.

6.c) The shares of parties not in $\mathcal{C}$ are used to reconstruct s as in Step 2.

In Step 2, the correctness holds when $B \in \Gamma \Leftrightarrow f(B) = 1$, which means there is some vector λ_B such that $M_B^t \lambda_B = \mathbf{a}$. Therefore, $\langle \lambda_B, [s]_B \rangle = \langle \lambda_B, M_B \mathbf{b} \rangle = \langle M_B^t \lambda_B, \mathbf{b} \rangle = \langle \mathbf{a}, \mathbf{b} \rangle = s$, as $\mathbf{a} = (1, 0, ..., 0)$ and $\mathbf{b} = (s, r_2, ..., r_e)$. Throughout the steps in `Reconstruct`, parties can detect all the potentially corrupted parties. Note that parties cannot reconstruct the secret value when the remaining parties in $\mathcal{G}$ are no longer in Γ.

For `LC-Reconstruct`$^{\texttt{M}}$, we add more explanations about the assumptions and how this protocol works in detail in addition to the one in [204] as it is not trivial. Intuitively, each party first broadcasts its share of q and reconstructs the value q if all broadcast shares are consistent. However, if they are inconsistent, they divide it into small chunks and see which parties sent the wrong values. Since the IC scheme does not satisfy the linearity, parties holding tags of two shared secrets cannot locally compute the right tag for the linear computation of two secret values. To be specific, authentication tags can be computed locally by adding two existing authentication tags because an authentication tag is defined as the y-intercept of a function. However, verification tags cannot be computed locally as the X coordinate value of each tag is randomly chosen. Hence, the probability of having the same X coordinate values for two verification tags is very low. That is, even though one party knows two verification tags (u, v) and (u', v') for function (of same degree) f and f', respectively, $(u + u', v + v')$ is not on $f + f'$ and he cannot locally compute $(u, v + f'(u))$ or $(u', f(u') + v')$ without knowing f or f'. For these reasons, the protocol `LC-Reconstruct` uses the “divide-and-conquer" method to find the parties who sent the wrong shares when the shares of q are inconsistent.

Now, let us see the assumptions and settings of this protocol. Assume that each party P_j shared l_j secrets, $s^{(j,1)}, s^{(j,2)}, ..., s^{(j,l_j)}$, and parties want to compute the total summation of multiple linear combinations of these l_j secrets for each P_j. i.e., parties in $\mathcal{P}$ want to reconstruct the value q, where $q := q^{(1)} + ... + q^{(n)}$ and $q^{(j)} := \sum_{i=1}^{l_j} a_i^{(j)} s^{(j,i)}$ for some $a_i^{(j)} \in \mathbb{F}$, $i = 1, ..., l_j$ for each $j = 1, ..., m$. Note that m can be up to n. For instance, if P_1 shares l_1 secret values, $s^{(1,1)}, ..., s^{(1,l_1)}$, P_2 shares l_2 secret values, $s^{(2,1)}, ..., s^{(2,l_2)}$, and P_3 shares l_3 secret values, $s^{(3,1)}, ..., s^{(3,l_3)}$ (i.e., $j = 3$), then we are assuming that the parties in $\mathcal{P} = \{P_1, ..., P_n\}$ want to reconstruct $q = q^{(1)} + q^{(2)} + q^{(3)}$, where $q^{(1)} := \sum_{i=1}^{l_1} a_i^{(1)} s^{(1,i)}$, $q^{(2)} := \sum_{i=1}^{l_2} a_i^{(2)} s^{(2,i)}$, and $q^{(3)} := \sum_{i=1}^{l_3} a_i^{(3)} s^{(3,i)}$.

---

[204] Protocol $\texttt{LC-Reconstruct}^{\texttt{M}}(w, [q]) \longrightarrow q$ or $\perp$

Input: a phase w and locally computed sharing of q (collectively), $[q] = \{\mathbf{q}_1, ..., \mathbf{q}_n\}$, where $q = q^{(1)} + ... + q^{(n)}$ and each $q^{(j)}$ is a linear combination of l_j secrets shared by P_j, i.e., $q^{(j)} := \sum_{i=1}^{l_j} a_i^{(j)} s^{(j,i)}$ for some $a_i^{(j)} \in \mathbb{F}$ and $\{s^{(j,i)}\}_{i=1}^{l_j}$: secrets shared by P_j. Note that $[s^{(j,i)}] = \{\mathbf{s}_1^{\mathbf{(j,i)}}, ..., \mathbf{s}_n^{\mathbf{(j,i)}}\}$ is a sharing of i-th secret that P_j shared.

Output: q or aborted

1. Each $P_i \notin \mathcal{C}$ broadcasts its share $\mathbf{q}_i$ of $[q]$.

2. a) If the sharing broadcast in Step 1 is consistent (i.e., in Span($M_{\mathcal{G}}$)), then q is reconstructed by $\langle \lambda_{\mathcal{G}}, [q]_{\mathcal{G}} \rangle$ and the protocol terminates, where $[q]_{\mathcal{G}}$ is the recomposition vector with all the shares $\mathbf{q}_i$'s for $P_i \in \mathcal{G}$.

b) Otherwise, each $P_i \notin \mathcal{C}$ broadcasts its share $\mathbf{q}_i^{(j)}$ of $[q^{(j)}]$, for each P_j.

3. a) If any party P_i broadcasted values such that $\mathbf{q}_i \neq \sum_{j=1}^{n} \mathbf{q}_i^{(j)}$, then all such parties are added to $\mathcal{C}$ and the protocol terminates.

b) Otherwise, for the lowest j such that the shares of $[q^{(j)}]$ broadcasted in Step 2.b are inconsistent, do one of the followings depending on $P_j \notin \mathcal{C}$ or not.

4. If $P_j \notin \mathcal{C}$, do the followings:

a) P_j broadcasts ($\texttt{accuse}, i$) for P_i he thinks to have sent an incorrect share.

b) Since $[q^{(j)}]$ is a linear combination of sharings generated by P_j, the parties internally know that $[q^{(j)}] = \sum_{k=1}^{l_j} a_k^{(j)} [s^{(j,k)}]$, where each $[s^{(j,k)}]$ was generated with $\texttt{Share}$ and each $a_k^{(j)}$ is non-zero. From l_j sharings $a_1^{(j)}[s^{(j,1)}], a_2^{(j)}[s^{(j,2)}], ..., a_{l_j}^{(j)}[s^{(j,l_j)}]$, P_i accused in Step 4.a broadcasts his shares of $\sum_{k=1}^{\lfloor l_j/2 \rfloor} a_k^{(j)} [s^{(j,k)}]$ and $\sum_{k=\lfloor l_j/2 \rfloor+1}^{l_j} a_k^{(j)} [s^{(j,k)}]$, i.e., $\sum_{k=1}^{\lfloor l_j/2 \rfloor} a_k^{(j)} \mathbf{s}_i^{\mathbf{(j,k)}}$ and $\sum_{k=\lfloor l_j/2 \rfloor+1}^{l_j} a_k^{(j)} \mathbf{s}_i^{\mathbf{(j,k)}}$.

c) If P_i's two broadcasted shares in Step 4.b do not match up with the previously sent share of their sum (e.g., $\mathbf{q}_i^{(j)}$ for the first round), then P_i is added to $\mathcal{C}$ and the protocol terminates.

d) P_j broadcasts which of shares broadcasted in Step 4.b he disagrees with. If this is a single sharing $a_k^{(j)}[s^{(j,k)}]$, then parties proceed to Step 4.e. Otherwise, parties return to Step 4.b with the sharings P_j disagreed with. i.e., If P_j disagrees with some sum $a_{k_1}^{(j)}[s^{(j,k_1)}] + ... + a_{k_2}^{(j)}[s^{(j,k_2)}]$, then parties repeat Step 4.b to Step 4.d with $a_{k_1}^{(j)}[s^{(j,k_1)}], ..., a_{k_2}^{(j)}[s^{(j,k_2)}]$ instead of $a_1^{(j)}[s^{(j,1)}], ..., a_{l_j}^{(j)}[s^{(j,l_j)}]$.

e) At this point, P_i broadcasted its share $a_k^{(j)}\mathbf{s}_i^{\mathbf{(j,k)}}$ of $a_k^{(j)}[s^{(j,k)}]$ for some k and P_j broadcasted that he disagrees with this share. For each $P_V \notin \mathcal{D}_j \cup \mathcal{D}_i$, parties invoke $\texttt{Verify}^{\texttt{M}}(P_j, P_i, P_V, w, \mathbf{v}_i, tags)$, where $\mathbf{v}_i = (\mathbf{s}_i^{(j,k)}, \mathbf{u}_i^{(1)}, ..., \mathbf{u}_i^{(n)})$ as in Step 2 in $\texttt{Share}^{\texttt{M}}$ protocol.

f) If the shares sent from P_i to P_V in $\texttt{Verify}^{\texttt{M}}$ do not match with the share of $a_k^{(j)}[s^{(j,k)}]$, then P_V broadcasts $(\texttt{accuse}, i)$, and $\{P_i, P_V\}$ is added to $\mathcal{D}$.

g) $\{P_i, P_V\}$ is added to $\mathcal{D}$ for each $P_V \notin \mathcal{D}_i$ who rejected in the invocation of $\texttt{Verify}^{\texttt{M}}$ in Step 4.e, or $\{P_j, P_V\}$ is added to $\mathcal{D}$ for each P_V who accepted it.

h) At this point, all parties are in dispute with either P_i or P_j and by the Q^2 property of Δ, one of $\mathcal{D}_i$ or $\mathcal{D}_j$ is no longer in Δ. If $\mathcal{D}_i \notin \Delta$, P_i is added to $\mathcal{C}$, and if $\mathcal{D}_j \notin \Delta$, P_j is added to $\mathcal{C}$. Then, the protocol terminates.

5. If $P_j \in \mathcal{C}$, do the followings:

a) Since $[q^{(j)}]$ is a linear combination of sharings generated by P_j, the parties internally know that $[q^{(j)}] = \sum_{k=1}^{l_j} a_k^{(j)}[s^{(j,k)}]$, where each $[s^{(j,k)}]$ was generated with $\texttt{Share}^{\texttt{M}}$ and each $a_k^{(j)}$ is non-zero. From l_j sharings $a_1^{(j)}[s^{(j,1)}], a_2^{(j)}[s^{(j,2)}], ..., a_{l_j}^{(j)}[s^{(j,l_j)}]$, each party $P_i \notin \mathcal{C}$ broadcasts its shares of $\sum_{k=1}^{\lfloor l_j/2 \rfloor} a_k^{(j)}[s^{(j,k)}]$ and $\sum_{k=\lfloor l_j/2 \rfloor+1}^{l_j} a_k^{(j)}[s^{(j,k)}]$, i.e., $\sum_{k=1}^{\lfloor l_j/2 \rfloor} a_k^{(j)}\mathbf{s}_i^{\mathbf{(j,k)}}$ and $\sum_{k=\lfloor l_j/2 \rfloor+1}^{l_j} a_k^{(j)}\mathbf{s}_i^{\mathbf{(j,k)}}$.

b) Any party P_i whose sum of two broadcasted shares in Step 5.a does not match up with the previously sent share of their sum (e.g. $\mathbf{q}_i^{(j)}$ for the first round) is added to $\mathcal{C}$ and the protocol terminates.

c) At this point, one of the two shares broadcasted in Step 5.a is inconsistent. If this is a single sharing $a_k^{(j)}[s^{(j,k)}]$, then parties proceed Step 5.d. Otherwise, if this is some sum $a_{k_1}^{(j)}[s^{(j,k_1)}] + ... + a_{k_2}^{(j)}[s^{(j,k_2)}]$, then parties return to Step 5.a with $a_{k_1}^{(j)}[s^{(j,k_1)}], ..., a_{k_2}^{(j)}[s^{(j,k_2)}]$ instead of $a_1^{(j)}[s^{(j,1)}], ..., a_{l_j}^{(j)}[s^{(j,l_j)}]$.

d) Parties invoke $\texttt{Reconstruct}^{\texttt{M}}(w, [s^{(j,k)}])$ for the single sharing $a_k^{(j)}[s^{(j,k)}]$ decided in Step 5.c, but skip Step 1, as they already broadcasted their shares. As a result of $\texttt{Reconstruct}^{\texttt{M}}$, a new party is added to $\mathcal{C}$ and the protocol terminates.

Note that if m parties only shared one secret $s^{(i,1)}$ for each party P_i, then each $\mathbf{q}_i^{(j)}$ of $[q^{(j)}]$ is already a single sharing, i.e., $q = q^{(1)} + ... + q^{(m)}$ where $q^{(j)} = a^{(j)}s^{(j,1)}$. Therefore, they can directly jump to Step 4.e or Step 5.d according to whether $P_j \notin \mathcal{C}$ or not.

Addition and Multiplication

By the linearity of the shares, addition can be done naturally without any communication or broadcast as below.

[204] Protocol $\texttt{Add}^{\texttt{M}}(w, [s], [t]) \longrightarrow [s+t]$

Input: phase w, shares of s, and shares of t
Output: new shares of $s+t$
Precondition: Two values s and t are shared with $\texttt{Share}^{\texttt{M}}$
Postcondition: $s+t$ is shared independently

1. Each party P_i locally adds each share of s to the share of t and keep the result as a share of $s+t$. i.e., $(\mathbf{s}+\mathbf{t})_i := \mathbf{s}_i + \mathbf{t}_i \in \mathbb{F}^{d_i}$, for each $i = 1, ..., n$.

The protocol $\texttt{Generate-Randomness}^{\texttt{M}}$ generates l random elements, which are publicly known in $\mathcal{P}$. This is used in $\texttt{Generate-Multiplication-Triples}^{\texttt{M}}$ protocol for error detection.

[204] Protocol $\texttt{Generate-Randomness}^{\texttt{M}}(w, l) \longrightarrow r^{(1)}, ..., r^{(l)}$

Input: a phase w and the non-negative integer l
Output: publicly known l random elements in $\mathbb{F}$

1. Every party $P_i \notin \mathcal{C}$ chooses l random values $r^{(1,i)}, ..., r^{(l,i)}$.
2. Each P_i invokes $\texttt{ShareMultiple}^{\texttt{M}}(w, \mathbf{r}, P_i)$, where $\mathbf{r} := (r^{(1,i)}, ..., r^{(l,i)})$ to verifiably share these l random values.
3. Parties in $\mathcal{P}$ call $\texttt{LC-Reconstruct}^{\texttt{M}}(w, [r^{(j)}])$ l times in parallel, to reconstruct l random values, $r^{(1)}, ..., r^{(l)}$, where $r^{(j)} := \sum_{P_i \notin \mathcal{C}} r^{(j,i)}$.

For multiplication gates, the protocol $\texttt{Generate-Multiplication-Triples}^{\texttt{M}}$ generates random sharings of l multiplication triples (a, b, c) such that $c = ab$, without revealing any values of a, b, or c to parties. These random triples can be used in each multiplication gate by computing $[st] := [c] + [s](t-b) + [t](s-a) - (s-a)(t-b)$ as in [50]. To generate a sharing of a random triple $(a^{(k)}, b^{(k)}, c^{(k)})$, a random element $a^{(k)}$ is generated and each P_i creates a random triple $a^{(k)} b^{(i,k)} = c^{(i,k)}$. After verifying each triple's correctness using a triple $a^{(k)} \widetilde{b}^{(i,k)} = \widetilde{c}^{(i,k)}$ also created by each P_i, the final triple is defined as $(a^{(k)}, \sum_{i=1}^{n} b^{(i,k)}, \sum_{i=1}^{n} c^{(i,k)})$ for each $k = 1, ..., l$. For simplicity, we present the reduced version, which generates a sharing of only one multiplication triple, say (a, b, c).

---

[204] Protocol $\texttt{Generate-Multiplication-Triples}^{\texttt{M}}(w) \longrightarrow [(a, b, c)]$

Input: phase w

Output: a sharing of random triple (a, b, c) such that $c = ab$

1. Each $P_i \notin \mathcal{C}$ invokes $\texttt{Share}^{\texttt{M}}$ $(2n+3)$ times (in parallel), for each $a^{(i)}, b^{(i)}, \widetilde{b}^{(i)}$, $\{r^{(i,j)}\}_{j=1}^{n}$, and $\{\widetilde{r}^{(i,j)}\}_{j=1}^{n}$, and $\texttt{Share}^{\texttt{M}}$ $2n$ times for generating sharings of 1 (in parallel), denoted by $\{1^{(i,j)}\}_{j=1}^{n}$ and $\{\widetilde{1}^{(i,j)}\}_{j=1}^{n}$. The sharings of parties in $\mathcal{C}$ are defined to be all-zero sharings.

2. Each party defines and locally computes $[a] := \sum_{m=1}^{n}[a^{(m)}]$, $[r^{(i)}] := \sum_{m=1}^{n}[r^{(i,m)}]$, $[1^{(i)}] := \sum_{m=1}^{n}[1^{(m,i)}] + w[1^{(i,i)}]$, and $[\widetilde{1}^{(i)}] := \sum_{m=1}^{n}[\widetilde{1}^{(m,i)}] + \widetilde{w}[\widetilde{1}^{(i,i)}]$, where each w and $\widetilde{w} \in \mathbb{F}$ is the unique element that makes $[1^{(i)}]$ and $[\widetilde{1}^{(i)}]$ a sharing of 1.

3. Each $P_j \notin \mathcal{C}$ sends its share of $[a][b^{(i)}] + [r^{(i)}][1^{(i)}]$ and $[a][\widetilde{b}^{(i)}] + [\widetilde{r}^{(i)}][\widetilde{1}^{(i)}]$ to $P_i \notin \mathcal{C}$.

4. Each $P_i \notin \mathcal{C}$ reconstructs $D^{(i)} := ab^{(i)} + r^{(i)}$ and $\widetilde{D}^{(i)} := a\widetilde{b}^{(i)} + \widetilde{r}^{(i)}$ with the shares received in Step 3, and broadcasts $D^{(i)}$ and $\widetilde{D}^{(i)}$.

5. Each party locally computes $[c^{(i)}] := D^{(i)} - [r^{(i)}]$ and $[\widetilde{c}^{(i)}] := \widetilde{D}^{(i)} - [\widetilde{r}^{(i)}]$.

6. parties invoke $\texttt{Generate-Randomness}^{\texttt{M}}(w, 1)$ to generate a random element s.

7. Each party $\notin \mathcal{D}_i$ broadcasts its share of $[\widehat{b}^{(i)}] := [\widetilde{b}^{(i)}] + s[b^{(i)}]$, for $i = 1, ..., n$.

8. If the sharing of some $[\widehat{b}^{(i)}]$ broadcast in Step 7 is inconsistent, P_i broadcasts $(\texttt{accuse}, P_j)$ for such sharing sent by $P_j \notin \mathcal{D}_i$. $\{P_i, P_j\}$ is added to $\mathcal{D}$ and the protocol terminates.

9. parties invoke $\texttt{LC-Reconstruct}^{\texttt{M}}$ n times (in parallel) to reconstruct $z^{(i)} := [a]\widehat{b}^{(i)} - [\widetilde{c}^{(i)}] - s[c^{(i)}]$, for $i = 1, ..., n$.

10. If all reconstructed values in Step 9 are zero, then the protocol terminates successfully with the triple (a, b, c) with $[b] := \sum_{m=1}^{n}[b^{(m)}]$ and $[c] := \sum_{m=1}^{n}[c^{(m)}]$. Otherwise, if any $z^{(i)}$ is non-zero, then proceed into Step 11 for the lowest index i such that $z^{(i)} \neq 0$.

11. a) Each P_j broadcasts its share of $[a^{(m)}], [\widetilde{r}^{(m,i)}]$, and $[r^{(m,i)}]$ for each $P_m \notin \mathcal{D}_j$.

b) If P_i sees that the shares of some $P_j \notin \mathcal{D}_i$ sent in Step 11.a are inconsistent with the share sent in Step 3 or 9, then P_i broadcasts $(\texttt{accuse}, P_j)$ and $\{P_i, P_j\}$ is added to $\mathcal{D}$ and the protocol terminates.

c) Each P_m examines the shares (broadcast in Step 11.a) of all sharings that P_m generated. If P_m notices that some $P_j \notin \mathcal{D}_m$ broadcast an incorrect share, then P_m broadcasts $(\texttt{accuse}, P_j)$ and $\{P_m, P_j\}$ is added to $\mathcal{D}$ and the protocol terminates.

d) If no one broadcasts, then P_i is added to $\mathcal{C}$ and the protocol terminates.

5.5.4 MSP-based PMPC Scheme for Dynamic GAS and Groups

Similarly, we build $\texttt{Refresh}^{\texttt{M}}$, $\texttt{Recover}^{\texttt{M}}$, and $\texttt{Redistribute}^{\texttt{M}}$ protocols to make this MSP-based MPC scheme to be proactively secure with dynamic groups. Recall that the protocol $\texttt{Refresh}^{\texttt{M}}$ re-randomizes each party's shares regularly so that the adversary cannot reconstruct the secret until he corrupts any set in the access structure in the period. By the linearity of the shares, the main idea is the same as before.

Protocol $\texttt{Refresh}^{\texttt{M}}(w, [s]) \longrightarrow [s]^{w+1}$

Input: a phase w and a sharing of s
Output: new sharing of s in phase $w+1$, $[s]^{w+1}$

1. Every party P_i in $\mathcal{P}$ invokes $\texttt{Share}^{\texttt{M}}(w, 0, P_i)$. (in parallel)
2. Each party locally does component-wise addition with all the shares received in Step 1 and the shares of s, and set it as the new share of s in phase $w+1$.
3. parties in $\mathcal{P}$ collectively output $[s]^{w+1}$.

Theorem 5.4. *(Correctness and Secrecy of* $\texttt{Refresh}^{\texttt{M}}$*) When the protocol* $\texttt{Refresh}^{\texttt{M}}$ *terminates, all parties receive new shares encoding the same secret as old shares they had before, and they get no information about the secret by the protocol execution. It communicates* $O((n^2d + n^3\kappa)\log|\mathbb{F}| + n^3\kappa\log d)$ *bits and broadcasts* $O(n^3\log d + (n^3 + nd)\log|\mathbb{F}|)$ *bits.*

Proof. **Correctness:** Recall that each party P_i has the vector $\mathbf{s}_i = M_i\mathbf{b} \in \mathbb{F}^{d_i}$ as the share of s, where $\mathbf{b} = (s, r_2, ..., r_e)$ for some random values r_j's. After Step 1, every party P_i receives n vectors $\{\mathbf{0}_i^{(j)}\}_{j=1}^n$ as shares of 0's, where $\mathbf{0}_i^{(j)} = M_i\mathbf{b}^{(j)}$ is the share of 0 from each party P_j, where $\mathbf{b}^{(j)} = (0, \$, ..., \$) \in \mathbb{F}^{d_i}$ with some random values (denoted by \$). Since these n vectors have the same lengths d_i, each party P_i can locally compute the vector addition $\mathbf{s}'_i := \mathbf{s}_i + \sum_{j=1}^n \mathbf{0}_i^{(j)} \in \mathbb{F}^{d_i}$. As all the summands of s and n zeros are shared by $\texttt{Share}^{\texttt{M}}$, for $s' = s + 0 + ... + 0$, the invocation of $\texttt{LC-Reconstruct}^{\texttt{M}}(w, [s'])$ with $[s'] := \{\mathbf{s}'_1, ..., \mathbf{s}'_n\}$

outputs s', which is equal to s. **Secrecy:** parties communicate only the shares of zeros but nothing about the secret s or the shares of, the protocol does not reveal any information about s. **Communication:** As every party shares zero to each other, they communicate and broadcast $n * Cost(\texttt{Share}^{\texttt{M}})$ bits. $\square$

For $\texttt{Recover}^{\texttt{M}}$ and $\texttt{Redistribute}^{\texttt{M}}$, we construct two sub-protocols: $\texttt{ShareRandom}^{\texttt{M}}$ and $\texttt{RobustReshare}^{\texttt{M}}$. The goals of the protocols are similar to the ones in Section 5.5.2. Still, due to the fact that each party holds the unique share of a secret, $\texttt{ShareRandom}^{\texttt{M}}$ can be generalized for multiple groups of parties, which enables to build the efficient $\texttt{Redistribute}^{\texttt{M}}$ protocol. The protocol $\texttt{ShareRandom}^{\texttt{M}}$ allows participating parties to generate multiple sharings of a random value $r \in \mathbb{F}$ for each group without reconstructing the value r. Note that $W = \{w\}$ for $\texttt{Recover}^{\texttt{M}}$, while $W = \{w, w+1\}$ for $\texttt{Redistribute}^{\texttt{M}}$. The protocol $\texttt{ShareRandom}^{\texttt{M}}$ outputs $|W|$ sharings of the same r, where r is the summation of all random elements from each party in each phase. For instance, when $W = \{w\}$, the output is one sharing of r, say $[r] = \{\mathbf{r}_1, ..., \mathbf{r}_n\}$, where $\texttt{LC-Reconstruct}^{\texttt{M}}(w, [r])$ reconstructs $r = \sum_{P_i \notin \mathcal{C}} r^{(i)}$. We denote $\texttt{ShareRandom}^{\texttt{M}}(w)$ in this case. On the other hand, when $W = \{w, w+1\}$, it outputs two sharings of r, $[r]^w := \{\mathbf{r}_1^w, ..., \mathbf{r}_n^w\}$ and $[r]^{w+1} := \{\mathbf{r}_1^{w+1}, ..., \mathbf{r}_m^{w+1}\}$, where both sharings reconstruct the same r. i.e., $\texttt{LC-Reconstruct}^{\texttt{M}}(w, [r]^w) = \texttt{LC-Reconstruct}^{\texttt{M}}(w+1, [r]^{w+1}) = r$, where $r = \sum_{P_i \notin \mathcal{C}^w} r^{(w,i)} + \sum_{P_j \notin \mathcal{C}^{w+1}} r^{(w+1,j)}$.

Protocol $\texttt{ShareRandom}^{\texttt{M}}(W) \longrightarrow \{[r]^w\}_{w \in W}$

```
   Input: a list $W$ of phases where participating parties generate sharing(s) of a random value
   Output: $|W|$ sharing(s) of a random value $r$, for each $\mathcal{P}^w$ in $w \in W$
1. For each $w \in W$:
2.     Every party $P_i \notin \mathcal{C}^w$ chooses a random value $r^{(w,i)}$ and invokes $\texttt{Share}^{\texttt{M}}(w', r^{(w,i)}, \mathcal{P}^{w'})$ $|W|$
       times in parallel with respect to $\mathbb{S}^w$, for each $w' \in W$.
3. For each $w \in W$:
4.     Each party $P_i \in \mathcal{P}^w$ locally computes $\mathbf{r}_i^w := \sum_{w' \in W} \sum_{P_j \notin \mathcal{C}^{w'}} [r^{(w',j)}]_i^w$, where $[r^{(w',j)}]_i^w$ is $P_i$'s
```

holding share of $r^{(w',j)}$ received in Step 2 from $P_j \notin \mathcal{C}^{w'}$.

5. $|W|$ sharings of r, $\{[r]^w\}_{w \in W}$, are collectively output, where $r := \sum_{P_j \notin \mathcal{C}^w, w \in W} r^{(w,j)}$ and $[r]^w := \{\mathbf{r}^w_1, ..., \mathbf{r}^w_{|\mathcal{P}^w|}\}$.

Note that all summand vectors $\{[r^{(w',j)}]^w_i\}$ have the same lengths for each party. For $N := \sum_{w \in W} |\mathcal{P}^w|$, the protocol communicates $O(N|W|((nd + n^2\kappa)\log|\mathbb{F}| + n^2\kappa \log d))$ bits and broadcasts $O(N|W|(n^2 \log d + (n^2 + d)\log|\mathbb{F}|))$ bits.

Recall that $Honest := \{\mathcal{P} \setminus A \mid A \in \overline{\Delta}\}$ is a set of potential honest parties sets.The protocol RobustReshare$^{\mathtt{M}}$ similarly works as the one in Section 5.5.2. Every party in $\mathcal{P}_S \subseteq \mathcal{P}^{w_S}$ in phase w_S knows the value r and wants to send a right sharing of r to the parties in $\mathcal{P}_R \subseteq \mathcal{P}^{w_R}$ in phase w_R. As the adversary picks one subset of parties in Δ in each phase, at least one set in $Honest$ consists of only honest parties in that phase.

Protocol RobustReshare$^{\mathtt{M}}(r, w_S, \mathcal{P}_S, w_R, \mathcal{P}_R) \longrightarrow [r]^{w_R}$

Input: a random element $r \in \mathbb{F}$, a phase w_S, a set of parties $\mathcal{P}_S$ in phase w_S, a phase w_R, and a set of parties $\mathcal{P}_R \in \Gamma$ in phase w_R
Output: a sharing of r in phase w_R, $[r]^{w_R}$
Precondition: All parties in $\mathcal{P}_S$ know the value of r.
Postcondition: Each party in $\mathcal{P}_R$ receives the share of new sharing of r.

1. Every party in $\mathcal{P}_S$ executes Share$^{\mathtt{M}}(w_R, r, \mathcal{P}_R)$ according to $\mathbb{S}_R$. Let $[r]^{(i)}$ be the sharing of r that $P_{k_i} \in \mathcal{P}_S$ shares.

2. For each $i = 1, 2, ..., |\mathcal{P}_S|$, Reconstruct$^{\mathtt{M}}(w_R, [r]^{(i)}, \mathcal{P}_R)$ is invoked. Let $r^{(i)}$ be the result of each reconstruction.

3. Choose a value v such that $v = r^{(i)}$ for all $P_{k_i} \in H$, for some $H \in Honest$. Each party chooses such set $H \in Honest$. If multiple such sets exist, the minimal set including P_i with lower id, i, is chosen.

4. Parties in $\mathcal{P}_R$ collectively outputs the sharing of r from the party P_i in H with the minimum id, i. i.e. Output $[r] \leftarrow [r]^{(min)}$, where $min := \min_{P_i \in H}\{i\}$.

Security of RobustReshare$^{\mathtt{M}}$ relies on the security of Share$^{\mathtt{M}}$ and Reconstruct$^{\mathtt{M}}$ and communicates $O(|\mathcal{P}_S|(|\mathcal{P}_R|^2\,\kappa\log d_R+(|\mathcal{P}_R|^3+|\mathcal{P}_R|^2\kappa+|\mathcal{P}_R|d_R)\log|\mathbb{F}|))$ and broadcasts $O(|\mathcal{P}_S|(|\mathcal{P}_R|^2\log d_R+(|\mathcal{P}_R|^2+d_R)\log|\mathbb{F}|))$ bits.

Using these sub-protocols, $\texttt{Recover}^{\texttt{M}}$ allows the rebooted/reset parties to recover their shares by generating new sharing of the same secret in $\mathcal{P}$ with the assistance of other parties. A sharing of a random element r is generated using $\texttt{ShareRandom}^{\texttt{M}}$, $\texttt{LC-Reconstruct}^{\texttt{M}}$ allows every party to reconstruct a publicly known random value $r' := r+s$, and $\texttt{RobustReshare}^{\texttt{M}}$ helps parties to set one same sharing of r'.

Protocol $\texttt{Recover}^{\texttt{M}}(w,[s],R)\longrightarrow[s]^{w+1}$ or $\perp$

Input: a phase w, a sharing of s, and a set of rebooted parties R
Output: new sharing of s in phase $w+1$, $[s]^{w+1}$, or aborted

1. Invoke $\texttt{ShareRandom}^{\texttt{M}}(w)$ and generate a sharing $[r] := \{\mathbf{r}_1,...,\mathbf{r}_n\}$ of a random r in $\mathbb{F}$.
2. Each party P_i in $\mathcal{P}\setminus R$ locally computes $\mathbf{r}_i+\mathbf{s}_i$, the share of $r' := r+s$.
3. $\texttt{LC-Reconstruct}^{\texttt{M}}(w,[r'])$ is invoked in $\mathcal{P}\setminus R$ and every party in $\mathcal{P}\setminus R$ gets r'.
4. $\texttt{RobustReshare}^{\texttt{M}}(r',w,\mathcal{P}\setminus R,w,\mathcal{P})$ is invoked, and each party in $\mathcal{P}$ gets $[r']^w := \{\mathbf{r}'_1,...,\mathbf{r}'_n\}$.
5. Each party locally computes $\mathbf{r}'_i-\mathbf{r}_i$ and sets it as new share of s.

Theorem 5.5. *(Correctness and Secrecy of* $\texttt{Recover}^{\texttt{M}}$*) The protocol* $\texttt{Recover}^{\texttt{M}}$ *allows a set R of parties who were rebooted to recover their shares encoding the same secret for any $R\in\Delta$, and does not reveal any additional information about the secret except the shares each party had before the execution of the protocol. It communicates $O(n^3\kappa\log d+(n^4+n^3\kappa+n^2d)\log|\mathbb{F}|)$ bits and broadcasts $O(n^3\log d+(n^3+nd)\log|\mathbb{F}|)$ bits.*

Proof. **Correctness:** For each party, as the length of its share is the same as the number of rows in M mapped to that party, the party can locally compute component-wise addition/subtraction with its shares. By the linearity of the shares, $\mathbf{r}'_i-\mathbf{r}_i$ is the i-th share of $r'-r=(r+s)-r=s$ for each P_i. Thus, all parties in $\mathcal{P}$, including parties in R, receive a new sharing of s, the same secret. **Secrecy:** In Step 3, every party receives the reconstruction result of $r' = r + s$, but as r is random in $\mathbb{F}$ and not reconstructed in $\mathcal{P}$, each party has any information about r. Thus, the value r' does not reveal any information about s in Step 3. Steps 1 and 4 are to share a sharing of the random elements, and Steps 2 and 5 are local computations. Therefore, executing this protocol does not reveal any information about the secret s. **Communication:** The protocol $\texttt{ShareRandom}^{\texttt{M}}$ is invoked by the group of parties in phase w, where $W = \{w\}$. i.e. $N = n$ and $|W| = 1$. In $\texttt{LC-Reconstruct}^{\texttt{M}}$, each parties only have $l = 1$ secret values to share. In $\texttt{RobustReshare}^{\texttt{M}}$, $\mathcal{P}_S = \mathcal{P} \setminus R$ and $\mathcal{P}_R = \mathcal{P}$ in the same phase w. Thus, the total number of communication bits is $O(n^3\kappa \log d + (n^4 + n^3\kappa + n^2 d) \log |\mathbb{F}|$ and the total number of broadcast bits is $O(n^3 \log d + (n^3 + nd) \log |\mathbb{F}|)$. □

Assuming the dynamic settings in Figure 5.1, recall that the protocol $\texttt{Redistribute}^{\texttt{M}}$ allows parties in the new group $\mathcal{P}'$ to receive the shares encoding the same secret. The main idea is similar to the one in $\texttt{Recover}^{\texttt{M}}$, but as parties might be different in two phases, it needs to be considered very carefully. To send a right sharing of s from $\mathcal{P}$ to $\mathcal{P}'$ without revealing the secret value s to the parties, both parties in two phases generate a sharing of random value r without reconstructing the value r using $\texttt{ShareRandom}^{\texttt{M}}$. Then, parties holding the share of s locally compute the share of r to the share of s and reconstruct $s + r$ using them. Now, all parties in $\mathcal{P}$ know the value $s + r$, but not s or r, so they invoke $\texttt{RobustReshare}^{\texttt{M}}$ to send a right sharing of $s + r$ to the parties in $\mathcal{P}'$. As parties in $\mathcal{P}'$ also hold the share of r, each party can locally compute the share of s.

Protocol $\texttt{Redistribute}^{\texttt{M}}(w, [s]^w) \longrightarrow [s]^{w+1}$

Input: a phase w and the sharing of s in phase w, $[s]^w = \{\mathbf{s}_1^w, ..., \mathbf{s}_n^w\}$
Output: new sharing of s for phase $w + 1$, $[s]^{w+1} = \{\mathbf{s}_1^{w+1}, ..., \mathbf{s}_m^{w+1}\}$

1. Parties in $\mathcal{P}$ and $\mathcal{P}'$ invoke $\texttt{ShareRandom}^{\texttt{M}}(W)$, where $W = \{w, w+1\}$, to generate two sharings of a random value r, unknown to every party. That is, parties in $\mathcal{P}$ separately receive a sharing $[r]^w := \{\mathbf{r}_1^w, ..., \mathbf{r}_n^w\}$, while parties in $\mathcal{P}'$ receive a sharing $[r]^{w+1} := \{\mathbf{r}_1^{w+1}, ..., \mathbf{r}_m^{w+1}\}$, and no one knows the value of r.

2. Each party P_i in $\mathcal{P}$ locally computes $\mathbf{x}_i := \mathbf{r}_i^w + \mathbf{s}_i^w$, where $\mathbf{s}_i^w$ is the share of s.
3. Parties in $\mathcal{P}$ invoke $\texttt{LC-Reconstruct}^{\texttt{M}}(w, [x])$ with $[x] := \{\mathbf{x}_1, ..., \mathbf{x}_n\}$ and the result is denoted by x. Note that $x = s + r$, where r is random and unknown to everyone.
4. Parties invoke $\texttt{RobustReshare}^{\texttt{M}}(x, w, \mathcal{P}, w+1, \mathcal{P}')$ so that parties in $\mathcal{P}'$ receive a sharing of x, say $[x] := \{\mathbf{z}_1, ..., \mathbf{z}_m\}$, for $\mathbf{z}_i := M_i'\mathbf{X}$, where the vector $\mathbf{X} = (x, \$, ..., \$) \in \mathbb{F}^{e'}$ with random \$'s.
5. Each party P_i' in $\mathcal{P}'$ locally computes $\mathbf{s}_i^{w+1} := \mathbf{z}_i - \mathbf{r}_i^{w+1}$, for $i = 1, ..., m$.
6. Parties in $\mathcal{P}'$ collectively output $\{\mathbf{s}_1^{w+1}, ..., \mathbf{s}_m^{w+1}\}$ as a sharing of s in new phase.

Theorem 5.6. *(Correctness and Secrecy of* $\texttt{Redistribute}^{\texttt{M}}$*) When the protocol terminates, all parties in the new participating group have the shares of the same secret as the old shares, and the protocol does not reveal any information about the secret. It communicates* $O(n^2\kappa \log d + nm^2\kappa \log d' + ((n^2+mn)d + (m^2+mn)d' + (n^3+m^3)\kappa + (m+n)mn\kappa + nm^3) \log |\mathbb{F}|)$ *bits and broadcasts* $O((n^3 + mn^2)\log d + nm^2 \log d' + (n^3 + (n+m)(mn+d) + nd')\log |\mathbb{F}|)$ *bits, where* $|\mathcal{P}| = n$, $|\mathcal{P}'| = m$, *size(M) = d, and size(M′) = d′.*

Proof. **Correctness:** Since there exists λ such that $(M_B')^t\lambda = \mathbf{a}$ for $\forall B \in \Gamma$, $\langle \lambda, \mathbf{s}\rangle = \langle \lambda, M_B'\mathbf{S}\rangle = \langle \lambda, M_B'(\mathbf{X} - \mathbf{R})\rangle = \langle \lambda, M_B'\mathbf{X}\rangle - \langle \lambda, M_B'\mathbf{R}\rangle = \langle (M_B')^t\lambda, \mathbf{X}\rangle - \langle (M_B')^t\lambda, \mathbf{R}\rangle = x - r = (s+r) - r = s$, where $\mathbf{s}$ is the recomposition vector with shares of parties in $\forall B \in \Gamma$, $\mathbf{S}$ is the recomposition vector with shares of all parties in $\mathcal{P}'$, $\mathbf{X}$ is the vector defined in Step 4, and $\mathbf{R}$ is the vector of length e' having r of Step 1 for the first component and $(e'-1)$ random values for the others. Thus, new sharing reconstructs the same secret s as the input sharing. **Secrecy:** Communicating values are either random value or the share of random value, and the shares of secret s are handled only by local computations. Also, since no one knows the value of r throughout the protocol, the reconstruction of x does not reveal any information about the secret value s. **Communication:** Apply $N = n + m$ and $|W| = 2$ for $\texttt{ShareRandom}^{\texttt{M}}$, $l = 1$ for $\texttt{LC-Reconstruct}^{\texttt{M}}$, and $|\mathcal{P}_S| = |\mathcal{P}| = n$ and $|\mathcal{P}_R| = |\mathcal{P}'| = m$ for $\texttt{RobustReshare}^{\texttt{M}}$, as the others are the local computations. Assuming $m = n$ and $d' = d$, the total number of communication bits is $O(n^3\kappa \log d + (n^2 d + n^4 + n^3\kappa)\log |\mathbb{F}|)$ and the total broadcast bits are $O(n^3 \log d + (n^3 + nd)\log |\mathbb{F}|)$. □

Table 5.3 shows the total analysis of MSP-based PMPC protocols based on the protocols in [204]. In the table, κ denotes the security parameter. Only IC scheme and `LC-Reconstruct` are based on multiple secret values, and the others are based on one secret value. In IC, l is the number of secret values and in `LC-Reconstruct`, $L := max_j(l_j)$, where l_j is the number of secrets from P_j. In `ShareRandom`, $N = \sum_{w \in W} |\mathcal{P}^w|$ and W is a set of phases which parties participate in the protocol. In `Redistribute`, it is assumed that $|\mathcal{P}| = |\mathcal{P}'| = n$ and $size(MSP) = d = d'$. Note that all protocols still have linear complexities in size of MSP, d, even after adding out new protocol, for both static and dynamic groups. Even after adding our new protocols, for both static groups and dynamic groups, the total communication/broadcast complexities remain linear in the size of MSP, d, the number of rows of the corresponding matrix M.

	MSP-based PMPC based on [204]		
	Protocol	**Communication Cost(bits)**	**Broadcast Cost(bits)**
Information Checking	`Authenticate`*	$O(\kappa(\log d + \log |\mathbb{F}|))$	$O(\log d + \log |\mathbb{F}|)$
	`Verify`*	$O(\kappa \log d + (l + \kappa) \log |\mathbb{F}|)$	1
Secret Sharing / MPC	`BasicShare`	$O(d \log |\mathbb{F}|)$	-
	`Share`	$O((nd + n^2\kappa) \log |\mathbb{F}| + n^2\kappa \log d)$	$O(n^2 \log d + (n^2 + d) \log |\mathbb{F}|)$
	`Reconstruct`	$O(n^2\kappa \log d + (n^3 + n^2\kappa) \log |\mathbb{F}|)$	$O(d \log |\mathbb{F}|)$
	`LC-Reconstruct`*	$O(n^2\kappa \log d + (n^3 + n^2\kappa) \log |\mathbb{F}|)$	$O(n(\log_2 L + 1)d \log |\mathbb{F}|)$
	`Add`	-	-
	`Gen-Rand`	$O((n^2 d + n^3\kappa) \log |\mathbb{F}| + n^3\kappa \log d)$	$O(n^3 \log d + (n^3 + nd) \log |\mathbb{F}|)$
	`Gen-Mult-Triples`	$O((n^4 + n^3\kappa + n^2 d) \log |\mathbb{F}| + n^3\kappa \log d)$	$O(n^3 \log d + (n^3 + n^2 d) \log |\mathbb{F}|)$
Our Additional Protocols	`Refresh`	$O((n^2 d + n^3\kappa) \log |\mathbb{F}| + n^3\kappa \log d)$	$O(n^3 \log d + (n^3 + nd) \log |\mathbb{F}|)$
	`ShareRandom`	$O(N|W|((nd + n^2\kappa) \log |\mathbb{F}| + n^2\kappa \log d))$	$O(N|W|(n^2 \log d + (n^2 + d) \log |\mathbb{F}|))$
	`RobustReshare`	$O(|\mathcal{P}_S|(|\mathcal{P}_R|^2\kappa \log d_R + (|\mathcal{P}_R|^3 + |\mathcal{P}_R|^2\kappa + |\mathcal{P}_R| d_R) \log |\mathbb{F}|))$	$O(|\mathcal{P}_S|(|\mathcal{P}_R|^2 \log d_R + (|\mathcal{P}_R|^2 + d_R) \log |\mathbb{F}|))$
	`Recover`	$O(n^3\kappa \log d + (n^4 + n^3\kappa + n^2 d) \log |\mathbb{F}|)$	$O(n^3 \log d + (n^3 + nd) \log |\mathbb{F}|)$
	`Redistribute`	$O(n^3\kappa \log d + (n^2 d + n^4 + n^3\kappa) \log |\mathbb{F}|)$	$O(n^3 \log d + (n^3 + nd) \log |\mathbb{F}|)$
Total	`MSP-based PMPC`	$O(n^3\kappa \log d + (n^2 d + n^4 + n^3\kappa) \log |\mathbb{F}|)$	$O(n^3 \log d + (n^3 + n^2 d) \log |\mathbb{F}|)$

Table 5.3: Total analysis of protocols in MSP-based PMPC scheme based on [204]

5.5.5 Conversions between Additive and MSP-based MPC

Now, we present how to convert the additive PMPC scheme into the MSP-based PMPC and in the opposite direction. Recall that the complexity of an additive PMPC scheme depends on the size of the sharing specification $|\mathbb{S}|$ (we use the basic secrecy structure $|\widetilde{\Sigma}|$), while the one of an MSP-based PMPC scheme depends on the size of the MSP, d. Since d can be varied from n to $|\widetilde{\Sigma}|^{2.7}$ depending on the adversary structures [204], there are some cases worth converting the schemes even though the conversion itself needs some resource. One PMPC scheme with better complexities can be chosen only when participating groups or GAS are changed. That is, when dynamic groups and structures of two consecutive phases are given, participating parties can continue the current PMPC scheme by executing `Redistribute` protocol or they can convert the scheme from one to the other by calling the protocols, called `ConvertAdditiveIntoMSP` or `ConvertMSPIntoAdditive`.

Let dynamic groups and structures in consecutive phases are given as $\mathcal{S}^w := (\mathcal{P}, \Gamma, \Sigma, \Delta, \mathbb{S})$ and $\mathcal{S}^{w+1} := (\mathcal{P}', \Gamma', \Sigma', \Delta', \mathbb{S}')$ and let the additive PMPC scheme has been using in phase w with sharing specification $\mathbb{S} = \{S_1, S_2, ..., S_k\}$. The protocol `ConvertAdditiveIntoMSP` converts current additive sharing of s into a MSP-based sharing of s. By definition, if no qualified subset of parties in the access structure Γ remains in $\mathcal{P}$, then the secret value s cannot be reconstructed even though the protocol is executed. That is, at least one honest party exists in each $S_i \in \mathbb{S}$. To deal with active adversaries, all parties in each S_i need to share their holding share s_i to the parties in $\mathcal{P}'$ using the $\texttt{Share}^{\texttt{M}}$ protocol. Then parties in $\mathcal{P}'$ hide their shares with the shares of a random number and open (reconstruct) the hidden values to decide one sharing of s_i from the honest party in S_i. By linearity of shares, each party in $\mathcal{P}'$ can locally compute the MSP-based share of s by component-wise adding all its receiving shares. The formal protocol is as follows.

Protocol $\texttt{ConvertAdditiveIntoMSP}([s]^w, w, \mathcal{S}^w, w+1, \mathcal{S}^{w+1}) \longrightarrow [s]^{w+1}$

Input: $\mathcal{S}^w := (\mathcal{P}, \Gamma, \Sigma, \Delta, \mathbb{S})$ in phase w and a sharing $\{s_1, ..., s_{|\mathbb{S}|}\}$ of s such that $\sum_{i=1}^{|\mathbb{S}|} s_i = s$
Output: a sharing of s for $\mathcal{S}^{w+1} := (\mathcal{P}', \Gamma', \Sigma', \Delta', \widehat{M})$ in phase $w+1$, where $M \in \mathcal{M}(d \times e)$ is

```
        corresponding matrix of the MSP $\widehat{M}$
1.  For each $i \in \{1, ..., |\mathbb{S}|\}$ (in parallel):
2.      Every party in $S_i$ invokes $\texttt{Share}^{\texttt{M}}(s_i, w+1, \mathcal{P}')$. Denote $|S_i|$ sharings of $s_i$ by $[s_i]^{(1)}, ..., [s_i]^{(|S_i|)}$.
3.      Parties in $\mathcal{P}'$ invoke $\texttt{ShareRandom}^{\texttt{M}}(w+1)$ to generate a sharing of a random number, say $r^{(i)}$.
4.      Parties in $\mathcal{P}'$ locally compute $[x_i^{(j)}] := [s_i]^{(j)} + [r^{(i)}]$, for $j = 1, ..., |S_i|$.
5.      Parties in $\mathcal{P}'$ execute $\texttt{LC-Reconstruct}^{\texttt{M}}(w+1, [x_i^{(j)}])$ (in parallel) $|S_i|$ times for each sharing
        and choose a set $H \in Honest := \{\mathcal{P} \setminus A | A \in \overline{\Delta}\}$ that $x_i^{(j)} = v$ for all $P_{k_j} \in (S_i \cap H)$. If there
        exists multiple such sets, they choose the minimal set including $P_{id}$ with lower $id$.
6.      The sharing $[s_i^{(min)}]$ of $s_i$ from $P_{min} \in H$ is chosen as a sharing of $s_i$, say $[s_i]$.
7.  At this point, parties in $\mathcal{P}'$ hold $|\mathbb{S}|$ sharings for each $s_i$ and each party $P_j \in \mathcal{P}'$ holds $|\mathbb{S}|$ vectors
    of length $d_j$, for each sharing. Each party locally computes component-wise addition with these
    vectors and sets it as its share of $s$. i.e., $P_j$ computes $\mathbf{s}_j := \sum_{i=1}^{|\mathbb{S}|}[s_i]_j \in \mathbb{F}^{d_j}$, where each share
    is the vector of length $d_j$.
8.  Parties in $\mathcal{P}'$ collectively output a sharing of $s$, $[s]^{w+1} := \{\mathbf{s}_1, ..., \mathbf{s}_m\}$, where $m = |\mathcal{P}'|$.
```

Theorem 5.7. *(Correctness and Secrecy of* `ConvertAdditiveIntoMSP`*) When the protocol* `ConvertAdditiveIntoMSP` *terminates, all parties in the new participating group have shares of the same secret encoded by the old shares, and the protocol does not reveal any information about the secret.* `ConvertAdditiveIntoMSP` *communicates $O(k((m^2+mn)d+(m^3+m^2n)\kappa+nm^3)\log|\mathbb{F}|+k(m^3+m^2n)\kappa\log d)$ bits and broadcasts $O(k(mnd+m^3+m^2n)\log|\mathbb{F}|+k(m^3+m^2n)\log d)$ bits, where $|\mathcal{P}| = n$, $|\mathcal{P}'| = m$, $|\mathbb{S}| = k$, and size(M) = d.*

Proof. **Correctness:** Since the adversary chooses one set in Δ to corrupt, there exists at least one subset of parties in $Honest$ that includes only honest parties. Thus, parties in $\mathcal{P}'$ can figure out the right sharing of s_i from $|S_i|$ reconstruction values $\{(x_i^{(j)}\}_{j=1}^{|S_i|}$ for each i. In Step 7, each party $P_j \in \mathcal{P}'$ locally computes the summation of $|\mathbb{S}|$ vectors, $\mathbf{s}_j := \sum_{i=1}^{|\mathbb{S}|}[s_i]_j \in \mathbb{F}^{d_j}$, where $[s_i]_j = M_j\mathbf{b}^{(i)}$ for $\mathbf{b}^{(i)} := (s_i, \$, ..., \$)^t \in \mathbb{F}^e$. By definition, $\exists\lambda \in \mathbb{F}^{d_B}$ such that $M_B^t\lambda = \mathbf{a} \in \mathbb{F}^e$, for $\forall B \in \Gamma'$. When parties in $\forall B \in \Gamma'$ reconstruct with their shares, the recomposition vector with B shares is $\mathbf{S}_B = M_B\mathbf{b}$, where $\mathbf{b} = \sum_{i=1}^{|\mathbb{S}|}\mathbf{b}^{(i)}$. Thus, $\langle\lambda, \mathbf{S}_B\rangle = \langle\lambda, M_B\mathbf{b}\rangle = \langle M_B^t\lambda, \mathbf{b}\rangle =$(the first component of $\mathbf{b}$) $= \sum_{i=1}^{|\mathbb{S}|} s_i = s$. **Secrecy:** Since each $S_i \in \mathbb{S}$ can include one or more parties, and some of them might be corrupted, parties who receive the sharing of s_i need to choose the sharing of the honest party in S_i. However, if parties in $\mathcal{P}'$ reconstruct s_i, all parties can compute s by adding all s_i's at the end. Therefore, they generate a sharing of a random element $r^{(i)}$ for each s_i, and the random number will never be reconstructed. In Step 4 of the loop, although each party sees the value of $x_i^{(j)}$, it does not reveal anything about s_i because it is hidden by a random number that no one knows. **Communications:** It costs $|\mathbb{S}| * \{n * Cost(\texttt{Share}^{\texttt{M}}) + Cost(\texttt{ShareRandom}^{\texttt{M}}) + n * Cost(\texttt{LC-Reconstruct}^{\texttt{M}})\}$, as $max_i|S_i| = |\mathcal{P}| = n$ when S_i include all parties in $\mathcal{P}$. □

On the other hand, when participating parties currently use the MSP-based PMPC scheme and want to convert it to the additive PMPC in the next phase, they can execute the protocol `ConvertMSPIntoAdditive`. It converts an MSP-based sharing of s in phase w into an additive sharing of s in phase $w+1$. Note that each party P_i has different shares of s in MSP-based PMPC, and P_i's share of s is the vector of length d_i. For these reasons, each party needs to invoke multiple $\texttt{Share}^{\texttt{A}}$ protocols to share each component of the vector according to the sharing specification $\mathbb{S}'$ in phase $w+1$. Each party in each $S_j \in \mathbb{S}'$ collects all the shares received from the same party P_i and forms a vector of length d_i. Then, all the parties in S_j hold the same n vectors of different lengths. When recomposing these n vectors according to the indexing function ρ of phase w, each party can compute its share of s by inner product with the vector λ such that $M^t\lambda = \mathbf{a}$.

Protocol `ConvertMSPIntoAdditive`$([s]^w, w, \mathcal{S}^w, w+1, \mathcal{S}^{w+1}) \longrightarrow [s]^{w+1}$

Input: $\mathcal{S}^w := (\mathcal{P}, \Gamma, \Sigma, \Delta, \widehat{M})$ and the sharing $\{\mathbf{s}_i\}_{i=1}^n$ of s such that $\mathbf{s}_i = M_i\mathbf{b}$ for each i, where $M \in \mathcal{M}(d \times e)$ of the MSP $\widehat{M}$ computes f and accepts Γ and $\mathbf{b} = (s, r_2, ..., r_e)$ for $r_i \xleftarrow{\$} \mathbb{F}$

Output: a sharing of s for $\mathcal{S}^{w+1} := (\mathcal{P}', \Gamma', \Sigma', \Delta', \mathbb{S}')$, where $\mathbb{S}' := \{S_1, S_2, ..., S_k\}$ is the sharing specification in phase $w+1$

1. Each party $P_i \in \mathcal{P}$ invokes $\texttt{Share}^{\texttt{A}}(w+1, s_j^{(i)}, \mathcal{P}')$, for each $s_j^{(i)}$ of $\mathbf{s}_i := (s_1^{(i)}, ..., s_{d_i}^{(i)})$.
2. Every party in $S_j \in \mathbb{S}'$ forms a vector of shares received in Step 1 from the same party P_i, as $\mathbf{s}_{ij} := ((s_1^{(i)})_j, ..., (s_{d_i}^{(i)})_j)$. i.e., Parties in S_j hold n different vectors $\{\mathbf{s}_{1j}, ..., \mathbf{s}_{nj}\}$ from every

party in $\mathcal{P}$ and each vector $\mathbf{s}_{ij}$ has length d_i.

3. Every parties in S_j forms the recomposition vector $\mathbf{Q}_j$ with n vectors $\{\mathbf{s}_{1j}, ..., \mathbf{s}_{nj}\}$ with respect to the indexing function ρ of $\widehat{M}$. Note that the length of $\mathbf{Q}_j$ is d for all $j = 1, ..., k$.
4. Each party in S_j sets $s_j := \langle \lambda, \mathbf{Q}_j \rangle$, where λ is the vector such that $M^t \lambda = \mathbf{a}$, for each $j = 1, ..., |\mathbb{S}|$.
5. Players in $\mathcal{P}'$ collectively output $[s]^{w+1} := \{s_1, ..., s_k\}$.

Theorem 5.8. *(Correctness and Secrecy of* `ConvertMSPIntoAdditive`*) When the protocol terminates, all parties in the new participating group have the shares of the same secret as the old shares, and the protocol does not reveal any information about the secret. It communicates $O(dkn^3 \log |\mathbb{F}|)$ bits and broadcasts $O(dkn^3 \log |\mathbb{F}|)$ bits, where $|\mathcal{P}| = n$, $|\mathcal{P}'| = m$, $|\mathbb{S}'| = k$, and size$(M) = d$.*

Proof. **Correctness:** After Step 1, each party in S_j of phase $w+1$ gets n vectors, $\{\mathbf{s}_{1j}, \mathbf{s}_{2j}, ..., \mathbf{s}_{nj}\}$, where $\mathbf{s}_{ij} := ((s_1^{(i)})_j, (s_2^{(i)})_j, ..., (s_{d_i}^{(i)})_j) \in \mathbb{F}^{d_i}$ is the vector of j-th additive shares of the share that $P_i \in \mathcal{P}$ has. Since each party in S_j received each vector $\mathbf{s}_{ij}$ from all n parties in $\mathcal{P}$, it can rearrange n vectors with respect to the indexing function ρ and form a vector of length d. Since parties in S_j have all j-th additive share of shares of phase w, $\mathbf{Q}_j$ is the vector of j-th shares of each component of $\mathbf{s} \in \mathbb{F}^d$, where $\mathbf{s} := M\mathbf{b}$, $\mathbf{b} := (s, r_2, ..., r_e) \in \mathbb{F}^e$ as in Section 5.5.4. i.e., $\sum_{j=1}^{k} \mathbf{Q}_j = \mathbf{s}$. By the properties of the inner product (over $\mathbb{R} \supset \mathbb{F} = GF(p)$) that $\langle u, v \rangle = \langle v, u \rangle$ and $\langle au + bv, w \rangle = a\langle u, w \rangle + b\langle v, w \rangle$, $\langle \lambda, \mathbf{Q}_1 + ... + \mathbf{Q}_k \rangle = \langle \lambda, \mathbf{Q}_1 \rangle + ... + \langle \lambda, \mathbf{Q}_k \rangle$. Since $\langle \lambda, \mathbf{s} \rangle = \langle \lambda, M\mathbf{b} \rangle = \langle M^t \lambda, \mathbf{b} \rangle = \langle \mathbf{a}, \mathbf{b} \rangle = s$, and $\langle \lambda, \mathbf{s} \rangle = \langle \lambda, \mathbf{Q}_1 + ... + \mathbf{Q}_k \rangle = \langle \lambda, \mathbf{Q}_1 \rangle + ... + \langle \lambda, \mathbf{Q}_k \rangle$, each $\langle \lambda, \mathbf{Q}_j \rangle$ can be set as the j-th additive share of s. **Secrecy:** It relies on the secrecy of the protocol `Share`$^{\mathtt{A}}$. As what each party receives after the protocol is one additive share of s, it does not reveal any information about the secret until `Reconstruct`$^{\mathtt{A}}$ is executed. **Communications:** As all other steps are local computations, complexities only rely on the $d * Cost($`Share`$^{\mathtt{A}})$. □

From the Theorem 5.7 and Theorem 5.8, we can derive the following corollary.

Corollary. *A proactive MPC scheme based on additive secret sharing and a proactive MPC scheme based on MSP-based secret sharing are convertible. That is, one can transform an additive sharing of a secret to a MSP-based sharing of the same secret and also transform a MSP-based sharing of a secret to an additive sharing of the same secret, without revealing any information about the secret among participating parties.*

5.6 Related Work in Proactive Secret Sharing and Proactive MPC

Proactive Secret Sharing (PSS): SS is typically utilized as a building block for MPC protocols. Table 5.4 summarizes existing PSS schemes, where n is the number of parties, t is the threshold for each refresh phase, $|\mathbb{S}|$ denotes the size of sharing specification, and d is the size of a monotone span program, which is the number of rows of the matrix M. Note that [125] also handles mixed adversaries which are characterized by two thresholds, one for passive corruptions and one for active corruptions. All previous work on PSS considers only the threshold adversary structures and are typically insecure when the majority of the parties are compromised. PSS schemes [241, 168, 300, 311] [269, 48, 49] typically store the secret as the free term in a polynomial of degree $t < n/2$; thus once an adversary compromises $t+1$ parties (even if only passively), it can reconstruct the polynomial and recover the secret. PSS schemes with optimal-communication [48, 49] also use a similar technique, but instead of storing the secret in the free term, they store a batch of b secrets at different points in the polynomial; similar to the single secret case, even when secrets are stored as multiple points on a polynomial, once the adversary compromises $t+1$ parties, it can reconstruct the polynomial and recover the stored secrets. Different techniques are required to construct PSS secure against GAS. Recently [125] developed the first PSS scheme for a dishonest majority but also only for a threshold adversary structure, the scheme cannot be generalized for other structures. Also, the work in [125] only describes a PSS scheme and does not specify how to perform PMPC for the same thresholds. While it may be possible to extend that PSS scheme to PMPC, it will remain limited to the threshold adversary structure.

Scheme	**Threshold Passive (Active)**	**Security**	**Network Type**	**Dynamic Groups**	**Adversary Structure**	**Comm. Complexity**
[300]	$t < n/2$ $(n/2)$	Cryptographic	Synchronous	Static	Threshold	exp(n)
[311]	$t < n/3$ $(n/3)$	Cryptographic	Asynchronous	Static	Threshold	exp(n)
[71]	$t < n/3$ $(n/3)$	Cryptographic.	Asynchronous	Static	Threshold	$O(n^4)$
[269]	$t < n/3$ $(n/3)$	Cryptographic	Asynchronous	Static	Threshold	$O(n^4)$
[168]	$t < n/2$ $(n/2)$	Cryptographic	Synchronous	Static	Threshold	$O(n^2)$
[48]	$t < n/3 - \epsilon$ $(n/3 - \epsilon)$	Perfect	Synchronous	Static	Threshold	$O(1)^*$
[48]	$t < n/2 - \epsilon$ $(n/2 - \epsilon)$	Unconditional	Synchronous	Static	Threshold	$O(1)^*$
[49]	$t < n/2 - \epsilon$ $(n/2 - \epsilon)$	Unconditional	Synchronous	Dynamic	Threshold	$O(1)^*$
[125]	$t < n - 1$ $(n/2 - 1)$	Cryptographic	Synchronous	Static	Threshold	$O(n^4)$
[222]	$(t < n/2)$	Cryptographic	Synchronous	Dynamic	Threshold	$O(n^3)$
This work						
Additive	N/A	Unconditional	Synchronous	Static	General	$O(\lvert\mathbb{S}\rvert * poly(n))$
Additive				Dynamic		$O(\lvert\mathbb{S}\rvert^2 * poly(n))$
MSP-based				Static		$O(d * poly(n))$
MSP-based				Dynamic		$O(d * poly(n))$

Table 5.4: Comparing existing proactive secret sharing (PSS) schemes; (*) Communication complexities in [48, 49] are amortized

Proactive Secure Multiparty Computation (PMPC): To the best of our knowledge, there are currently only a few PMPC protocols, e.g., [241] requiring $O(Cn^3)$ communication, where C is the size of the circuit to be computed via MPC, and [48] requiring $O(Clog^2(C)polylog(n) + Dpoly(n)log^2(C))$. Existing PMPC protocols are only specified for threshold adversary structures and cannot be easily[4] augmented to handle GAS; this is because these protocols all rely on secret sharing via polynomials. In addition, all current PMPCs can only tolerate dishonest minorities, except one protocol [134], but even that one is limited to the threshold adversary structure. The reason is that the underlying SS stores secrets as points on polynomials, so once the adversary compromises enough parties (even if only passively), it can reconstruct the polynomial and recover the secret. The only structure that can be described is one in terms of a fraction of the degree of the polynomial (typically also a fraction of number of parties), and once the adversary compromises enough parties (even if only passively), it can reconstruct the polynomial and recover the secret.

[4]Or at least it is not obvious to us how to easily augment them to accommodate GAS.

5.7 Summary

This work constructed the first communication-efficient structure-adaptive proactive MPC schemes for dynamic GAS settings. Considering two MPC schemes based on additive secret sharing and monotone span programs, we first made these MPC schemes adapt to the proactive security settings by adding `Refresh` and `Recover` protocols. Then, we extended our PMPC schemes to dynamic group settings by adding `Redistribute` protocols. Finally, we presented the share conversions to efficiently and securely convert back and forth between these two PMPC schemes.

Chapter 6

Balancing Security and Privacy in Genomic Range Queries

This chapter explores application of cryptographic protocols in a specific use-case: genomic testing, and shows the challenges of balancing security and privacy when applying cryptographic techniques to real-world scenarios.

6.1 Introduction

Dramatic recent technical advances in DNA sequencing technology [223, 140, 166] and reduced sequencing costs paved the way for ubiquitous and affordable genomic testing. As a result, genomic tests, such as paternity/parentage and pre-symptomatic disease diagnosis, that were used in the past mainly by doctors and legal authorities are becoming available to the general public.

In a typical genomic testing scenario, a testing facility (“tester"), such as 23andMe [1] or CRI Genetics [2], requests genomic data from an individual (“Alice") regarding specific locations and/or ranges on the DNA of that individual. Alice naturally wants to reveal minimal genomic data, since DNA – besides one's own highly personal and sensitive material – includes significant information about her past, current, and future relatives. The tester also needs to keep specifics (such as queried locations) secret, due to the often-proprietary nature of these genomic tests. Consequently, genomic privacy has justifiably attracted lots of attention from the research community and numerous privacy techniques have been proposed [45, 235, 47, 103, 102, 40, 41, 289, 188].

On the other hand, even though at least as important, genomic security received considerably less attention than genomic privacy. In particular, the authenticity and integrity of genomic data are often ignored or over-simplified, though they are crucial to accurate outcomes of genomic tests. An erroneous (whether or not maliciously caused) genomic test result can have grave health consequences when used for medical diagnoses or treatment. It could also involve social risks when determining familial relationships or other non-health-related traits.

At first glance, the genomic security problem seems simple and easily solvable with traditional cryptographic primitives, such as digital signatures. However, the main challenge stems from conflicting requirements in the triad of security, privacy, and efficiency. For example, a single signature on Alice's whole genome provides security – specifically, authenticity and integrity. However, it requires Alice to send the entire genome to the tester (for signature verification), which results in Alice having zero genomic privacy and incurs significant communication costs. Another intuitive approach is to individually sign each smallest genomic unit (called a "base") and provide the tester with only those signed bases needed for a given test. This would offer much better privacy for Alice and incur much lower communication overhead. However, it is expensive to compute (at sequencing time) and requires the tester to verify potentially many signatures.

As an alternative to the whole genome representation, other more compact DNA representations can be used. One such example called Single-Nucleotide Polymorphisms (SNPs) –

one-base genomic mutations, account for only about 0.1% of the entire genome. Therefore, they are significantly more efficient in representing a genome. However, this increase in efficiency introduces additional security problems. If we sign each SNP individually, since SNP locations are not consecutive and their positions are unpredictable (sprinkled throughout the entire genome), Alice could cheat by omitting signed SNPs from the requested range.[1] There are similar trade-offs for other candidate representations.

Inspired by these challenges, this work focused on reconciling genomic security, privacy, and efficiency. Genomic security requires authenticity, which comprises origin authentication, integrity, and completeness. It aims to counter potentially malicious owners and/or outsiders tampering with genomic data. Whereas, privacy (against malicious testers) demands flexibility and sufficiently fine granularity controlled by the genome owner so as to reveal minimal information. At the same time, efficiency motivates minimizing genomic data processing, which complicates both privacy and security. After carefully examining security, privacy, and efficiency needs, we propose techniques based on the combination of established cryptographic tools that achieve a good balance for genomic testing. Anticipated contributions are as follows:

- We construct a secure and private range query technique as a building block for various protocols and genomic representations. Range queries allow us to perform efficient genomic tests on various regions[2] that control similar functions.
- To demonstrate the applicability of the proposed technique, we use it to improve both the security and performance of prior protocols for private genomic substring matching [103].

[1] Of course, she cannot introduce fake SNPs.

[2] <u>*Note:*</u> One example of such a region is the *Major Histocompatibility Complex (MHC)*, a large locus on vertebrate DNA, which consists of a set of genes coding for proteins responsible for detecting foreign molecules at the cellular level. MHC ranges from `6p22.1` to `6p21.3` and consists of about 4 megabases; see Cytogenetic location [4]. Various SNPs in this region (e.g., `rs9264942, rs4418214, rs2395029, and rs3131018`) have been shown to protect against Human Immunodeficiency Virus (HIV) [282].

- We prototype proposed protocols and evaluate their performance. In the course of the evaluation, we investigate various optimizations and analyze the performance of two additively homomorphic encryption schemes (ElGamal [136] and Paillier [242]) and two range-proof schemes (signature-based [72] and BulletProofs [69]).
- Finally, we generalize the problem setting to sparse integer sets and discuss applications of the proposed schemes beyond genomics.

Organization: Section 6.2 overviews background, followed by Section 6.3 providing our system and security models, as well as the range completeness problem. Our main contribution is introduced in Section 6.4, followed by its application to the private substring matching problem (SPH-PSM [103]) in Section 6.5. Sections 6.6 and 6.7 discuss the implementation and evaluation of the proposed construction and its application to SPH-PSM. Next, Section 6.8 generalizes the problem to the sparse integer sets and comments on its security. Then, Section 6.9 overviews related work, followed by some limitations of this work and future work in Section 6.10. Table 6.1 summarizes all notation and acronyms used in this work.

6.2 Preliminaries

6.2.1 Genomics

The human genome is composed of around 3.2 billion [3] base pairs packed into 23 chromosomes of different size [220]. Each base can be represented by four letters of the genetic code – [A]denine, [C]ytosine, [G]uanine, and [T]hymine, where [A] always bonds with [T], and [G] always bonds with [C]. According to The Human Genome Project (HGP), only around 0.1%

[3] A diploid human genome has around 6 billion base pairs, whereas the HGP reference genome contains only around 3.2 billion base pairs. This is because most human cells contain 23 chromosomes in pairs (i.e., 46 chromosomes in total), which are almost identical. Only one of each pair, 24 chromosomes in total – 22 non-sex and 2 sex (X and Y) – can represent the whole human genome information.[17]

Table 6.1: Notation & Acronyms used in Chapter 6

Notation	**Description**
SL	A sequencing lab that digitalizes analog genomic sample
T	A tester performing genomic tests on digitized genomes
Alice	An individual requesting her digitized genome from SL and later interacting with T for a genomic testing
$Q = [a, b]$	T's range query, where a and b are integer boundaries such that $0 < a \leq \ b < N(\approx 3.2 * 10^9)$
Bio_{Alice}	Alice's (physical) DNA sample
$G_{Alice} = \{g_1, g_2, ..., g_N\}$	Alice's whole genome represented as an ordered set of tuples of the form: $g_i = (i, l_i)$, $i = 1, ..., N$, where i is an integer position and l_i is an integer in [0,15] representing a combination of two base letters: $\{A, G, C, T\}$. (e.g., 0 for 'AA', 1 for 'AC', etc.)
$\mathcal{M} = \{m_1, m_2, ..., m_n\}$	SNP representation of Alice's genome, where n is the number of genetic variations. Denoted by a tuple $m_i = (pos_i, l_i), i = 1, ..., n$, where $\mathcal{M} \subset G_{Alice}$. Note that $pos_i < pos_{i+1}$ for $\forall i$
$\mathcal{M}_Q \subseteq \mathcal{M}$	The subset of $\mathcal{M}$ including all Alice's SNPs located in Q. i.e., $\mathcal{M}_Q = \{m_i \in \widehat{\mathcal{M}} \mid a \leq pos_i \leq b\}$, say $\{m_{k+1}, ..., m_{k+j}\}$ for some k, j
$(Sign(.), Verify(.))$	A secure digital signature scheme
$Com(x)$	A secure commitment scheme allowing zero-knowledge range proofs over x. Equivalent to $Com(x; s)$, where s is a random salt
$\mathsf{NIZK}(z : R(z))$	A non-interactive zero-knowledge proof of knowledge of z such that $R(z)$ holds. Denote proof and verification protocol as $\mathsf{NIZK\_Prove}(.)$ and $\mathsf{NIZK\_Verify}(.)$, respectively
$(Enc(.), Dec(.))$	An additively homomorphic encryption scheme
$H(.)$	A cryptographic hash function
(pk_X, sk_X)	A pair of public and private key of an entity X

of base pairs differ between individuals [15]. Although it is not yet possible to determine or predict exactly where these differences occur, many types of genetic variations can be used to identify an individual and assess one's susceptibility to diseases and sensitivity to drugs. Single-nucleotide polymorphisms (SNPs) are the most common type of genetic variation (a.k.a. mutations) among people, which represents a difference in a single base, e.g., an [A] changes to a [C] or a [G]. A variation is classified as an SNP if over 1% of a population does not carry the same nucleotide at a specific position in the DNA sequence [13].

Normally, SNP data sequenced by, e.g., 23andMe [1] contains two base letters, one per each chromosome. For example, let the genotype of an SNP among Alice's DNA sequenced result be 'AG' at position 169. This means that 'A' is on one strand of one chromosome, and 'G' is on one strand of the other chromosome – naturally, the opposite strands have paired 'T' and 'C', respectively at position 169 – and another person can have a different genotype at position 169, e.g., 'CC'. To represent the genotypes efficiently, we use an 8-bit[4] integer $\in [0, 15]$, instead of representing all 2-character combinations of {A,C,G,T}. Thus, we denote a base as (pos, l), where pos is a non-negative integer $< 3.2 * 10^9$ representing the position of the base and l is a non-negative integer $\in [0, 15]$ representing two base letters from each chromosome at that position.

Due to the relatively small volume of SNPs, a reference genome model that contains only the list of mutations can be used to reduce storage and computation costs. Using a compact reference form, such as the $1,000$ Genomes Project variant call format[5], the human genome can be represented using only about 120 megabytes, while the entire (raw) representation takes up to 200 gigabytes. This work focuses on these two representations: the *whole genome*, and the compact *SNP*-based. We defer other genomic representation formats, e.g., Short Tandem Repeat and Restriction Fragment Length Polymorphism , to future work.

[4]Although 4-bit is enough to represent all 16 combinations, we use the standard `uint8` type to contain any insertions/deletions in future work.

[5]See: www.internationalgenome.org/wiki/Analysis/vcf4.0

6.2.2 Cryptographic Commitments

A commitment scheme is a cryptographic primitive that involves a prover and a verifier. The prover first commits to a chosen (secret) value and reveals it later to the verifier. A commitment scheme thus has two phases. In the first, *commit*, phase, the prover sends a message (*commitment*) to the verifier committing to a secret value. The commitment must reveal no information about the committed value; this is called the *hiding* property. In the second, *reveal*, phase, the prover sends to the verifier a message (*decommitment*) which reveals the previously committed value. The verifier validates the revealed value against the committed one. The former must uniquely match the latter, called the *binding* property. We denote a commitment using $Com(z; r)$, where Com is a commitment scheme, z is the committed value, and r is a random bit-string used as a salt.

We use two types of commitments: one based on the discrete logarithm problem (DLP) and the other – based on a strong cryptographic hash function. For DLP-based commitments, we use the well-known Pedersen [245] and Fujisaki-Okamoto [147] commitment schemes.

A Pedersen commitment is defined in a subgroup of $\mathbb{Z}_p^*$, where p is a large prime, albeit any group where the DLP is hard can be used. Let $\mathbb{G}$ be the group of order p, generated by element g, i.e., $\mathbb{G} = \langle g \rangle$ with $o(G) = p$. Two non-identity elements, g and h, in $\mathbb{G}$ are used as public keys, where $log_g h$ is unknown to either the prover or the verifier. To commit to a message $z \in \mathbb{Z}_p$, a Pedersen commitment is computed as: $Com = Com(z; r) = g^z h^r$, where r is a random number in $\mathbb{Z}_p$. To decommit, later prover reveals z and r, and the verifier checks if indeed $g^z h^r = Com$.

A Fujisaki-Okamoto commitment is an extension of the Pedersen commitment to the RSA setting. Instead of Z_P^*, it uses Z_N^*, where $N = PQ$ and P, Q are distinct primes, such that $P = 2p + 1$ and $Q = 2q + 1$, where p and q are also primes. Now, two generators g_p and g_q generate each group, $\mathbb{G}_p$ and $\mathbb{G}_q$, of prime orders p and q, respectively, i.e., $\mathbb{G}_p$ and $\mathbb{G}_q$ are subgroups of $\mathbb{Z}_P^*$ and $\mathbb{Z}_Q^*$, respectively. Generator g of the group $\mathbb{G}_{pq}$ is computed using the Chinese Remainder Theorem, such that $g = g_p \pmod{P}$ and $g = g_q \pmod{Q}$. Then, h is computed by $h = g^\alpha \pmod{N}$, where α is uniformly chosen from $\mathbb{Z}_{pq}^*$, and (g, h) is set as the public key. To commit a message $z \in \mathbb{Z}_N$, the Fujisaki-Okamoto commitment is computed by: $Com(z;r) = g^z h^r \pmod{N}$, where r is a random number in $\mathbb{Z}_{\lambda N}^*$ with a security parameter λ. Both Pederson and Fujisaki-Pkamoto commitment schemes are statistically hiding and computationally binding.

Finally, the commitments based on secure cryptographic hash functions rely on the fact that modern hash functions practically reveal no information about the value they *hide* if they are used with a sufficiently long random salt. In addition, since secure cryptographic hash functions offer weak or strong collision-resistance properties, the commitments based on such functions are *binding*. A commitment is of the form: $H(z||r)$, where H is a cryptographically-secure hash function (e.g., SHA2 [20]), z is the committed value, and r is a sufficiently long random bit-string.

6.2.3 Zero-Knowledge Range Proofs

A Zero-Knowledge Range Proof (ZKRP) allows a prover to convince a verifier that a committed secret value is within a given range without revealing that value. The three standard zero-knowledge proof (ZKP) properties: *completeness*, *soundness*, and *zero-knowledge*, also hold for ZKRPs. i.e., When the secret is in the given range, an honest prover can always convince an honest verifier of the fact, which yields *completeness*. Meanwhile, no dishonest prover can convince an honest verifier if the secret is not in the range, which yields *soundness*. Also, the verifier learns nothing from the execution of the protocol about the secret value other than that it lies in the range, which corresponds to *zero-knowledge*. *Non-Interactive Zero-Knowledge Proofs (NIZKs)* are a class of ZKP that requires no interaction between the prover and the verifier. It is well-known that NIZK can be constructed from ZKP in a random oracle model using the Fiat-Shamir heuristic [141].

Boudot's range proof [64] is the first practical construction of ZKRP proposed in 2001. It includes two protocols: one for the extended range and the other for the exact range. In the first, for a requested range $[a, b]$, the prover shows that the secret integer v resides in $[a - \theta, b + \theta]$, where $\theta = 2^{t+l+1}\sqrt{b-a}$ and t and l are security parameters, i.e., with the expansion rate[6] $\delta = 2\theta$. Whereas, in the second protocol, the prover shows that v resides in the exact requested range, $[a, b]$, i.e., with $\delta = 1$.

Following Boudot, there has been a long line of work on ZKRPs. As mentioned in survey papers such as [231, 119], ZKRP methods can be classified based on their main characteristics: (1) *square decomposition*, (2) *signature-based*, (3) *multi-based decomposition*, and (4) *Bulletproofs*.

Square decomposition constructions, such as [64, 213, 159], use the fact that any non-negative integer can be represented by the sum of squares. *Signature-based constructions* [72] are generalized from a zero-knowledge set membership test by showing that the prover knows a signature on the secret among the entire set of signatures on integers in the given range. In *multi-based decomposition constructions* [214, 74], the secret is decomposed by a u-ary (usually binary) representation, and the prover shows that each coefficient is 0 or 1, which proves that the secret value is in a given range. Lastly, unlike other approaches, *Bulletproofs* [69] is a technique without a trusted set-up phase. It uses the binary representation of the secret value and inner product proofs, which allows for smaller proof sizes.

Any ZKRP protocol can be used in our construction. For example, we use Boudot's range proof [64] as the ZKRP of our main construction(Section 6.4) and Bulletproofs [69] in the

[6] The expansion rate of a range proof allows for tolerance, defined as $\delta = \frac{B-A}{b-a}$, where $[a, b]$ is the requested range and $[A, B]$ is a larger range including $[a, b]$ wherein the prover shows the value resides. Note that if this rate is 1, the range proof convinces a verifier that the value is exactly in the requested range $[a, b]$.

extended version (Section 6.5). We briefly summarize these protocols below.

Assume that the prover wants to show that a secret value $v \in [a, b]$ is in $[a - \theta, b + \theta]$, where θ is defined as above. To do so, the prover shows (1) $v - a \geq -\theta$ and (2) $b - v \geq -\theta$. To demonstrate (1), the prover writes $(v - a)$ as the sum of the greatest square less than v and a non-negative number, i.e., $v - a = v_1^2 + p$, where $v_1 := \lfloor \sqrt{v - a} \rfloor$ and $p = (v - a) - v_1^2$. Next, the prover shows, in zero-knowledge, that the commitment to v_1^2 hides a square, and the commitment to p hides a number with the absolute value less than θ, using the method in [81]. The same procedure is done for (2), but with r_1', r_2' such that $r_1' + r_2' = -r$. As a result, the prover shows that $v \in [a - \theta, b + \theta]$ where $\theta = 2^{t+l+1}\sqrt{b - a}$.

Showing that the secret value v is exactly in $[a, b]$ is similar. However, it is now shown that the expanded secret value lies in the expanded range of $[a, b]$, which implies that $v \in [a, b]$. More specifically, the prover first expands v by computing $v' = v \cdot 2^T$, where $T = 2(t+l+1) + |b-a|$. Then it shows that $v' \in [2^T a - \theta', 2^T b + \theta']$, where $\theta' = 2^{t+l+T/2}\sqrt{b - a}$ using the previous protocol with expanded commitment $Com(z; r)^{2^T}$. Since $\theta' < 2^T$, the verifier is convinced that $v' \in]2^T a - 2^T, 2^T b + 2^T[\iff v \in]a - 1, b + 1[$ so that $v \in [a, b]$ as $v \in \mathbb{Z}$.

With Bulletproofs [69], the prover performs ZKRP twice: once for $v \in [a, a + 2^n]$, and then for $v \in [b - 2^n, b]$, in order to show that $v \in [a, b]$. To show that $v \in [0, 2^n - 1]$, the prover first vectorizes v to n-bit value, $v_L := (v_1, ..., v_n) \in \{0, 1\}^n$. Then, for $v_R := v_L - 1^n$, the Hadamard product of v_L and v_R is zero, i.e., $v_L \circ v_R = 0$. To show these properties, the prover needs to show that:

1) the inner product $\langle v_L, 2^n \rangle = v$, 2) $\langle v_L, v_R \circ y \rangle = 0$, and 3) $\langle v_L - 1^n - v_R, y^n \rangle = 0$, for $\forall y \in \mathbb{Z}_p$

These equalities can be combined into one, by the verifier choosing a random $z \in \mathbb{Z}_p$ and prover showing that:

$$\langle v_L - z \cdot 1^n, y^n \circ (v_R + z \cdot 1^n) + z^2 \cdot 2^n \rangle = z^2 \cdot v + \delta(y, z) \tag{6.1}$$

where $\delta(y, z) = (z - z^2) \cdot \langle 1^n, y^n \rangle - z^3 \langle 1^n, 2^n \rangle \in \mathbb{Z}_p$. Also, to hide the information about v_L and v_R, additional blinding terms $s_L, s_R \in \mathbb{Z}_p^n$ are used, so that the prover sends l, r, and t instead, where:

$$\begin{aligned}
&\mathbf{l} := l(X) = v_L - z \cdot 1^n + s_L \cdot X, \\
&\mathbf{r} := r(X) = y^n \circ (v_R + z \cdot 1^n + s_R \cdot X) + z^2 \cdot 2^n \in \mathbb{Z}_p^n[X], \text{ and} \\
&t(X) := \langle l(X), r(X) \rangle \in \mathbb{Z}_p[X]
\end{aligned}$$

To convince the verifier that the constant term t_0 of $t(X) = t_0 + t_1 \cdot X + t_2 \cdot X^2$ becomes the right-hand side of the equation (6.1), i.e., $t_0 = v \cdot z^2 + \delta(y, z)$, the prover commits to the remaining coefficients, $t_1, t_2 \in \mathbb{Z}_p$, receives a random point $x \in \mathbb{Z}_p^*$ from the verifier, and replies the $t(x)$ value to the verifier. By checking all the commitments and values, the verifier is convinced that $v \in [0, 2^n - 1]$.

6.2.4 Homomorphic Encryption

Homomorphic encryption (HE) is a special type of encryption that allows users to perform certain arithmetic operations on encrypted data such that results are reflected in the plaintext. If it supports *both* unlimited addition and multiplication of ciphertexts, the HE scheme is called *Fully Homomorphic Encryption (FHE)*. Whereas, a scheme that supports a limited number of operations of either type is called *Somewhat Homomorphic Encryption (SWHE)*. Finally, a scheme that supports only one operation type (e.g., addition or multiplication) is called *Partially Homomorphic Encryption (PHE)*. We use PHE in this work. (Typically, in terms of computation costs, $FHE > SWHE > PHE$).

There are many PHE schemes, e.g., [262, 155, 136, 91, 96, 232, 240, 242, 100, 189]. For example, the well-known ElGamal encryption scheme [136] is a PHE which supports multiplication, as for any $m_1, m_2 \in \langle g \rangle$ and random r_1, r_2,

$$Enc(m_1) * Enc(m_2) = (g^{r_1}, m_1 * h^{r_1}) * (g^{r_2}, m_2 * h^{r_2}) = (g^{r_1+r_2}, m_1 * m_2 * h^{r_1+r_2}) = Enc(m_1 m_2)$$

Another variant of ElGamal [96] is additively homomorphic. It uses g^m instead of m as the ciphertext. i.e.,

$$Enc(m_1) * Enc(m_2) = (g^{r_1}, g^{m_1} * h^{r_1}) * (g^{r_2}, g^{m_2} * h^{r_2}) = (g^{r_1+r_2}, g^{m_1+m_2} * h^{r_1+r_2}) = Enc(m_1 + m_2)$$

Another popular PHE is Paillier [242], which also supports addition. In it:

$$\begin{aligned} Enc(m_1) * Enc(m_2) &= (g^{m_1} r_1^n \pmod{n^2}) * (g^{m_2} r_2^n \pmod{n^2}) \\ &= g^{m_1+m_2} (r_1 + r_2)^n \pmod{n^2} = Enc(m_1 + m_2) \end{aligned}$$

where r_1, r_2 are randomly chosen elements in $\mathbb{Z}_n^*$, for any $m_1, m_2 \in \mathbb{Z}_n$. Although Paillier is widely used, we show in Section 6.7 that additively homomorphic ElGamal (AH-ElGamal) scheme is more efficient in our context. (This is mainly because we only need to check if the ciphertext is the encryption of zero.)

6.3 System & Security Models

6.3.1 System Model

The system model includes three types of entities: (1) individuals, (2) sequencing labs, and (3) testers, where each entity's role is as follows:

1. Each individual provides their DNA sample to a sequencing lab.
2. Sequencing lab is a service provider, certified by a trusted authority, that extracts and generates the digitized genomic data from the received DNA sample, e.g., hospitals and Direct-To-Consumer (DTC) service providers.
3. Tester offers various genomic tests, e.g., paternity, pharmacogenetics, or cancer marker screenings, which entail one or more queries for genomic data on some specific locations required for a test. The exact locations for each test are certified by a trusted authority. Each query represents a range $Q = [a, b]$ with genomic (integer) positions a and b, which indicates that the tester needs all genomic data in Q to perform the test.

We assume a global pre-existing public key infrastructure (PKI) establishes trust among entities based on certified public keys. Although there would be a multitude of each entity type, we assume (for the sake of clarity) only one individual (*Alice*), one sequencing lab (SL), and one tester (T).

6.3.2 Security Model

We assume that SL and T are certified by a trusted authority and SL is fully trusted by Alice and T. Sequencing and preparing digitized genomic data by SL are performed offline. We assume T is Honest-but-Curious (HbC): although it faithfully follows the protocol, it aims to learn more about Alice's genome than is required for the test. The ranges queried by T are considered to be pre-approved by a trusted authority. For instance, the trusted authority provides T a signed white list of legitimate ranges or genome positions for all authorized genomic tests. For this reason, Alice is willing to reveal the required genomic data to T for the given test.

We assume that T does not trust Alice since she might supply altered genomic data (whether by modification and/or omission) to influence the outcome of a test. For correct test results, T needs to ensure that all information Alice supplies is both *authentic* and *complete*. To this end, our goal is a **secure and private range query protocol,** π**,** between Alice and T. π takes T's range query Q and Alice's set of SNPs $\mathcal{M}$ as inputs, and outputs the response $\mathcal{M}_Q$ to T, i.e., the set of all SNPs in $\mathcal{M}$ located in Q. Concrete security goals are:

1. **Authenticity:** All SNPs reported by Alice must be authentic, i.e., Alice cannot modify any part of her digitized genome or introduce new parts without being detected.
2. **Completeness:** All SNPs in Q should be reported, i.e., Alice should not omit any SNP in Q from her reply to T's query.
3. **Alice's Privacy:** T should not learn any information about Alice's genome beyond SNPs in Q.

Caveat: Nucleotides (individual bases) at different positions can be correlated. Some such correlations are well-known. Therefore, based on the specific mutations in a given range that Alice reveals to T as part of a legitimate test, T could infer genomic information from other regions of Alice's genome. We believe that such side-channel inference is unavoidable and consider it out of the scope of this work.

6.4 Genomic Range Query Protocol

6.4.1 Intuitive Approaches

One trivial approach is to use Alice's whole genome, G_{Alice}, and let SL sign all g_i's at sequencing time. When T asks for all mutations in Q, Alice provides all pairs within that range along with their signatures. T can easily detect if anything is missing since it receives all g_i's. Including fake bases is impossible, as Alice cannot forge SL's signatures. This approach provides authenticity and integrity, and leaks no information about bases outside the queried range. However, it has high computation and storage costs because every single base in Q needs to be signed, sent, and verified. To reduce costs, certain cryptographic methods, such as condensed and aggregated signatures can be employed, albeit the final cost would still be far from optimal. We refer to [65] for a detailed discussion and comparison of such methods.

Another intuitive approach is to use Alice's SNP-represented genome, $\mathcal{M}$, and let SL sign all m_i's at sequencing time. This would substantially reduce storage and computation costs due to the relatively small volume of SNPs. However, it introduces the completeness problem, since nothing prevents Alice from omitting some (or all) SNPs when she sends T the mutations and signatures on Q. To avoid any omissions and ensure the correct ordering of mutations, SL can sign tuples consisting of two adjacent mutations sorted in ascending order, as suggested by [122]. However, this entails revealing two tuples containing mutations in positions immediately beyond lower and upper range boundaries, respectively. Due to the general sparsity of genomic mutations, this could leak a substantial amount of sensitive information to T.

6.4.2 Proposed Approach

Assume that Alice gives her physical DNA sample (Bio_{Alice}) to SL to obtain a digital representation of her genome in SNPs. SL forms the SNP representation $\mathcal{M} = \{m_1, m_2, ..., m_n\}$ by comparing Alice's genome to a reference genome, as discussed in Section 6.2.1. Now, SL adds two special mutations, m_0 and m_{n+1}, to represent the lower and upper genome boundaries. At the end of the sequencing process, Alice receives the list:

$$\widehat{\mathcal{M}} = \{m_0, m_1, m_2, ..., m_n, m_{n+1}\} = \mathcal{M} \cup \{m_0, m_{n+1}\}$$

For the special sentinel mutations m_0 and m_{n+1}, positions pos_0 and pos_{n+1} are integers out of the normal human genomic position range, i.e., $pos_0 = -\infty < 0$, $pos_{n+1} = +\infty > N$, and the base letters l_0 and l_{n+1} are dummy letters of the same sizes.

Alice also receives the lists of signatures, $\Gamma = \{\gamma_0, \gamma_1, ..., \gamma_n\}$, and salts, $Salt$, used to generate these signatures. To show the ordering of mutations without revealing additional information on boundary ones, SL signs *the tuple of commitments* for each adjacent mutation instead of signing each mutation, i.e., $\gamma_i = Sign(sk_{SL}, Tup_i)$, for $\forall i$. Each tuple consists of four commitments for the adjacent mutations and their positions:

$$Tup_i = (Com(pos_i; s_{i,1}), Com(m_i; s_{i,2}), Com(pos_{i+1}; s_{i+1,1}), Com(m_{i+1}; s_{i+1,2}))$$

for some commitment scheme Com and salts $s_{i,j}$ for i-th SNP and $j = 1, 2$. The latter two commitments for pos_{i+1} and m_{i+1} are reused in the next tuple, Tup_{i+1}. This can be done in the offline phase between SL and Alice.

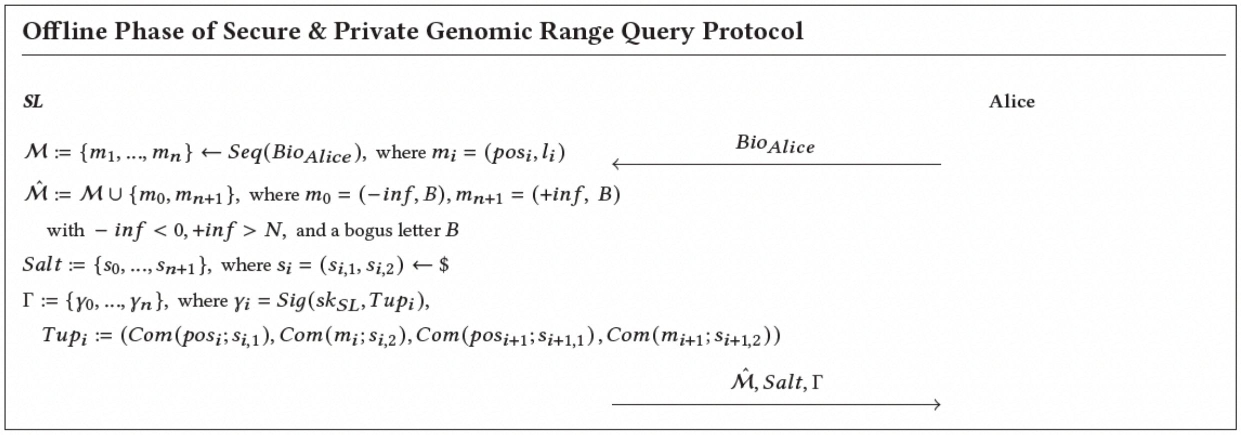
In the online phase, Alice returns the mutations within the queried range Q, along with the corresponding signatures and salts. She also generates two range proofs for the positions of the first mutations outside the queried range: one for the mutation with the highest position below the lower bound of Q and the other for the mutation with the lowest position above the upper bound of Q. Alice sends these proofs and corresponding commitments of those positions. Denoting all SNPs in Q as $\mathcal{M}_Q = \{m_{k+1}, ..., m_{k+j}\}$, Alice sends: $(\mathcal{M}_Q, Salt_Q, \Gamma_Q, C_k, C_{k+j+1}, l, h)$ to T, where:

$$\begin{aligned}
Salt_Q &= \{s_{k+1}, ..., s_{k+j}\} = \{(s_{k+1,1}, s_{k+1,2}), ..., (s_{k+j,1}, s_{k+j,2})\},\\
\Gamma_Q &= \{\gamma_k, \gamma_{k+1}, ..., \gamma_{k+j}\},\\
C_k &= (Com(pos_k, s_{k,1}), Com(m_k, s_{k,2})),\\
C_{k+j+1} &= (Com(pos_{k+j+1}, s_{k+j+1,1}), Com(m_{k+j+1}, s_{k+j+1,2})),\\
l \leftarrow \texttt{NIZKRP\_Prove}&\{(pos_k, s_{k,1}) \mid C_{k,1} = Com(pos_k, s_{k,1}) \wedge pos_k < a\}\\
h \leftarrow \texttt{NIZKRP\_Prove}&\{(pos_{k+j+1}, s_{k+j+1,1}) \mid C_{k+j+1,1} = Com(pos_{k+j+1}, s_{k+j+1,1}) \wedge pos_{k+j+1} > b\}
\end{aligned}$$

where l and h are the proofs for the non-interactive zero-knowledge proof of range proof (NIZKRP). T reconstructs intermediate tuples using received mutations and salts, and boundary tuples – using received C_k and C_{k+j+1}, as follows:

$$\begin{aligned}
&Tup_k = (C_k, Com(pos_{k+1}; s_{k+1,1}), Com(m_{k+1}; s_{k+1,2})),\\
&Tup_i := (Com(pos_i; s_{i,1}), Com(m_i; s_{i,2}), Com(pos_{i+1}; s_{i+1,1}), Com(m_{i+1}; s_{i+1,2})),\\
&\qquad\qquad \text{for } i = k+1, ..., k+j-1, \text{ and}\\
&Tup_{k+j} = (Com(pos_{k+j}; s_{k+j,1}), Com(m_{k+j}; s_{k+j,2}), C_{k+j+1})
\end{aligned}$$

Then, T verifies the signatures using SL's public key and NIZKRP proofs l, h with the boundaries of Q, a, and b. i.e., T sees if (1) $Verify(pk_{SL}, Tup_i, \gamma_i) = 1$ for all $i = k, ..., k+j$, (2) $\texttt{NIZKRP\_Verify}(C_{k,1}, l) = 1$, and (3) $\texttt{NIZKRP\_Verify}(C_{k+j+1,1}, h) = 1$, and it aborts if any of those fails. Otherwise, it proceeds with the test using received $\mathcal{M}_Q$. Figures 6.1 and 6.2 show offline and online phases, respectively. Communication and computation costs are reflected in Table 6.2, where n is the number of Alice's SNPs and j is the number of mutations in Q.



Offline Phase of Secure & Private Genomic Range Query Protocol

SL | **Alice**

$\mathcal{M} := \{m_1, ..., m_n\} \leftarrow Seq(Bio_{Alice})$, where $m_i = (pos_i, l_i)$ $\xleftarrow{Bio_{Alice}}$

$\hat{\mathcal{M}} := \mathcal{M} \cup \{m_0, m_{n+1}\}$, where $m_0 = (-inf, B), m_{n+1} = (+inf,\ B)$

with $-\ inf < 0, +inf > N$, and a bogus letter B

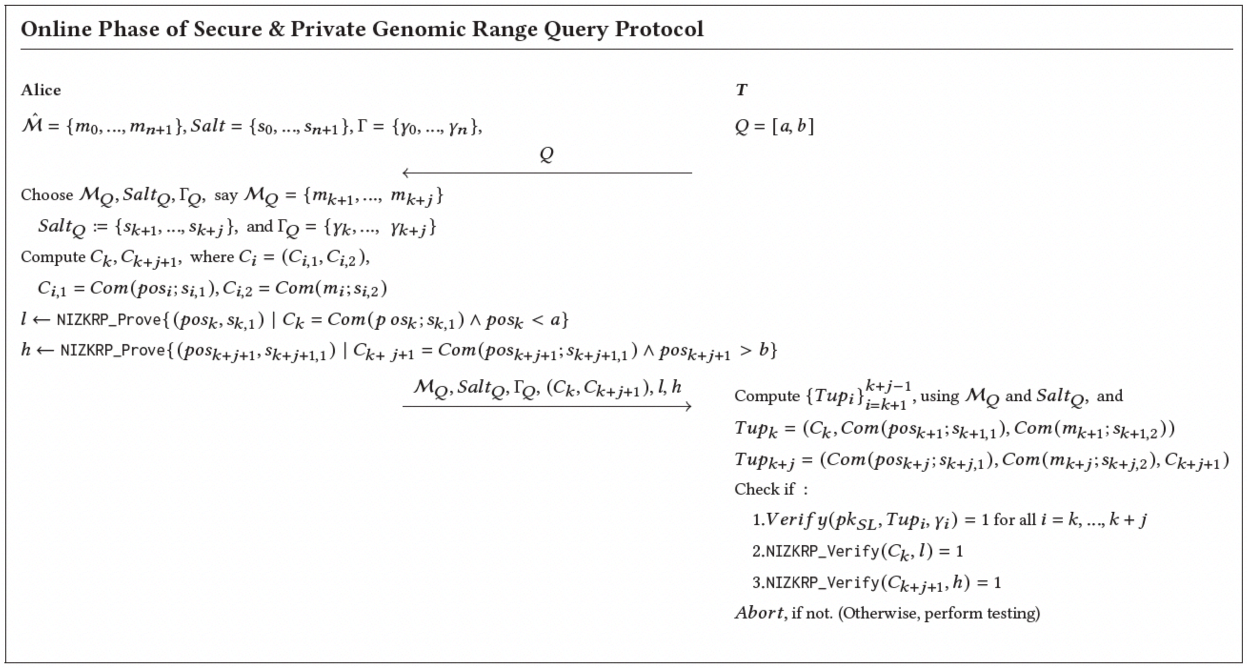
$Salt := \{s_0, ..., s_{n+1}\}$, where $s_i = (s_{i,1}, s_{i,2}) \leftarrow \$$

$\Gamma := \{\gamma_0, ..., \gamma_n\}$, where $\gamma_i = Sig(sk_{SL}, Tup_i)$,

$Tup_i := (Com(pos_i; s_{i,1}), Com(m_i; s_{i,2}), Com(pos_{i+1}; s_{i+1,1}), Com(m_{i+1}; s_{i+1,2}))$

$\xrightarrow{\hat{\mathcal{M}}, Salt, \Gamma}$

Figure 6.1: (Offline Phase) Digitizing Alice's SNPs



Online Phase of Secure & Private Genomic Range Query Protocol

Alice | ***T***

$\hat{\mathcal{M}} = \{m_0, ..., m_{n+1}\}, Salt = \{s_0, ..., s_{n+1}\}, \Gamma = \{\gamma_0, ..., \gamma_n\}$, | $Q = [a, b]$

$\xleftarrow{Q}$

Choose $\mathcal{M}_Q, Salt_Q, \Gamma_Q$, say $\mathcal{M}_Q = \{m_{k+1}, ...,\ m_{k+j}\}$

$Salt_Q := \{s_{k+1}, ..., s_{k+j}\}$, and $\Gamma_Q = \{\gamma_k, ...,\ \gamma_{k+j}\}$

Compute C_k, C_{k+j+1}, where $C_i = (C_{i,1}, C_{i,2})$,

$C_{i,1} = Com(pos_i; s_{i,1}), C_{i,2} = Com(m_i; s_{i,2})$

$l \leftarrow$ `NIZKRP_Prove`$\{(pos_k, s_{k,1}) \mid C_k = Com(pos_k; s_{k,1}) \wedge pos_k < a\}$

$h \leftarrow$ `NIZKRP_Prove`$\{(pos_{k+j+1}, s_{k+j+1,1}) \mid C_{k+j+1} = Com(pos_{k+j+1}; s_{k+j+1,1}) \wedge pos_{k+j+1} > b\}$

$\xrightarrow{\mathcal{M}_Q, Salt_Q, \Gamma_Q, (C_k, C_{k+j+1}), l, h}$

Compute $\{Tup_i\}_{i=k+1}^{k+j-1}$, using $\mathcal{M}_Q$ and $Salt_Q$, and

$Tup_k = (C_k, Com(pos_{k+1}; s_{k+1,1}), Com(m_{k+1}; s_{k+1,2}))$

$Tup_{k+j} = (Com(pos_{k+j}; s_{k+j,1}), Com(m_{k+j}; s_{k+j,2}), C_{k+j+1})$

Check if :

1.$Verify(pk_{SL}, Tup_i, \gamma_i) = 1$ for all $i = k, ..., k+j$

2.`NIZKRP_Verify`$(C_k, l) = 1$

3.`NIZKRP_Verify`$(C_{k+j+1}, h) = 1$

$Abort$, if not. (Otherwise, perform testing)

Figure 6.2: (Online Phase) Genomic Range Query between Alice and Tester (T)

6.4.3 Security Analysis

Assume a non-empty $\mathcal{M}_Q = \{m_{k+1}, ..., m_{k+j}\}$ for some non-negative integers k, j. T checks the authenticity of $m_i \in \mathcal{M}_Q$ by verifying the received signatures $\Gamma_Q = \{\gamma_i\}_{i=k}^{k+j}$ using the reconstructed tuples $\{Tup_i\}_{i=k}^{k+j}$. The links between two adjacent tuples prevent Alice from excluding any mutations. Also, T ensures completeness by verifying the two NIZKRP proofs, showing that the boundary position commitments hide the integers beyond the queried range. This allows Alice to maintain the privacy of all mutations outside Q.

Suppose that $\mathcal{M}_Q$ is empty, i.e., Alice has no SNPs within Q. Then, Alice sends one tuple Tup_l for some l, of the form: $(Com(pos_l), Com(m_l), Com(pos_{l+1}), Com(m_{l+1}))$ such that $pos_l < a$ and $pos_{l+1} > b$ and its signature γ_l to T. T verifies γ_l, which satisfies authenticity, and checks the ZKPs that committed values are outside Q, ensuring completeness and Alice's privacy.

One special case occurs when no mutations exist before position a and/or after position b. The required range is large enough, and it reveals all SNPs to conduct the test with $m_k = m_0$, and/or $m_{k+j+1} = m_{n+1}$. Since sentinel commitments are indistinguishable from other commitments, our security goals are also achieved in this case. In summary,

Goal 1. Authenticity is based on the security of the underlying digital signature scheme used by SL to sign tuples.

Goal 2. Completeness is achieved by sequential linking of elements, allowing T to detect any omissions.

Goal 3. Alice's Privacy holds due to the use of ZKP and the *hiding* property of commitments, which reveals no information about either mutations' positions or mutations outside Q.

Table 6.2: Cost Analysis of Secure & Private Genomic Range Query Protocol

(From → To)	**Communication Cost**
$SL \to$ Alice	$(n+2)$ mutations, $(n+1)$ signatures, and $2(n+1)$ salts
$T \to$ Alice	two integers, a and b, denoting the range $Q = [a, b]$
Alice $\to T$	j mutations, $2j$ salts, $(j+1)$ signatures, 4 commitments, 2 range proofs

Entity	**Computation Cost**
SL (Offline)	$2(n+2)$ commitments, $(n+1)$ signatures
T	$4j$ commitments, $(j+1)$ signature verifications, 2 range proof verifications
Alice	4 commitments, 2 range proof generations

6.5 Other Applications

In this section, we show how to improve a private substring matching protocol for genomic data using our technique. SPH-PSM [103] operates on encrypted bases and allows the genome owner (Alice) to learn whether a test pattern (a list of contiguous bases on some specific locations required for a given test) held by a tester (T) exists in her genome while not revealing anything about their inputs to each other. Though this protocol provides privacy for Alice's genome and T's pattern, it guarantees neither the authenticity nor integrity of Alice's genome. Furthermore, SPH-PSM incurs high communication and computational costs since it requires Alice to encrypt her whole genome and send the entire encrypted genome to T.

We denote our proposed protocols variants with the following acronyms: *secure SPH-PSM (S-SPH-PSM)*, *efficient and secure SPH-PSM (ES-SPH-PSM)*, and *flexible, efficient, and secure SPH-PSM (FES-SPH-PSM)*, respectively. Table 6.3 provides a high-level comparison of these variants over SPH-PSM.

Table 6.3: Comparisons of SPH-PSM variants

	Genomic Representation	**Security (for T)**	**Privacy**	
			Alice	T
SPH-PSM [103]	Whole	✗	✓	✓
S-SPH-PSM (§6.5.2)	Whole	✓	✓	✓
ES-SPH-PSM (§6.5.3)	SNP	✓	✓	✓
FES-SPH-PSM (§6.5.4)	SNP	✓	✓	◗

✓ and ✗ denotes supported and unsupported, respectively, and ◗ denotes degree of support can vary.

6.5.1 Size- and Position-Hiding Private Substring Matching Protocol (SPH-PSM) [103]

First, Alice generates a public-private key pair for an AHE scheme and encrypts each base of her genome using the public key. T computes the additive inverse of each base in its specific test pattern and encrypts each inverse using Alice's public key.

In the online phase, Alice sends the entire encrypted genome to T. For each position where the pattern is located, T homomorphically adds its encrypted pattern to Alice's encrypted base. Then, T adds the resulting values and returns the final (encrypted) sum to Alice. Alice decrypts the received ciphertext using her private key and learns the test result. The decrypted result is 0 if Alice's genome matches the test pattern; otherwise, it is a random value. During the whole process, T does not learn any information on Alice's genome or the test result.

6.5.2 Secure SPH-PSM (S-SPH-PSM)

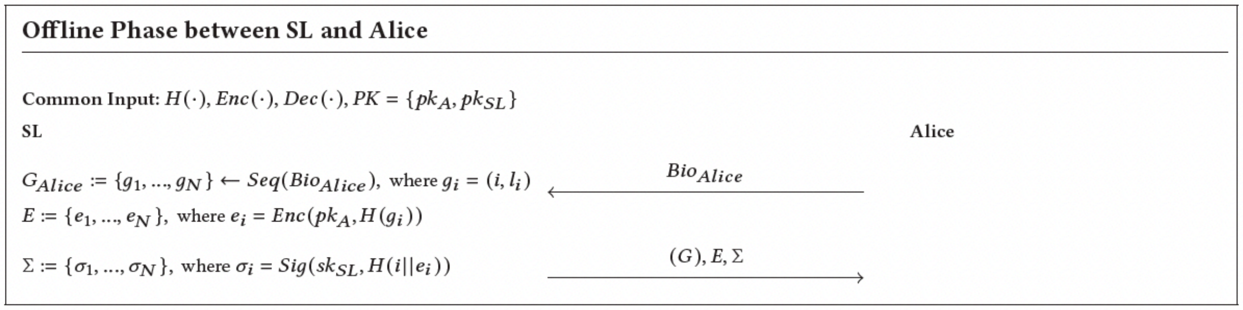
In SPH-PSM, Alice can modify her digitized genome and influence the test result, since SPH-PSM does not guarantee the authenticity or integrity of Alice's genome. To prevent this, we design *secure SPH-PSM (S-SPH-PSM)* and let SL act as a certification authority for the genomic data.

When SL sequences Alice's DNA sample (Bio_{Alice}), it encrypts the hash of each base using an AHE scheme under Alice's public key.[7] Then, it signs the hash of each ciphertext – along with its position – using its private key and sends the list of ciphertexts and corresponding signatures to Alice.

The online phase is similar to SPH-PSM, except that T verifies the signatures and checks the authenticity and integrity of ciphertexts on the positions required for the test. Figures 6.3 and 6.4 show the offline and online phases of S-SPH-PSM, respectively.

S-SPH-PSM also ensures authenticity and completeness via signing of both ciphertext and its position. T can detect omissions or rearrangements, as it verifies signatures for all consecutive

[7]The choice of this public key depends on who will learn the test result at the end of the protocol. For instance, for court-mandated tests, the court's public key can be used instead.



Offline Phase between SL and Alice

Common Input: $H(\cdot), Enc(\cdot), Dec(\cdot), PK = \{pk_A, pk_{SL}\}$

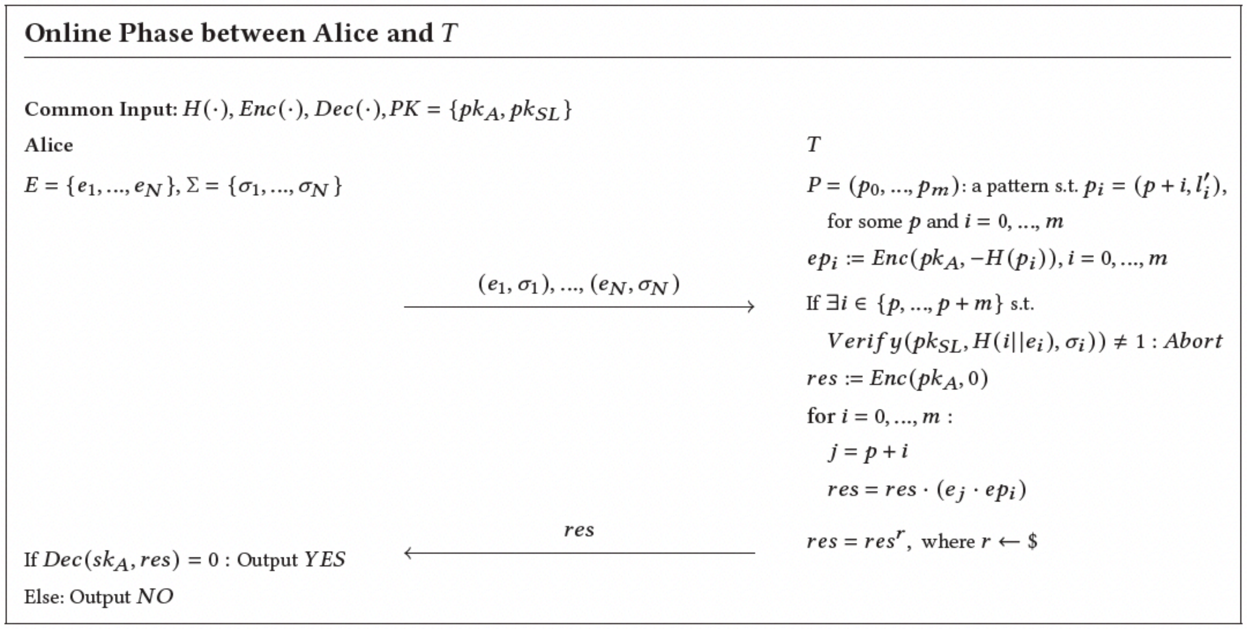
SL | **Alice**

$G_{Alice} := \{g_1, ..., g_N\} \leftarrow Seq(Bio_{Alice})$, where $g_i = (i, l_i)$ $\xleftarrow{Bio_{Alice}}$

$E := \{e_1, ..., e_N\}$, where $e_i = Enc(pk_A, H(g_i))$

$\Sigma := \{\sigma_1, ..., \sigma_N\}$, where $\sigma_i = Sig(sk_{SL}, H(i||e_i))$ $\xrightarrow{(G), E, \Sigma}$

Figure 6.3: Offline Phase of Secure SPH-PSM (S-PSH-PSM)



Online Phase between Alice and T

Common Input: $H(\cdot), Enc(\cdot), Dec(\cdot), PK = \{pk_A, pk_{SL}\}$

Alice

$E = \{e_1, ..., e_N\}, \Sigma = \{\sigma_1, ..., \sigma_N\}$

$\xrightarrow{(e_1, \sigma_1), ..., (e_N, \sigma_N)}$

$\xleftarrow{res}$

If $Dec(sk_A, res) = 0$: Output YES

Else: Output NO

T

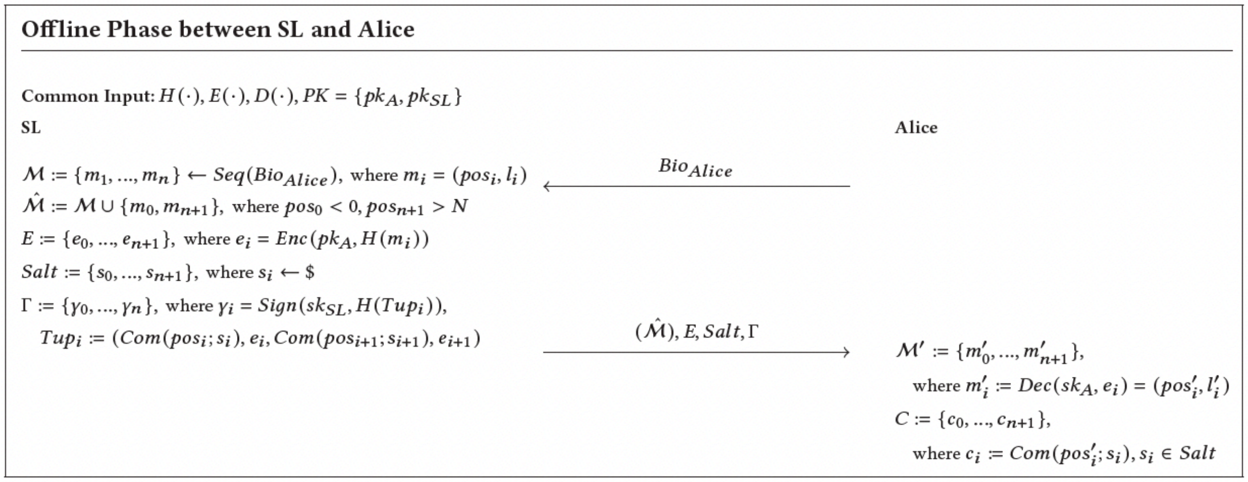
```
$P = (p_0, ..., p_m)$: a pattern s.t. $p_i = (p + i, l'_i)$,
  for some $p$ and $i = 0, ..., m$
$ep_i := Enc(pk_A, -H(p_i)), i = 0, ..., m$
If $\exists i \in \{p, ..., p+m\}$ s.t.
  $Verify(pk_{SL}, H(i||e_i), \sigma_i)) \neq 1 : Abort$
$res := Enc(pk_A, 0)$
for $i = 0, ..., m$ :
  $j = p + i$
  $res = res \cdot (e_j \cdot ep_i)$
$res = res^r$, where $r \leftarrow \$$
```

Figure 6.4: Online Phase of Secure SPH-PSM (S-SPH-PSM)

positions in the range. Also, to prevent accidental matches, we hash both the position and the base letter when encrypting the bases, as in the AH-ElGamal-based protocol [103].

6.5.3 Efficient & Secure SPH-PSM (ES-SPH-PSM)

We design *efficient and secure SPH-PSM protocol (ES-SPH-PSM)* to improve efficiency. We use SNPs instead of the whole genome representation and our techniques proposed in Section 6.4. Offline and online phases of this protocol are given in Fig. 6.5 and Fig.6.6, respectively. In the former, we add special sentinel SNPs and sign the tuples of adjacent bases,



Offline Phase between SL and Alice

Common Input: $H(\cdot), E(\cdot), D(\cdot), PK = \{pk_A, pk_{SL}\}$

SL **Alice**

$\mathcal{M} := \{m_1, ..., m_n\} \leftarrow Seq(Bio_{Alice})$, where $m_i = (pos_i, l_i)$ $\xleftarrow{Bio_{Alice}}$

$\hat{\mathcal{M}} := \mathcal{M} \cup \{m_0, m_{n+1}\}$, where $pos_0 < 0, pos_{n+1} > N$

$E := \{e_0, ..., e_{n+1}\}$, where $e_i = Enc(pk_A, H(m_i))$

$Salt := \{s_0, ..., s_{n+1}\}$, where $s_i \leftarrow \$$

$\Gamma := \{\gamma_0, ..., \gamma_n\}$, where $\gamma_i = Sign(sk_{SL}, H(Tup_i))$,

$Tup_i := (Com(pos_i; s_i), e_i, Com(pos_{i+1}; s_{i+1}), e_{i+1})$ $\xrightarrow{(\hat{\mathcal{M}}), E, Salt, \Gamma}$

$\mathcal{M}' := \{m'_0, ..., m'_{n+1}\}$,

where $m'_i := Dec(sk_A, e_i) = (pos'_i, l'_i)$

$C := \{c_0, ..., c_{n+1}\}$,

where $c_i := Com(pos'_i; s_i), s_i \in Salt$

Figure 6.5: Offline Phase of Efficient & Secure SPH-PSM (ES-SPH-PSM) and Flexible, Efficient, & Secure SPH-PSM (FES-SPH-PSM)

as in Section 6.4, whereas the tuple consists of commitments of position and the ciphertext of each SNPs instead. SL sends to Alice the encrypted genomic data and signatures along with the salts used for the commitments. In the online phase, T verifies “all" received tuples and computes multiple results for all possible starting positions. The computational complexity of this approach is $\mathcal{O}(nm)$, where n is the size of Alice's genome and m is the number of bases in the pattern that T holds. This is because SNP positions are hidden, unlike in the whole genome representation. (See unoptimized commented-out pseudo-code in Figure 6.6).

However, this computational cost can be significantly improved and reduced to $\mathcal{O}(n)$ with some optimizations. First, T performs an initial calculation as before by matching the first m (encrypted) SNPs with its m (encrypted) inverses of the pattern. T then keeps the sum of this operation in a temporary result. Since most encrypted inverses and SNPs are reused and aggregated in the next computation, the temporary result helps reduce the computational cost. That is, the next result is computed by subtracting the first encrypted SNP and adding the next encrypted SNP to the previous result. For each computation, the temporary result is stored in the result list with randomization. Whenever the pattern and Alice's SNPs match, the aggregated result will be an encrypted zero. When Alice receives randomly permuted results, she sees if any decrypted results are zero (See Figure 6.6).

Online Phase between Alice and T

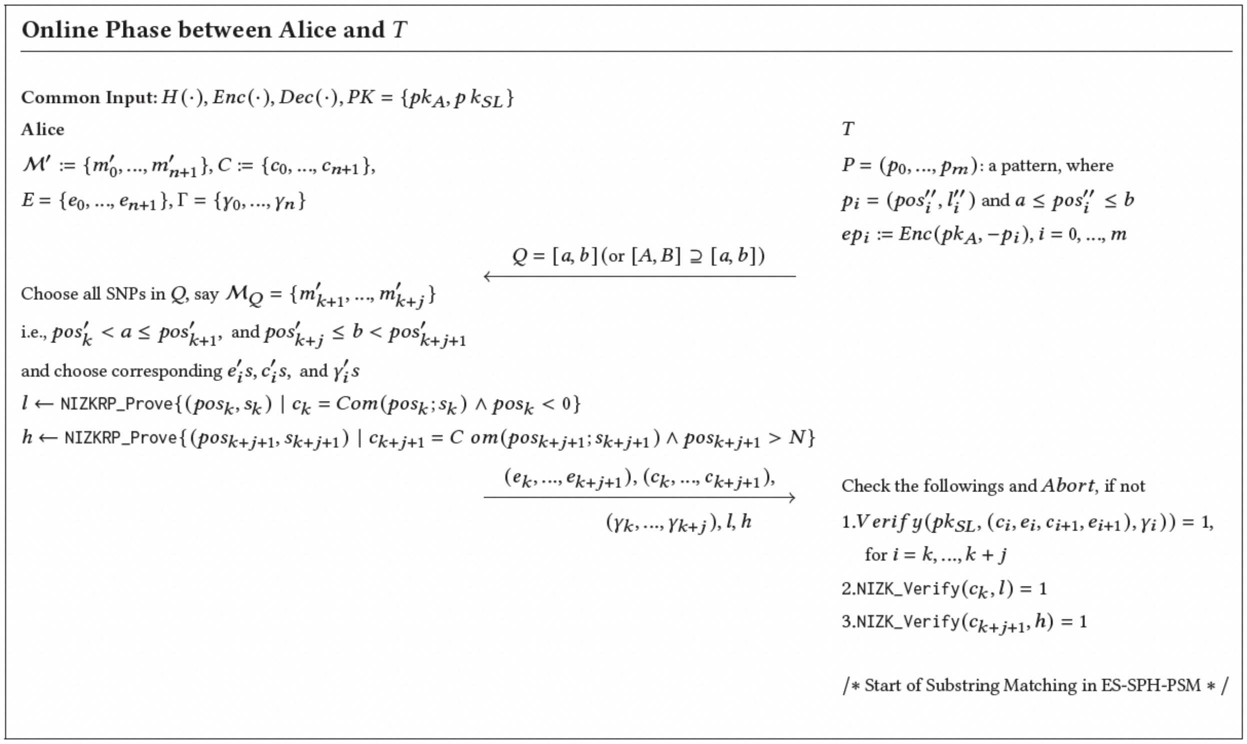
Common Input: $H(\cdot), Enc(\cdot), Dec(\cdot), PK = \{pk_A, pk_{SL}\}$

```
Alice
$\mathcal{M}' := \{m'_0, ..., m'_{n+1}\}, C := \{c_0, ..., c_{n+1}\}$,
$E = \{e_0, ..., e_{n+1}\}, \Gamma = \{\gamma_0, ..., \gamma_n\}$

T
$P = (p_0, ..., p_m)$: a pattern
where $p_i = (pos''_i, l''_i)$, for $i = 0, ..., m$
$ep_i := Enc(pk_A, -H(p_i)), i = 0, ..., m$

Alice $\xrightarrow{E, C, \Gamma, pos_0, pos_{n+1}, s_0, s_{n+1}}$ T

If $Verify(pk_{SL}, (c_i, e_i, c_{i+1}, e_{i+1}), \gamma_i)) \neq 1$
    for any $i$ : $Abort$
See if 1. $Com(pos_0; s_0) = c_0 \wedge pos_0 < 0$
and 2. $Com(pos_{n+1}; s_{n+1}) = c_{n+1} \wedge pos_{n+1} > N$

/* Start of Subtring Matching */
// Unoptimized
// for $i = 1, ..., n - m$ :
//    $res_i := Enc(pk_A, 0)$
//    for $j = 0, ..., m$ :
//        $r_j \leftarrow \$, k := i + j$
//        $res_i = res_i \cdot (e_k \cdot ep_j)^{r_j}$

// Optimized
$tmp\_res := Enc(pk_A, 0)$
for $j = 0, ..., m$ :
    $tmp\_res = tmp\_res \cdot (e_{j+1} \cdot ep_j)$
$r_1 \leftarrow \$, res_1 = (tmp\_res)^{r_1}$
for $k = m + 1, ..., n - 1$ :
    $tmp\_res = tmp\_res \cdot e_{k+1} \cdot (e_{k-m})^{-1}$
    $r_{k-m+1} \leftarrow \$, res_{k-m+1} = (tmp\_res)^{r_{k-m+1}}$
$res_{n-m+1}, ..., res_n \leftarrow \$$
$res := \pi(res_1, ..., res_n)$, where $\pi$ : a random permutation

Alice $\xleftarrow{res}$ T

If $\exists i$ s.t. $Dec(sk_A, res_i) = 0$ : Output $YES$
Else: Output $NO$
```

Figure 6.6: Online Phase of Efficient & Secure SPH-PSM (ES-SPH-PSM)

To maintain the size-hiding property, we add m additional results, encryption of random values, so that the number of results is always n, independent of the pattern size. The pattern's position-hiding property is still held by randomly shuffling multiple results before they are sent to Alice. For completeness, Alice can just reveal the boundary positions, pos_0 and pos_{n+1}. In ES-SPH-PSM, we achieve orders of magnitude efficiency improvement with the same security and privacy guarantees.

6.5.4 Flexible, Efficient, & Secure SPH-PSM (FES-SPH-PSM)



Online Phase between Alice and T

Common Input: $H(\cdot), Enc(\cdot), Dec(\cdot), PK = \{pk_A, pk_{SL}\}$

Alice

$\mathcal{M}' := \{m'_0, ..., m'_{n+1}\}, C := \{c_0, ..., c_{n+1}\}$,

$E = \{e_0, ..., e_{n+1}\}, \Gamma = \{\gamma_0, ..., \gamma_n\}$

T

$P = (p_0, ..., p_m)$: a pattern, where

$p_i = (pos''_i, l''_i)$ and $a \le pos''_i \le b$

$ep_i := Enc(pk_A, -p_i), i = 0, ..., m$

$\xleftarrow{Q = [a, b] (\text{or } [A, B] \supseteq [a, b])}$

Choose all SNPs in Q, say $\mathcal{M}_Q = \{m'_{k+1}, ..., m'_{k+j}\}$

i.e., $pos'_k < a \le pos'_{k+1}$, and $pos'_{k+j} \le b < pos'_{k+j+1}$

and choose corresponding $e'_i s$, $c'_i s$, and $\gamma'_i s$

$l \leftarrow$ `NIZKRP_Prove`$\{(pos_k, s_k) \mid c_k = Com(pos_k; s_k) \wedge pos_k < 0\}$

$h \leftarrow$ `NIZKRP_Prove`$\{(pos_{k+j+1}, s_{k+j+1}) \mid c_{k+j+1} = Com(pos_{k+j+1}; s_{k+j+1}) \wedge pos_{k+j+1} > N\}$

$\xrightarrow[(\gamma_k, ..., \gamma_{k+j}), l, h]{(e_k, ..., e_{k+j+1}), (c_k, ..., c_{k+j+1}),}$

Check the followings and $Abort$, if not

1.$Verify(pk_{SL}, (c_i, e_i, c_{i+1}, e_{i+1}), \gamma_i)) = 1$,
for $i = k, ..., k + j$

2.`NIZK_Verify`$(c_k, l) = 1$

3.`NIZK_Verify`$(c_{k+j+1}, h) = 1$

/* Start of Substring Matching in ES-SPH-PSM */

Figure 6.7: (Online Phase) Flexible, Efficient & Secure SPH-PSM (FES-SPH-PSM)

Genomic tests whose nature is known (e.g., immunity) often operate on a small range of the genome and may be public. This allows efficiency gains by querying only the mutations located in that small range. Our proposed techniques in Section 6.4 can be applied as in ES-SPH-PSM to preserve security goals for Alice's genome with adjustable privacy for the pattern. To keep the pattern's privacy (size- and position-hiding properties), a wider range including the required range can be used. The offline phase of the FES-SPH-PSM protocol is the same as the one of ES-SPH-PSM, so presented in Fig. 6.5, and the online phase is in Fig. 6.7. The two main differences are: (1) T now queries a range, and (2) Alice provides encrypted mutations only in that range with NIZKRP proofs for boundary values.

Tables 6.4 and 6.5 show the computation and communication costs of each SPH-PSM variant.

Table 6.4: Operation complexity of each SPH-PSM variant.

	Offline			**Online**	
	SL	**Alice**	**T**	**Alice**	**T**
SPH-PSM	-	N Enc	m EncInv	1 IsZero	1 Enc, MultConstant $2m$ MultCiphers
S-SPH-PSM	N Enc, Hash, Sign	-	m EncInv	1 IsZero	m SigVerify, $2m$ MultCiphers, 1 MultConstant, 1 Enc
ES-SPH-PSM	$n+2$ Enc, $n+2$ Comm, $n+1$ Hash, Sign	$n+2$ Comm	m EncInv	$n/2$ IsZero (on average)	2 BoundaryCheck, CommCheck, $n+1$ Hash, SigVerify, n Enc, $n-m+1$ MultConstant **No opt:** $2(n-m+1)*m$ MultCiphers **Optimized:** $3n-m$ MultCiphers
FES-SPH-PSM	Same as ES-SHP-PSM			ComputeBoundaryIndex, 2 RangeProofGen $k/2$ IsZero (on average)	2 RangeProofVerify, $k+1$ Hash, SigVerify, k Enc, $(k-m+1)$ MultConstant **No opt.:** $2(k-m+1)*m$ MultCiphers **Optimized:** $3k-m$ MultCiphers

where Enc: encryption of an AHE, MultConstant: $\text{Enc}(m)^r = \text{Enc}(m*r)$, MultCiphers: $\text{Enc}(m_1)$ * $\text{Enc}(m_1) = \text{Enc}(m_1+m_2)$,
EncInv: $\text{Enc}(m^{(-1)}) = \text{Enc}(-m)$, IsZero: check if the input ciphertext is encryption of zero or not, Hash: computation of a cryptographic hash function, (Sign, SigVerify): a digital signature scheme, (Comm, CommCheck): a commitment scheme, (RangeProofGen, RangeProofVerify): a NIZKRP, and (ComputeBoundaryIndex, BoundaryCheck): integer comparison

Table 6.5: Data transfer complexity of each SPH-PSM variant.

	Offline	**Online**	
	SL $\rightarrow$ Alice	**Alice $\rightarrow$ Tester**	**Tester $\rightarrow$ Alice**
SPH-PSM	-	N ciphertexts	1 ciphertext
S-SPH-PSM	N ciphertexts, signatures	N ciphertexts, signatures	Same as SPH-PSM
ES-SPH-PSM	$n+2$ ciphertexts, salts $n+1$ signatures	$n+2$ ciphertext, commitments $n+1$ signatures 2 positions (integers), salts	n ciphertexts
FES-SPH-PSM	Same as ES-SPH-PSM	$k+2$ ciphertexts, commitments $k+1$ signatures 2 range proofs	k ciphertexts

6.5.5 Discussion

Different Test Result Learner. One may question the need for security in SPH-PSM since Alice learns the test result. From a legal point of view, genomic data, as well as the result, have to be authentic. To support such cases, SPH-PSM can be updated as follows: Alice's role in the protocol can be divided into genome owner (Alice) and test requester (e.g., court). When SL encrypts the mutations, it uses the latter's public key. The rest of the protocol remains the same, but T sends the encrypted result to the court, not Alice. As a result, the court gets only the (correct) boolean result for the test, and Alice also keeps her privacy by not revealing all the genomic data to T or the court.

Genomic Similarity Testing. In genomic tests, sometimes not an exact match but a similarity score needs to be calculated (e.g., some paternity tests). S-SPH-PSM in Section 6.5.2 can be modified to support such tests by simply not aggregating the result in one ciphertext. Specifically, consider two input genomes for the similarity test, and one of the parties (Alice) learns the result. In this case, both parties encrypt their genomes under Alice's public key and send it to T, which homomorphically adds received ciphertexts for each position, shuffles the order, and sends the shuffled list to Alice. Alice checks for similarity by decrypting each entry and counting the number of decrypted zeros.

6.6 Implementation

6.6.1 Genomic Range Query Protocol

For SL's signatures, we use the Elliptic Curve Digital Signature Algorithm (ECDSA) with elliptic curve `secp256r1`, as its signatures are relatively short (512-bit), compared to other schemes with the same security level (256 bits). For commitments, we use the Fujisaki-Okamoto (FO) commitment scheme [147] that allows us to create Boudot's range proofs [64]. Parameters used for these commitments are: $s = 552$, and, for range proofs: $t = 128$, $l = 40$. For mutation commitments, we use a secure cryptographic hash function, `SHA2-256` [20], with 128-bit salts. A tuple is computed as:

$$[\ FO(pos_i, s_{i,1}), SHA2(m_i, s_{i,2}), FO(pos_{i+1}, s_{i+1,1}), SHA2(m_{i+1}, s_{i+1,2}))\ ] \tag{6.2}$$

with randomly generated salts $s_{i,1}$ in $Z^*_{\mathcal{N}}$, where $\mathcal{N}$ is a composite number with two 512-bit prime factors, and 128-bit salts $s_{i,2}$ for $SHA2$.

To implement the commitments and range proofs, we used the code from [16]. We also used the Bouncy Castle [6] crypto library for cryptographic primitives, e.g., hash functions and signatures, and Java's SecureRandom class [11] for generating salts. The code for offline and online phases was written in Java and was evaluated on a PC with an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz chip and 16GB of RAM.

6.6.2 SPH-PSM Variants

For the SPH-PSM variants, we use Pedersen commitments [245], the additively homomorphic variant of ElGamal [136] (AH-ElGamal), and Bulletproofs [69] for the NIZKRP in FES-SPH-PSM. The structures of a tuple for ES-SPH-PSM and FES-SPH-PSM are similar to the tuple (6.2) in Section 6.6.1 of secure range query protocol, except FO is replaced with Pedersen commitments, while $SHA2$ commitments are replaced with encrypted bases. The same ECDSA setting is used for SL's signatures as in the previous section.

Our codebase is in Golang [3], evaluated on a server with 64 Intel(R) Xeon(R) CPUs E5-4610 v2 @ 2.30GHz and 128GB of RAM. We modified the official Golang implementation for ElGamal to implement an AH-ElGamal. As in the previous section, we use the official Golang crypto library for `SHA256` and `ECDSA` with the curve `secp256r1` for hashing and digital signatures, respectively; we also use the range proof code from [16].

6.7 Evaluation

This section evaluates the proposed construction. For the sake of reproducibility, all our implementation and evaluation results are publicly available at [175].

6.7.1 Synthetic Genomic Data Generation

For the experiments, we generated synthetic genomic data with $3 \cdot 10^9$ bases for the whole genome and $3 \cdot 10^6$ bases for SNP representations. Each base pair is structured with a 32-bit integer representing its position and an 8-bit integer for a combination of two base letters. 8-bit integers are defined with alphabetical order of combinations, i.e., 0='AA', 1='AC',...,15='TT'. For the evaluation of Section 6.7.2, we used a single type of SNP ('AA') since our measurements do not depend on the specific SNP type. For SPH-PSM variants, we generated the necessary genomic files before each experiment and let each experiment read the required files during the evaluation. To generate a synthetic genomic file, we input the total size of the data `n`, starting and ending positions, `s` and `e`, of the pattern, and a boolean variable `isSNP` to check if it is for SNP representation or the whole genome representation. Note that which base letters are used does not affect the evaluation result. For simplicity, we generate bases with 'AA' for all other positions and bases with 'TT' for the positions in [`s`,`e`]. For T's pattern (marked with `n`=0), we generate bases with 'TT' for the positions in the range, [`s`,`e`]. If `isSNP` is `True`, we generate bases for every 1000 positions, whereas we generate bases for all integers if it is `False`.

6.7.2 Secure Genomic Range Queries

We used $3 \cdot 10^6$ SNPs for the experiment. The entire offline phase, including commitments and signature generations, took 4.2 hours on the platform above. We note that this process can be easily parallelized. Times for individual operations are:

- Fujisaki-Okamoto commitment: **3.5ms**
- Boudot's range proof: **47.7ms** for proof generation and **37ms** for validation
- SHA2 commitment: **0.3ms** (including salt creation) and **0.1ms** for validation

Salts can be alternatively (re-)generated using a seed and a Pseudorandom number generator (PRNG), or HKDF [200], a simple key derivation function based on HMAC [53], to reduce the storage cost. Verification cost incurred by T scales linearly with the number of SNPs in Q, while verification cost of the range proof is dominated by signature verification, as shown in the figure below (Figure 6.8).

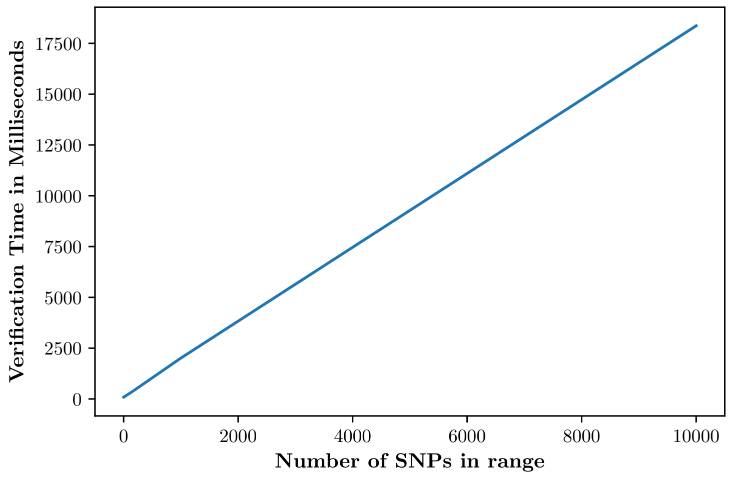


Figure 6.8: T's Verification Time given the number of SNPs in the queried range

6.7.3 SPH-PSM Variants

To assess the computational efficiency of AH-ElGamal, we first compared it to Paillier [242], a well-known additive PHE scheme, using its popular implementation [8]. Figure 6.9 shows the comparison of AH-ElGamal and Paillier for each operation. Recall that `Enc` is encryption and `EncInv` is modular inversion of encryption result. `MultCiphers` and `MultConstant` are multiplication of a ciphertext with another ciphertext and a constant, respectively. `IsZero` is used to check whether a given ciphertext is an encryption of zero. Since we do not use the full decryption operation and only check if a ciphertext is encryption of zero, results show that AH-ElGamal is significantly more efficient for all operations used in SPH-PSM variants. Thus, we use AH-ElGamal for a homomorphic encryption scheme for the rest of the evaluation.

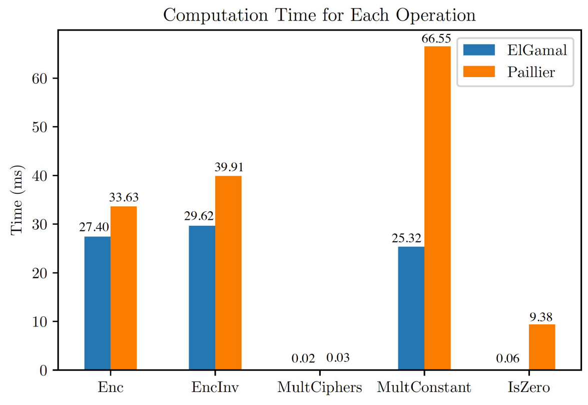


Figure 6.9: Computation time comparison between AH-ElGamal and Paillier for operations in SPH-PSM variants

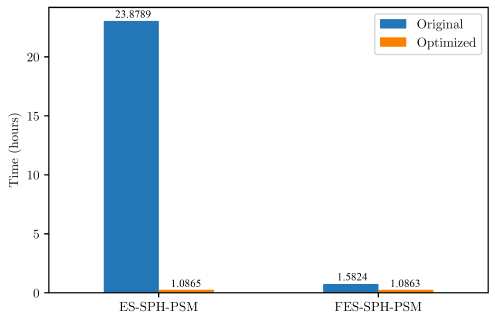


Figure 6.10: Total Runtime Comparison between Optimized vs. Original ES-SPH-PSM and FES-SPH-PSM

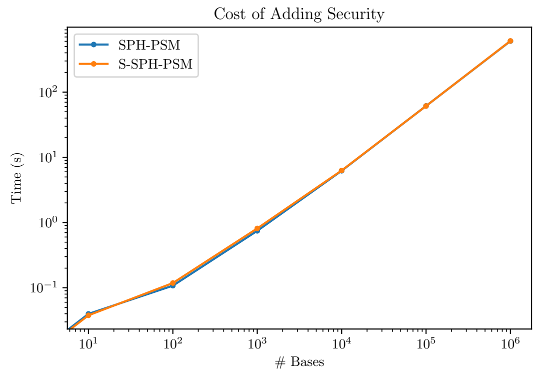


Figure 6.11: Offline Computation Cost Comparison for SPH-PSM and S-SPH-PSM

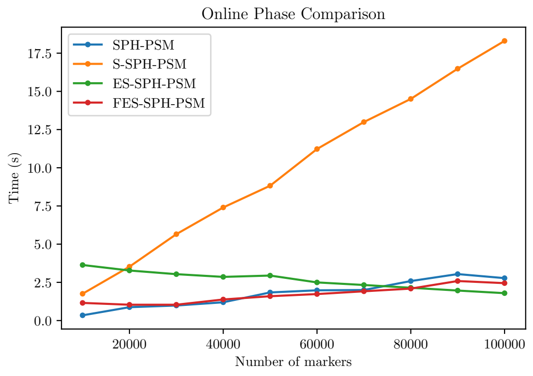


Figure 6.12: Online phase comparison of All SPH-PSM Variants

For individual operations, computation times averaged over ten executions were as follows:

- Salt Generation: **2.7** μs
- Hash computations: **4.8** μs for $H(pos_i, l_i)$, **3.9** μs for $h2 := H(pos_i, e_i)$, and

  10.4 μs for $h3 := H(Tup_i)$
- Digital Signature Generation: **91.1** μs for $sig1 := Sign(h2)$ and **70.2** μs for $sig2 := Sign(h3)$
- Digital Signature Verification: **171.4** μs for $Verify(sig1)$ and **157.6** μs for $Verify(sig2)$
- Commitment Generation: **511.5** μs
- Range Proof (for FES-SPH-PSM):

  Bulletproofs [69] : **262.2 ms** for proof generation and **168.0 ms** for validation

  Signature-based [72]: **1324.6 ms** for proof generation and **1299.4 ms** for validation

For the rest of the evaluation, we used the most efficient range proof scheme, Bulletproofs, for NIZKRP.

Next, we measured the effects of the optimization described in Section 6.5.3 for both ES-PSH-PSM and FES-SPH-PSM. We used 10^8 bases for Alice's genome and 10^7 bases for T's pattern. Figure 6.10 shows that it drastically lowers test completion times, confirming that even complex tests can be performed fast using this optimization. It also shows that FES-SPH-PSM is much more efficient than ES-SPH-PSM, i.e., when the location of the test pattern is public.

To measure the overhead of adding security to SPH-PSM, we compared the computation costs of SPH-PSM and S-SPH-PSM. We implemented the algorithm from [103] in GoLang and compared the cost for offline phases. We compared only the offline phases because the online phases are almost the same, and the only difference – signature verifications – depends on the pattern size, which is relatively small to the whole genome size. Since these signing operations are independent, we use multi-threading to parallelize them and reduce delay. Results in Figure 6.11 show that the overhead of adding security is negligible compared to the cost of homomorphic encryption operations for the whole genome.

For the online phase, we vary the number of markers and use the whole genome. Figure 6.12 shows the results: test completion time for S-SPH-PSM grows linearly with the number of markers in Alice's genome compared to SPH-PSM. This is because T verifies signatures within the range of its markers. ES-SPH-PSM and FES-SPH-PSM benefit from the optimization, and their runtimes are comparable to that of SPH-PSM, even though they add security and reduce data transfer by three orders of magnitude.

To show that proposed constructs can be parallelized, we implemented multi-threading for offline and online phases of SPH-PSM variants. We believe that this is reasonable as a consumer-facing sequencing lab would be equipped with high-end storage and computing devices capable of handling massive amounts of genomic data. Figure 6.13 compares single-threading and multi-threading with the increasing number of bases for S-SPH-PSM and ES-SPH-PSM. Since each signature generation and verification are independent, parallelization greatly reduces computation time. Specifically, S-SPH-PSM offline and online phases using multi-threading are 32 and 12 times faster, respectively, compared to using a single thread. Similarly, the offline phase of ES-SPH-PSM is 32 times faster using multi-threading with over 10^7 bases. In contrast, multi-threading only slightly improves the online phase of ES-SPH-PSM, since signature verification is the only operation that can be parallelized.

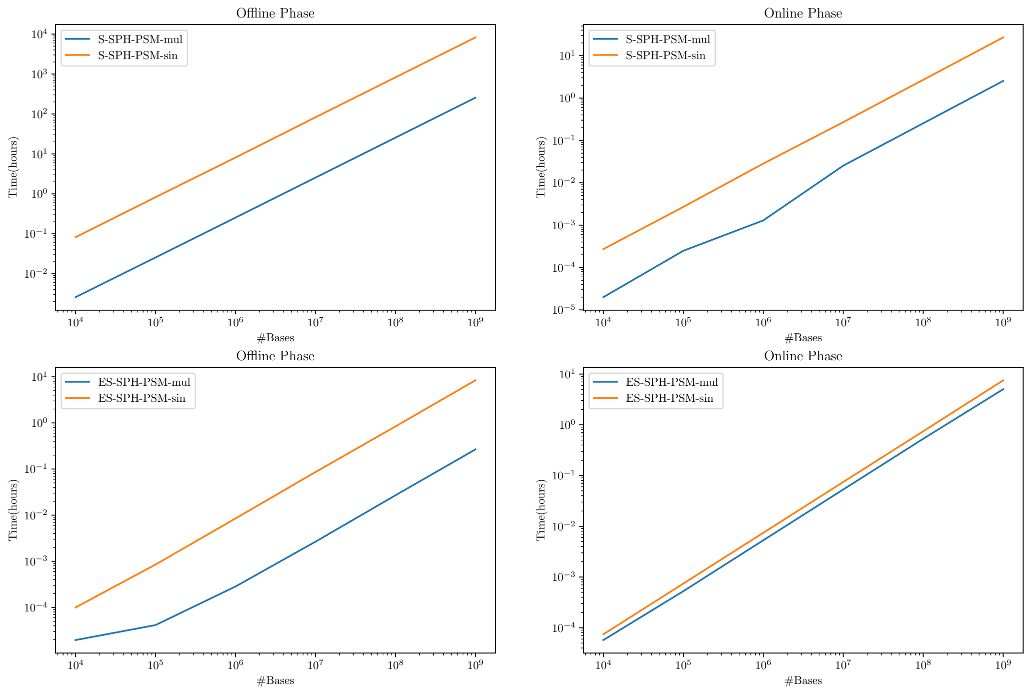


Figure 6.13: Comparison between multi-threaded and single-threaded results.

6.8 Generalization to Sparse Integers

Although our primary application domain is genomics, the proposed technique applies to other settings. For example, suppose that a forensics team, after obtaining a court order, wants to examine an Internet Service Provider (ISP)'s log regarding a particular account's activities for a time period relevant to a cyberspace attack. This setting resembles genomic testing in terms of security and privacy requirements as well as the sparsity of sensitive data. From the security perspective, the forensics team needs to ensure that the log entries supplied by the ISP for the period and account of interest are genuine (authentic) and complete. From the privacy perspective, the ISP does not want to reveal any non-relevant log entries to protect its customers' privacy and its own interests. Whereas the relevant log entries (i.e., log entries corresponding to the period and account of interest) are likely to be sparsely distributed within an extensive data set.

Similar situations might arise regarding range queries over firewall logs or ticket transaction databases. The proposed technique can be adapted to address their security and privacy challenges. This section defines a generalized version of the proposed scheme geared for secure and private range queries over sparse integers.

6.8.1 Secure & Private Range Queries over Sparse Integers

Let $\mathcal{D}$ be a domain set of consecutive integers from A to B. i.e., $\mathcal{D} = [A, B] \subseteq \mathbb{Z}$, for some $A, B \in \mathbb{Z}$. Though there are many ways to mathematically define a *sparse set* [303, 62, 217], informally, we call a subset $\mathcal{X}$ of a set $\mathcal{D}$ is *sparse* if $\frac{|\mathcal{X}|}{|\mathcal{D}|}$ is small. Note that our approach also works for the improper subset $\mathcal{X}$, i.e., when $\mathcal{X} = \mathcal{D}$, however, it is more meaningful in terms of privacy if the subset $\mathcal{X}$ is sparse in $\mathcal{D}$.

We assume Alice and Bob act as a replier and a querier, respectively. Alice has a sparse subset $\mathcal{X}$ of $\mathcal{D} = [A, B]$, with n integers, and Bob has a range query $Q = [a, b]$ such that $A \leq a \leq b \leq B$ to receive all integers in $\mathcal{X} \cap Q$. i.e., if Alice's sparse set is: $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, then Bob wants to receive all x_i's such that $a \leq x_i \leq b$.

Definition 6.1. *Let $\mathcal{D} = [A, B] \subseteq \mathbb{Z}$ be a domain set of consecutive integers, for some*

$A, B \in \mathbb{Z}$. The ideal functionality $\mathcal{F}$ of the protocol between Alice and Bob, on input $\mathcal{X}$ and Q is, $\mathcal{F} : (\mathcal{X}, Q) \rightarrow (Q, \mathcal{X} \cap Q)$, where $\mathcal{X}$ is a sparse subset of $\mathcal{D}$ and Q is a consecutive subset of $\mathcal{D}$.

Security goals are the same as in Section 6.3: authenticity, completeness, and privacy. i.e.,

1. **Authenticity.** Alice can not modify $\mathcal{X}$ after it is chosen, and Bob can check if the integers returned by Alice are the ones that Alice chose, i.e., authentic.
2. **Completeness.** Bob should be convinced that the integers returned by Alice are complete, i.e., they are within Q without any omissions.
3. **Alice's Privacy.** Bob learns no information about the integers in $\mathcal{X}$ outside Q.

6.8.2 Construction & Its Security

We now describe a concrete construction that meets our stated goals. We assume a trusted offline authority called *Auth* (as *SL* in Section 6.4) that facilitates trust among the protocol participants. The offline phase is for *Auth* to authorize Alice's sparse integer set $\mathcal{X}$, given in Figure 6.15, and the online phase where Alice and Bob interact is for Bob to get integers in $\mathcal{X} \cap Q$, given in Figure 6.16.

Similar to Section 6.4, *Auth* first chooses two additional integers outside of $\mathcal{D}$ to indicate the boundaries and random salts for commitments. It then commits each element in $\mathcal{X}$, makes adjacent commitments in a tuple, and signs those tuples. Since only one commitment is required for each element, the overall size of salts being sent is halved. In the online phase, Alice chooses the elements in Q and computes two commitments for the sentinel values outside of Q and the two range proofs for them. Bob's computation cost is also reduced since the number of commitments is halved. NIZKRP steps for checking $x_k < a$ and $x_{k+j+1} > b$ also show that any integers (other than the returned ones) are out of range, as integers are sorted before they are committed. Details are shown in Figure 6.14.

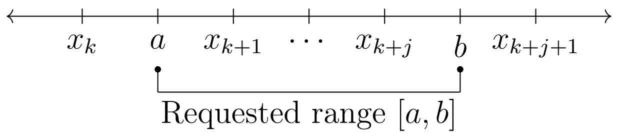


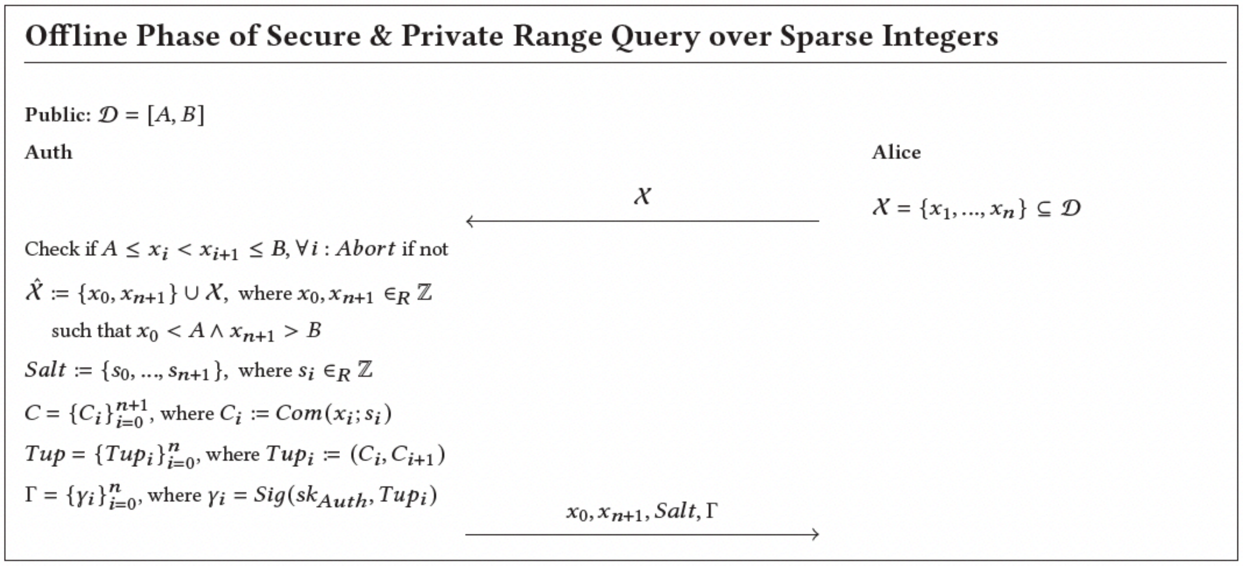
Figure 6.14: Proving $x_k < a$ states x_w, for $\forall w \leq k$ is out of range.
Similarly, proving $x_{k+j+1} > b$ states x_w, for $\forall w \geq (k + j + 1)$ is out of range.

We now give a proof sketch of security and privacy for the proposed construction:

1. Goal 1 is satisfied by the binding property of the commitment scheme, along with signatures on the tuples containing two commitments of neighboring integers. Signatures on commitments ensure that no element of $\mathcal{X}$ can be changed.
2. Signatures on commitment tuples ensure that any inclusion or omission of elements is detectable.
3. Consider tuples outside $Q = [a, b]$, i.e., $\{Com(x_k), Com(x_{k+1})\}$ and $\{Com(x_{k+j}), Com(x_{k+j+1})\}$ in Figure 6.14. Due to the hiding property of the commitment scheme, Bob learns no information about either x_k or x_{k+j+1} from $Com(x_k)$ and $Com(x_{k+j+1})$, respectively. Also, NIZKRP guarantees that no information about x_k and x_{k+j+1} is revealed other than the fact $x_k < a$ and $x_{k+j+1} > b$, respectively.

6.9 Related Work

Privacy of genetic material and tests has attracted much attention, and numerous methods have been proposed based on various cryptographic techniques. Genomic security, on the other hand, remained in the background due to a (mistakenly) perceived lack of challenges. In this section, we briefly go over the cryptographic privacy building blocks in the genomics domain and then discuss related techniques for achieving data authenticity and integrity in the context of range queries.



Offline Phase of Secure & Private Range Query over Sparse Integers

Public: $\mathcal{D} = [A, B]$

Auth

Alice

$\mathcal{X} = \{x_1, ..., x_n\} \subseteq \mathcal{D}$

$\xleftarrow{\mathcal{X}}$

Check if $A \le x_i < x_{i+1} \le B, \forall i : Abort$ if not

$\hat{\mathcal{X}} := \{x_0, x_{n+1}\} \cup \mathcal{X}$, where $x_0, x_{n+1} \in_R \mathbb{Z}$ such that $x_0 < A \wedge x_{n+1} > B$

$Salt := \{s_0, ..., s_{n+1}\}$, where $s_i \in_R \mathbb{Z}$

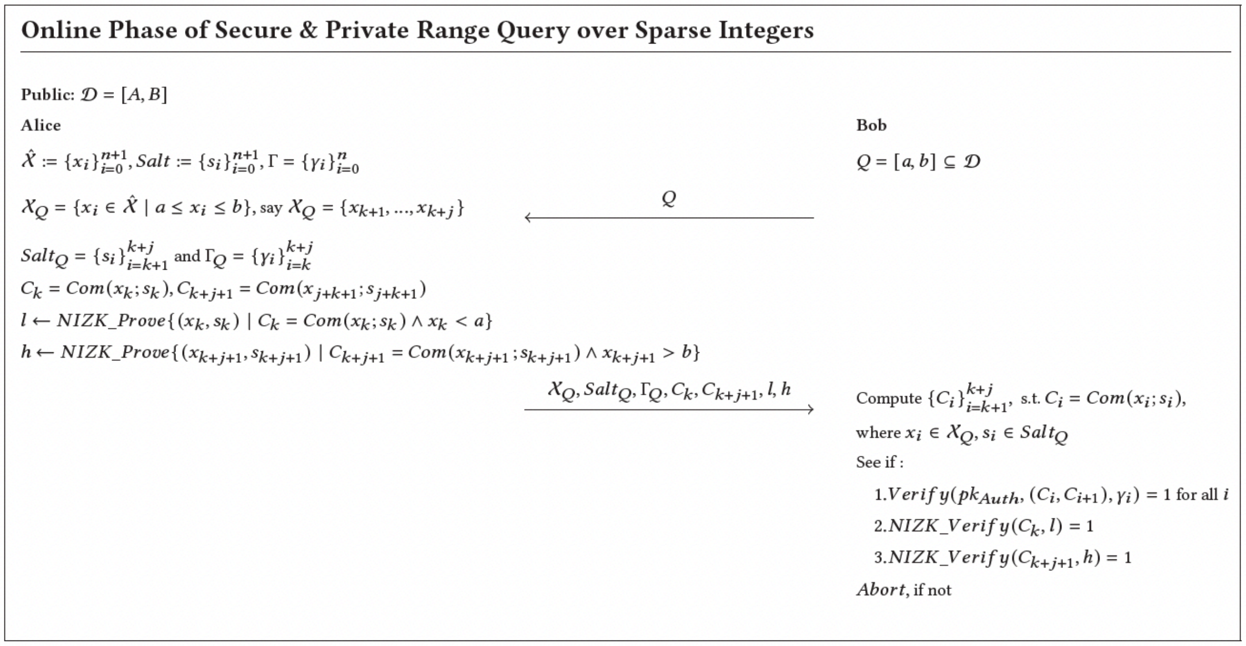
$C = \{C_i\}_{i=0}^{n+1}$, where $C_i := Com(x_i; s_i)$

$Tup = \{Tup_i\}_{i=0}^{n}$, where $Tup_i := (C_i, C_{i+1})$

$\Gamma = \{\gamma_i\}_{i=0}^{n}$, where $\gamma_i = Sig(sk_{Auth}, Tup_i)$

$\xrightarrow{x_0, x_{n+1}, Salt, \Gamma}$

Figure 6.15: (Offline Phase) Authorizing Alice's sparse integer set $\mathcal{X}$ from $Auth$



Online Phase of Secure & Private Range Query over Sparse Integers

Public: $\mathcal{D} = [A, B]$

Alice

Bob

$\hat{\mathcal{X}} := \{x_i\}_{i=0}^{n+1}, Salt := \{s_i\}_{i=0}^{n+1}, \Gamma = \{\gamma_i\}_{i=0}^{n}$

$Q = [a, b] \subseteq \mathcal{D}$

$\mathcal{X}_Q = \{x_i \in \hat{\mathcal{X}} \mid a \le x_i \le b\}$, say $\mathcal{X}_Q = \{x_{k+1}, ..., x_{k+j}\}$

$\xleftarrow{Q}$

$Salt_Q = \{s_i\}_{i=k+1}^{k+j}$ and $\Gamma_Q = \{\gamma_i\}_{i=k}^{k+j}$

$C_k = Com(x_k; s_k), C_{k+j+1} = Com(x_{j+k+1}; s_{j+k+1})$

$l \leftarrow NIZK\_Prove\{(x_k, s_k) \mid C_k = Com(x_k; s_k) \wedge x_k < a\}$

$h \leftarrow NIZK\_Prove\{(x_{k+j+1}, s_{k+j+1}) \mid C_{k+j+1} = Com(x_{k+j+1}; s_{k+j+1}) \wedge x_{k+j+1} > b\}$

$\xrightarrow{\mathcal{X}_Q, Salt_Q, \Gamma_Q, C_k, C_{k+j+1}, l, h}$

Compute $\{C_i\}_{i=k+1}^{k+j}$, s.t. $C_i = Com(x_i; s_i)$, where $x_i \in \mathcal{X}_Q, s_i \in Salt_Q$

See if :

1.$Verify(pk_{Auth}, (C_i, C_{i+1}), \gamma_i) = 1$ for all i

2.$NIZK\_Verify(C_k, l) = 1$

3.$NIZK\_Verify(C_{k+j+1}, h) = 1$

$Abort$, if not

Figure 6.16: (Online Phase) Range query-response between Alice and Bob

6.9.1 Genomic Privacy

Several recent survey papers, [230, 28, 63, 293] overview recent advances in genomic privacy. This section lists the techniques used in this domain and their related work. First, MPC is commonly used for genomic privacy. For example, [185] utilizes oblivious transfer (OT) and oblivious circuit evaluation to compute the edit distance and Smith-Waterman similarity score to measure the similarity between two DNA sequences. [47] uses PSI Cardinality for paternity tests by observing the sizes of DNA fragments cut by restricting enzymes. It also shows how to use Authorized PSI for personalized medicine, where an authority authorizes the markers to be checked in the DNA. [102] uses an Android smartphone to store encrypted genomic data and PSI techniques to provide results of personal medicine, paternity, and ancestry tests using the smartphone as the end-user computation device. [306] utilizes secret-sharing-based MPC techniques to compute minor allele frequencies (MAF) and chi-squared statistics in the context of Genome-Wide Association Study computation and the Hamming distance between two genomic datasets. To improve the efficiency of edit distance computation, [295] approximates the edit distance by compressing genome sequences to sets and privately computing the set difference size (or the threshold of the set difference size) using garbled circuit and OT techniques.

HE is another popular tool for privacy in genomic tests. For example, [40] uses PHE (see Section 6.2.4) and dynamic programming techniques to compute the edit distance of two DNA sequences without revealing their sequences to each other, which is later improved in [41]. Subsequently, [87] suggests a way to compute the edit distance on encrypted genomic data using SWHE (see Section 6.2.4). [188] constructs a secure genomic data query architecture with third parties, where one party stores and computes encrypted genomic data – e.g., storage and processing unit (SPU) – while the other party manages the keys used to encrypt and decrypt genomic data. It uses the additively homomorphic property of the Paillier [242] cryptosystem to secure count query. Similarly, [44] suggests an architecture for disease susceptibility tests with a third-party SPU. This technique utilizes Paillier Partial HE and proxy re-encryption [60] so that only the two parties who receive the partial secret keys from the patient can participate in the test. i.e., SPU homomorphically computes the test on the encrypted DNA and partially decrypts the result, such that the medical center can obtain the final result by partial decryption of the message received from SPU. These ideas are further refined in [45] and [101] which showed that [45] can be more efficiently implemented by using additively homomorphic Elliptic Curve-based ElGamal and simpler encoding method of genomic data. [246] presents a method to search encrypted biomedical data stored in the server using Bloom filters and HE, so that the user can perform the search query to the server. [205] uses a ring-based FHE scheme [218] to encrypt genomic data and demonstrates common genomic computations over encrypted data, e.g., Pearson Goodness-of-Fit test, the D' and r^2 measures of linkage disequilibrium, the Estimation Maximization (EM) algorithm for haplotyping, and the Cochran-Armitage Test for Trend. [307] and [294] also assume encrypted genomic data with an FHE [161] on untrusted public cloud. [307] focuses on the chi-square statistics and proposes two protocols for secure division operation, while [294] suggests a framework for estimating the P-value of exact logistic regression parameters over encrypted data. [163] shows how to construct an index tree of the genomic data and send the index tree to the cloud server after encryption with Paillier; the server then traverses the encrypted tree according to the encrypted query from a query initiator using a secure function evaluation via an interactive protocol using Yao's garbled circuits [302].

6.9.2 Range Query Security

Range query completeness was explored in the context of outsourced databases where the data owner assigns query handling to a cloud-based data publisher. The latter, if malicious, can reply to a range query with incomplete and/or fake results. To counter such misbehavior, [172] focused on minimizing privacy leakages on data attributes using data partitioning algorithms that are aware of the distribution of query ranges. [243] developed methods based on the continual linking of elements and collision-resistant hash functions to prevent malicious actions. [207] used Merkle hash and B^+ trees, as well as aggregated signatures to provide authenticity and integrity (with less strict privacy requirements than [243]) and improve efficiency in the dynamic database case.

Other cryptographic range query techniques incorporated so-called range proofs. Early examples of range proofs include [221, 68, 81]. [221] uses the bit-length of the committed value to prove that the number is in the range $[0, 2^k - 1]$ where k is the number of bits in that committed value. [68] only proves that the committed value lies in a broader range: $[-a, 2a]$, instead of $[0, a]$. [81] convinces a verifier that the committed value lies in a range with the expansion rate of 2^{t+l+1}, where t and l are security parameters. [64] proposes two efficient protocols for proving that the committed value which is in $[a, b]$ lies in $[a - \theta, b + \theta]$ where $\theta = 2^{t+l+1}\sqrt{b-a}$ and t and l are security parameters. The second protocol (that uses Fujisaki-Okamoto commitments [147]) is the one we used in this work; it has the expansion rate of 1. [72] proposes range proofs based on set membership protocols by extending the u-ary notation and proving that the secret $z \in [0, u^l - 1]$.

6.10 Limitations and Future Work

In this work, we focused on genomic tests that reveal the genomic data in some queried range. We also showed that our proposed technique can be used for private substring matching-type tests where genomic data can be encrypted under an additively homomorphic encryption scheme. However, different types of genomic tests may require more operations other than addition, subtraction, and constant multiplication that additively homomorphic encryption cannot support. We cannot avoid using some SWHE or FHE in such cases to keep the privacy, sacrificing the performance, as some previous work [205, 87, 307, 294] suggested.

We avoided going into discussions regarding low-level side-channels in our work. For instance, Alice can infer the size of the queried range from execution time when applying our technique to SPH-PSM as in Section 6.5. However, this can be easily prevented in practice by letting T send the replies after a fixed period instead of sending them right after the computation is finished.

Lastly, we only consider SNP for efficient genomic representation; however, there are multiple genomic representation formats (as mentioned in Section 6.2.1). Also, for the base letters, we only consider well-sequenced genomic data without any gene duplications, insertions, deletions, or lateral gene transfers that can commonly occur in genomic materials. Therefore, there is still room for improving our proposed techniques, and we consider these to be opportunities for future work.

6.11 Summary

This work motivated and constructed a secure and private genomic range query technique, balancing genomic security and privacy. To achieve authenticity of genomic material, completeness of mutations within a given range, and total privacy for genomic data outside of the range, we applied zero-knowledge range proofs to show that a committed value (the position of a genomic mutation) is outside the queried range, digital signatures to prevent any alterations on genomic material and linkage among two consecutive mutations to preclude any omissions. We also abstracted away from genomics and defined a more general problem of secure and private range queries over sparse integers. In addition, we showed the applicability of our approach with the private substring matching problem as an example.

Chapter 7

Conclusion and Future Work

This dissertation addressed several research challenges to advance closer toward end-to-end data privacy from generation to consumption. This section summarizes each chapter and suggests potential future research directions for each work.

Chapter 2 presented $\mathcal{P}$ARseL, a provable attestation root-of-trust for mid-range devices over a formally verified microkernel, seL4. It reduces the size of TCB at runtime and separates it from all the user-dependent components, which makes it feasible to verify its implementation. We formally verified the runtime components' memory safety, functional correctness, and secret independence. Our verified implementation (including boot-time components) is open-sourced, and our evaluation showed its feasibility for the existing hardware platform.

$\mathcal{P}$ARseL provides *static* root-of-trust for assessing user-space processes, i.e., the binary of processes are measured at their loading time. This is credible because $\mathcal{P}$ARseL enforces no new processes spawned during runtime, and no code modifications are allowed without re-booting the device. On the other hand, an adversary compromising the user-space processes can still perform control-flow hijacking attacks. Therefore, extending $\mathcal{P}$ARseL to support control-flow integrity would be an interesting research direction.

In Chapter 3, we formalized *PfB* for data privacy from its generation and presented VERSA, a secure architecture for low-end devices realizing *PfB*. VERSA guarantees that only correct execution of expected and explicitly authorized software can access and manipulate the sensing interfaces – which can be generalized to access control to any memory regions. Our VERSA implementation is formally verified and open-sourced, and its evaluation results showed its affordability in low-end devices.

Following the related work, VERSA also triggers immediate MCU *reset* whenever any violations occur, including interrupts and DMA. However, it may not be ideal for real-time applications that require interrupts as resetting/rebooting the device causes delay. Thus, other than our suggestion to remedy this issue in Section 3.9, one can design a secure interrupt-friendly architecture for low-end devices.

Chapter 4 proposed how to enforce the input correctness in PSI with input size conditions. We first identified possible malicious behaviors using duplicates and/or bogus elements. To prevent duplicates, we defined PoED and presented a construction satisfying PoED using homomorphic encryption and shuffling. We extended this to AD-PSI which outputs the intersection only when the PoED is successfully verified and discussed its variants. Then, applying the AD-PSI to the U-SH-PSI, we completed the B-SH-PSI which outputs the result only when both lower and upper bounds for input set size are satisfied. Finally, we showed a protocol for AD-APSI to avoid both duplicates and bogus elements involving a TTP.

On the other hand, some MPC protocols reveal more information than intended when they are repeatedly executed. For example, although a secure PSI-CA protocol reveals only the intersection set's cardinality, manipulating input sets carefully can reveal exact intersecting items. Such information leakage from repetitive executions of a secure MPC protocol would need to be researched as it can be critical in real-life applications. In addition, our input correctness check is specific to the PSI with lower bounds as it checks the distinctness of input elements. Therefore, more general methods would be needed to check the correctness of MPC with input conditions, such as specific formatting.

Chapter 5 presented PMPC schemes for dynamic groups and dynamic GAS settings, considering mobile adversaries. We proactivized two MPC schemes based on linear secret sharing by adding `Refresh` and `Recover` protocols, where the former refreshes the distributed sharings, and the latter recovers partial shares with the others' help. We also built `Redistribute` protocols for dynamic group settings, which redistribute the shares from previous to new groups participating in the computation. Also, we presented the share conversions between these two PMPC schemes for dynamic GASs.

Although our work considered strong adversaries who can corrupt parties for a certain period and eventually corrupt everyone, *universal composability* (UC) [75] was not considered. Since MPC systems are usually considered building blocks for larger security systems, building PMPC schemes with UC security would be meaningful in real-life applications.

In Chapter 6, our work showed the challenges of balancing the triad of security, privacy, and efficiency in genomic domains. We focused on an efficient genomic representation, SNP, which induces completeness problems while effectively improving efficiency. To resolve this issue, we applied linking over cryptographic commitments and ZKRP to balance security and privacy, guaranteeing authenticity and integrity while revealing the minimum amount of data necessary for testing. We also applied these techniques to SPH-PSM-based genomic tests and showed the feasibility of adding security.

Although SNP representations are common in genomic tests, protocols may vary for different genomic representations, such as Indels, CNVs, and STRs. In addition, not all genomic tests are based on range queries. Therefore, to further apply security and privacy in genomic data, close collaboration with relevant disciplines is essential to identify commonly used genomic tests and apply cryptographic techniques carefully.

Bibliography

[1] 23andme. https://www.23andme.com/. Accessed: 2022-01-24.

[2] Cri genetics. https://www.crigenetics.com/. Accessed: 2022-01-24.

[3] Go. https://golang.org/. Accessed: 2022-01-24.

[4] How do geneticists indicate the location of a gene? https://ghr.nlm.nih.gov/primer/howgeneswork/genelocation. Accessed: 2022-01-24.

[5] The karamel compiler (2017). https://github.com/FStarLang/karamel.

[6] The legion of the bouncy castle. https://www.bouncycastle.org/. Accessed: 2022-01-24.

[7] Motion sensor code. https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/pir_motion_sensor.

[8] paillier. https://github.com/didiercrunch/paillier. Accessed: 2022-01-24.

[9] Project everest bibiliography. https://project-everest.github.io/papers/.

[10] Researchers discover lg smart tv vulnerabilities allowing root access. https://thehackernews.com/2024/04/researchers-discover-lg-smart-tv.html. Accessed: 2024-05-04.

[11] Securerandom (java platform se 8). https://docs.oracle.com/javase/8/docs/api/java/security/SecureRandom.html. Accessed: 2022-01-24.

[12] Smart light bulbs can hack your personal information. https://gulfnews.com/technology/smart-light-bulbs-can-hack-your-personal-information-1.1571846229201. Accessed: 2024-05-04.

[13] Snp. https://www.nature.com/scitable/definition/snp-295/#:~:text=If%20more%20than%201%25%20of,having%20more%20than%20one%20allele. Accessed: 2022-01-24.

[14] Temperature sensor code. https://github.com/Seeed-Studio/LaunchPad_Kit/tree/master/Grove_Modules/temp_humi_sensor.

[15] Whole genome association studies. https://www.genome.gov/17516714/2006-release-about-whole-genome-association-studies. Accessed: 2022-01-24.

[16] Zero-knowledge proofs. https://github.com/ing-bank/zkproofs. Accessed: 2022-01-24.

[17] Mapping and sequencing the human genome. National Research Council (US) Committee on Mapping and Sequencing the Human Genome. Washington (DC): National Academies Press (US); 1988. 2, Introduction., 1988. Available from: https://www.ncbi.nlm.nih.gov/books/NBK218247/, Accessed: 2022-01-31.

[18] Avr atmega 1284p 8-bit microcontroller. http://ww1.microchip.com/downloads/en/DeviceDoc/doc8059.pdf, 2009.

[19] pages 295–300. John Wiley & Sons, Ltd, 2010.

[20] Secure hash standard. FIPS PUB 180-4, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD, 2012.

[21] Darpa high assurance cyber military systems (hacms) heavy equipment transporter. https://www.youtube.com/watch?v=6cllzGGxRfE, 2017.

[22] Palisade homomorphic encryption software library. Online: https://palisade-crypto.org/, 2017.

[23] Msp430 flash memory characteristics. https://www.ti.com/lit/an/slaa334b/slaa334b.pdf?ts=1638460551489, 2018.

[24] Lattigo v4. Online: https://github.com/tuneinsight/lattigo, Aug 2022. EPFL-LDS, Tune Insight SA.

[25] VERSA source code. https://github.com/sprout-uci/pfb, 2022.

[26] T. Abera, N. Asokan, L. Davi, J. Ekberg, T. Nyman, A. Paverd, A. Sadeghi, and G. Tsudik. C-FLAT: control-flow attestation for embedded systems software. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 743–754. ACM, 2016.

[27] T. Abera, N. Asokan, L. Davi, F. Koushanfar, A. Paverd, A.-R. Sadeghi, and G. Tsudik. Invited: Things, trouble, trust: on building trust in IoT systems. In *ACM/IEEE Design Automation Conference (DAC)*, page 121, Austin, TX, USA, 2016. ACM.

[28] B. Abinaya and S. Santhi. A survey on genomic data by privacy-preserving techniques perspective. *Computational Biology and Chemistry*, 93:107538, 2021.

[29] E. Aliaj, I. D. O. Nunes, and G. Tsudik. GAROTA: generalized active root-of-trust architecture. *CoRR*, abs/2102.07014, 2021.

[30] J. F. Almansa, I. Damgård, and J. B. Nielsen. Simplified threshold rsa with adaptive and proactive security. In *Proceedings of the 24th annual international conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, pages 593–611, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] M. Ammar and B. Crispo. Verify&revive: Secure detection and recovery of compromised low-end embedded devices. In *Annual Computer Security Applications Conference*, pages 717–732, 2020.

[32] M. Ammar, B. Crispo, and G. Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020.

[33] Anonymous. Parsel open-source code. `https://anonymous.4open.science/r/parsel-submission-1EC5/`.

[34] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster. Keeping the smart home private with smart (er) iot traffic shaping. *arXiv preprint arXiv:1812.00955*, 2018.

[35] N. Apthorpe, D. Reisman, and N. Feamster. Closing the blinds: Four strategies for protecting smart home privacy from network observers. *arXiv preprint arXiv:1705.06809*, 2017.

[36] N. J. Apthorpe, D. Reisman, and N. Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. *CoRR*, abs/1705.06805, 2017.

[37] E. Aras, M. Ammar, F. Yang, W. Joosen, and D. Hughes. Microvault: Reliable storage unit for iot devices. In *2020 16th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 132–140, 2020.

[38] Arm Ltd. Arm TrustZone. `https://www.arm.com/products/security-on-arm/trustzone`, 2018.

[39] N. Asokan, T. Nyman, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11), 2018.

[40] M. J. Atallah, F. Kerschbaum, and W. Du. Secure and private sequence comparisons. In *Proceedings of the 2003 ACM Workshop on Privacy in the Electronic Society*, WPES '03, pages 39–44, New York, NY, USA, 2003. ACM, Association for Computing Machinery.

[41] M. J. Atallah and J. Li. Secure outsourcing of sequence comparisons. *International Journal of Information Security*, 4(4):277–287, Oct 2005.

[42] G. Ateniese, E. De Cristofaro, and G. Tsudik. (if) size matters: size-hiding private set intersection. In *International Workshop on Public Key Cryptography*, pages 156–173. Springer Berlin Heidelberg, 2011.

[43] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich. Notary: A device for secure transaction approval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 97–113, New York, NY, USA, 2019. Association for Computing Machinery.

[44] E. Ayday, J. L. Raisaro, and J.-P. Hubaux. Privacy-enhancing technologies for medical tests using genomic data. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[45] E. Ayday, J. L. Raisaro, J.-P. Hubaux, and J. Rougemont. Protecting and evaluating genomic privacy in medical tests and personalized medicine. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 95–106, New York, NY, USA, 2013. ACM, Association for Computing Machinery.

[46] M. Backes, C. Cachin, and R. Strobl. Proactive secure message transmission in asynchronous networks. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*, pages 223–232, 2003.

[47] P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering gattaca: Efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 691–702, New York, NY, USA, 2011. ACM, Association for Computing Machinery.

[48] J. Baron, K. Eldefrawy, J. Lampkins, and R. Ostrovsky. How to withstand mobile virus attacks, revisited. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 293–302, New York, NY, USA, 2014. ACM.

[49] J. Baron, K. Eldefrawy, J. Lampkins, and R. Ostrovsky. Communication-optimal proactive secret sharing for dynamic groups. In *Proceedings of the 2015 International Conference on Applied Cryptography and Network Security*, ACNS '15, 2015.

[50] D. Beaver. Efficient multiparty protocols using circuit randomization. In J. Feigenbaum, editor, *Advances in Cryptology — CRYPTO '91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[51] Z. Beerliová-Trubíniová and M. Hirt. Efficient multi-party computation with dispute control. In *TCC*, pages 305–328, 2006.

[52] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure mpc with linear communication complexity. In *TCC*, pages 213–230, 2008.

[53] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 1–15, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[54] M. Bellare and O. Goldreich. On defining proofs of knowledge. In *Advances in Cryptology — CRYPTO' 92*, pages 390–420, 1993.

[55] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10, 1988.

[56] E. Ben-Sasson, S. Fehr, and R. Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *CRYPTO*, pages 663–680, 2012.

[57] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. In *Journal of Cryptographic Engineering*, 2011.

[58] G. R. Blakley. Safeguarding cryptographic keys. *Proc. of AFIPS National Computer Conference*, 48:313–317, 1979.

[59] M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.

[60] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT'98*, pages 127–144, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[61] S. Blazy and X. Leroy. Mechanized Semantics for the Clight Subset of the C Language. *Journal of Automated Reasoning*, 2009.

[62] H. Blumberg. Exceptional sets. In *Fundamenta Mathematicae*, pages 3–32, 1939.

[63] L. Bonomi, Y. Huang, and L. Ohno-Machado. Privacy challenges and research opportunities for genomic data sharing. *Nature Genetics*, 52(7):646–654, July 2020.

[64] F. Boudot. Efficient proofs that a committed number lies in an interval. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 431–444. Springer, 2000.

[65] T. Bradley, X. Ding, and G. Tsudik. Genomic security (lest we forget). *IEEE Security & Privacy*, 15(5):38–46, 2017.

[66] T. Bradley, S. Faber, and G. Tsudik. Bounded size-hiding private set intersection. In *International Conference on Security and Cryptography for Networks*, pages 449–467. Springer, Springer International Publishing, 2016.

[67] F. Brasser, B. E. Mahjoub, A. Sadeghi, C. Wachsmann, and P. Koeberl. Tytan: tiny trust anchor for tiny devices. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 34:1–34:6. ACM, 2015.

[68] E. F. Brickell, D. Chaum, I. B. Damgård, and J. van de Graaf. Gradual and verifiable release of a secret. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 156–166. Springer, 1987.

[69] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[70] M. Busi, J. Noorman, J. Van Bulck, L. Galletta, P. Degano, J. T. Mühlberg, and F. Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 262–276. IEEE, 2020.

[71] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *ACM Conference on Computer and Communications Security*, pages 88–97, 2002.

[72] J. Camenisch, R. Chaabouni, and A. Shelat. Efficient protocols for set membership and range proofs. In J. Pieprzyk, editor, *Advances in Cryptology - ASIACRYPT 2008*, pages 234–252, Berlin, Heidelberg, 2008. Springer, Springer Berlin Heidelberg.

[73] J. Camenisch and M. Stadler. Efficient group signature schemes for large groups. In *Advances in Cryptology — CRYPTO '97*, pages 410–424, 1997.

[74] S. Canard, I. Coisel, A. Jambert, and J. Traoré. New results for the practical use of range proofs. In S. Katsikas and I. Agudo, editors, *Public Key Infrastructures, Services and Applications*, pages 47–64, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[75] R. Canetti. Universally composable security: a new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145, 2001.

[76] R. Canetti and A. Herzberg. Maintaining security in the presence of transient faults. In *CRYPTO*, pages 425–438, 1994.

[77] X. Carpent, S. Faber, T. Sander, and G. Tsudik. Private set projections & variants. In *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society (WPES '17)*. Association for Computing Machinery, 2017.

[78] X. Carpent, S. Hwang, and G. Tsudik. Element distinctness and bounded input size in private set intersection and related protocols. In C. Pöpper and L. Batina, editors, *Applied Cryptography and Network Security*, pages 26–57, Cham, 2024. Springer Nature Switzerland.

[79] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[80] A. Cerulli, E. De Cristofaro, and C. Soriente. Nothing refreshes like a repsi: Reactive private set intersection. In *Applied Cryptography and Network Security*, pages 280–300, 2018.

[81] A. H. Chan, Y. Frankel, and Y. Tsiounis. Easy come - easy go divisible cash. In *Advances in Cryptology - EUROCRYPT '98, International Conference on the Theory and Application of Cryptographic Techniques, Espoo, Finland, May 31 - June 4, 1998, Proceeding*, volume 1403 of *Lecture Notes in Computer Science*, pages 561–575. Springer, 1998.

[82] M. Chase and P. Miao. Private set intersection in the internet setting from lightweight oblivious prf. In *Advances in Cryptology – CRYPTO 2020*, pages 34–63. Springer International Publishing, 2020.

[83] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM.

[84] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Advances in Cryptology – CRYPTO' 88*, pages 319–327. Springer New York, 1990.

[85] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, 2018.

[86] Y. Cheng, X. Ji, X. Zhou, and W. Xu. Homespy: Inferring user presence via encrypted traffic of home surveillance camera. In *ICPADS*, pages 779–782, 2017.

[87] J. H. Cheon, M. Kim, and K. Lauter. Homomorphic computation of edit distance. In M. Brenner, N. Christin, B. Johnson, and K. Rohloff, editors, *Financial Cryptography and Data Security*, pages 194–212, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[88] M. Ciampi and C. Orlandi. Combining private set-intersection with secure two-party computation. In *Security and Cryptography for Networks*, 2018.

[89] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, 2002.

[90] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking.* MIT press, 2018.

[91] J. B. Clarkson. Dense probabilistic encryption. In *In Proceedings of the Workshop on Selected Areas of Cryptography*, pages 120–128, 1994.

[92] D. Cofer, A. Gacek, J. Backes, M. W. Whalen, L. Pike, A. Foltzer, M. Podhradsky, G. Klein, I. Kuz, J. Andronick, G. Heiser, and D. Stuart. A formal approach to constructing secure air vehicle software. *Computer*, 2018.

[93] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[94] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, 2014.

[95] R. Cramer, I. Damgård, and U. Maurer. Span programs and general secure multi-party computation. *BRICS Report Series*, 4(28), Jan. 1997.

[96] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT '97*, pages 103–118, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[97] D. Dachman-Soled, T. Malkin, M. Raykova, and M. Yung. Efficient robust private set intersection. In *Applied Cryptography and Network Security*, pages 125–142, 2009.

[98] I. Damgård, Y. Ishai, and M. Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *EUROCRYPT*, pages 445–465, 2010.

[99] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, pages 241–261, 2008.

[100] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In K. Kim, editor, *Public Key Cryptography*, pages 119–136, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[101] G. Danezis and E. De Cristofaro. Fast and private genomic testing for disease susceptibility. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 31–34. ACM, 2014.

[102] E. De Cristofaro, S. Faber, P. Gasti, and G. Tsudik. Genodroid: are privacy-preserving genomic tests ready for prime time? In *Proceedings of the 2012 ACM workshop on Privacy in the electronic society*, pages 97–108. ACM, 2012.

[103] E. De Cristofaro, S. Faber, and G. Tsudik. Secure genomic testing with size-and position-hiding private substring matching. In *Proceedings of the 12th ACM workshop on Workshop on privacy in the electronic society*, pages 107–118. ACM, 2013.

[104] E. De Cristofaro, P. Gasti, and G. Tsudik. Fast and private computation of cardinality of set intersection and union. In *Cryptology and Network Security*, pages 218–231. Springer, 2012.

[105] E. De Cristofaro, J. Kim, and G. Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *Advances in Cryptology - ASIACRYPT 2010*, pages 213–231, 2010.

[106] E. De Cristofaro and G. Tsudik. Practical private set intersection protocols with linear complexity. In *Financial Cryptography and Data Security*, pages 143–159, 2010.

[107] E. De Cristofaro and G. Tsudik. Experimenting with fast private set intersection. In *Trust and Trustworthy Computing*, pages 55–73, 2012.

[108] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, M. Steiner, and G. Tsudik. VRASED: A verified hardware/software co-design for remote attestation. In *USENIX Security*, 2019.

[109] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. Pure: Using verified remote attestation to obtain proofs of update, reset and erasure in low-end embedded systems. 2019.

[110] I. De Oliveira Nunes, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. APEX: A verified architecture for proofs of execution on remote devices under full software compromise. In *29th USENIX Security Symposium (USENIX Security 20)*, Boston, MA, Aug. 2020. USENIX Association.

[111] I. De Oliveira Nunes, S. Hwang, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. $\mathcal{P}$ARseL: Towards a verified root-of-trust over sel4. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–9, 2023.

[112] I. De Oliveira Nunes, S. Hwang, S. Jakkamsetti, and G. Tsudik. Privacy-from-birth: Protecting sensed data from malicious sensors with versa. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2413–2429, 2022.

[113] I. De Oliveira Nunes, S. Jakkamsetti, Y. Kim, and G. Tsudik. Casu: Compromise avoidance via secure update for low-end embedded systems. ICCAD '22.

[114] I. De Oliveira Nunes, S. Jakkamsetti, N. Rattanavipanon, and G. Tsudik. On the toctou problem in remote attestation. *CCS*, 2021.

[115] I. De Oliveira Nunes, S. Jakkamsetti, and G. Tsudik. Dialed: Data integrity attestation for low-end embedded devices. 2021.

[116] I. De Oliveria Nunes, S. Jakkamsetti, and G. Tsudik. Tiny-CFA: Minimalistic control-flow attestation using verified proofs of execution. In *Design, Automation and Test in Europe Conference (DATE)*, 2021.

[117] S. K. Debnath and R. Dutta. Secure and efficient private set intersection cardinality using bloom filter. In *Information Security*, pages 209–226. Springer International Publishing, 2015.

[118] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu. PIR-PSI: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, pages 159–178, 2018.

[119] C. Deng, J. Fan, Z. Wang, Y. Luo, Y. Zheng, Y. Li, and J. Ding. A survey on range proof and its applications on blockchain. In *2019 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 1–8, 2019.

[120] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi. Litehax: lightweight hardware-assisted attestation of program execution. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, ICCAD '18, pages 1–8. IEEE, 2018.

[121] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. Lo-fat: Low-overhead control flow attestation in hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, page 24. ACM, 2017.

[122] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet 1. *Journal of Computer Security*, 11(3):291–314, 2003.

[123] B. Devices. Boundary devices bd-sl-i.mx6. `https://boundarydevices.com/product/bd-sl-i-mx6/`.

[124] X. Ding, E. Ozturk, and G. Tsudik. Balancing security and privacy in genomic range queries. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, WPES'19, page 106–110, New York, NY, USA, 2019. Association for Computing Machinery.

[125] S. Dolev, K. Eldefrawy, J. Lampkins, R. Ostrovsky, and M. Yung. Proactive secret sharing with a dishonest majority. In *Proceedings of the 10th International Conference on Security and Cryptography for Networks - Volume 9841*, pages 529–548, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[126] S. Dolev, J. Garay, N. Gilboa, and V. Kolesnikov. Swarming secrets. In *Communication, Control, and Computing, 2009. Allerton 2009. 47th Annual Allerton Conference on*, pages 1438–1445, Sept 2009.

[127] S. Dolev, J. A. Garay, N. Gilboa, and V. Kolesnikov. Secret sharing krohn-rhodes: Private and perennial distributed computation. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 32–44, 2011.

[128] S. Dolev, J. A. Garay, N. Gilboa, V. Kolesnikov, and Y. Yuditsky. Towards efficient private distributed computation on unbounded input streams. *J. Mathematical Cryptology*, 9(2):79–94, 2015.

[129] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. In *CCS*, 2013.

[130] T. Duong, D. H. Phan, and N. Trieu. Catalic: Delegated psi cardinality with applications to contact tracing. In *Advances in Cryptology – ASIACRYPT 2020*, pages 870–899. Springer International Publishing, 2020.

[131] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0—a framework for ltl and ω-automata manipulation. In *International Symposium on Automated Technology for Verification and Analysis*, 2016.

[132] J. Edelson. A timeline of facebook's privacy issues — and its responses. https://www.nbcnews.com/tech/social-media/timeline-facebook-s-privacy-issues-its-responses-n859651, 2018.

[133] K. Eldefrawy, S. Hwang, R. Ostrovsky, and M. Yung. Communication-efficient (proactive) secure computation for dynamic general adversary structures and dynamic groups. In C. Galdi and V. Kolesnikov, editors, *Security and Cryptography for Networks*, pages 108–129, Cham, 2020. Springer International Publishing.

[134] K. Eldefrawy, R. Ostrovsky, S. Park, and M. Yung. Proactive secure multiparty computation with a dishonest majority. In *Security and Cryptography for Networks - 11th International Conference, SCN 2018, Amalfi, Italy, September 5-7, 2018, Proceedings*, pages 200–215, 2018.

[135] K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In *Wisec*, 2017.

[136] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[137] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology*, pages 10–18, 1985.

[138] W. Englund, E. Nakashima, and T. Telford. Colonial pipeline 'ransomware' attack shows cyber vulnerabilities of u.s. energy grid. https://www.washingtonpost.com/business/2021/05/10/colonial-pipeline-gas-oil-markets/, May 2021.

[139] D. Evans, V. Kolesnikov, and M. Rosulek. *A Pragmatic Introduction to Secure Multi-Party Computation.* 2018.

[140] Y. Feng, Y. Zhang, C. Ying, D. Wang, and C. Du. Nanopore-based fourth-generation dna sequencing technology. *Genomics, proteomics & bioinformatics*, 13(1):4–16, 2015.

[141] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptology—CRYPTO '86*, page 186–194, Berlin, Heidelberg, 1987. Springer-Verlag.

[142] U. Fiege, A. Fiat, and A. Shamir. Zero knowledge proofs of identity. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 210–217. Association for Computing Machinery, 1987.

[143] Y. Frankel, P. Gemmell, P. D. MacKenzie, and M. Yung. Proactive rsa. In *Proceedings of the 17th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '97, pages 440–454, London, UK, UK, 1997. Springer-Verlag.

[144] M. J. Freedman, C. Hazay, K. Nissim, and B. Pinkas. Efficient set intersection with simulation-based security. *Journal of Cryptology*, 29(1):115–155, 2016.

[145] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography*, pages 303–324. Springer Berlin Heidelberg, 2005.

[146] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT 2004*, pages 1–19, 2004.

[147] E. Fujisaki and T. Okamoto. Statistical zero knowledge protocols to prove modular polynomial relations. In *Annual International Cryptology Conference*, pages 16–30. Springer, Springer Berlin Heidelberg, 1997.

[148] R. W. Gardner, S. Garera, and A. D. Rubin. Detecting code alteration by creating a temporary memory bottleneck. *IEEE TIFS*, 2009.

[149] M. Geden and K. Rasmussen. Hardware-assisted remote runtime attestation for critical embedded systems. In *2019 17th International Conference on Privacy, Security and Trust (PST)*, pages 1–10. IEEE, 2019.

[150] S. Ghosh and T. Nilges. An algebraic approach to maliciously secure private set intersection. In *Advances in Cryptology – EUROCRYPT 2019*, pages 154–185. Springer International Publishing, 2019.

[151] O. Girard. openMSP430, 2009.

[152] V. D. Gligor and S. L. M. Woo. Establishing software root of trust unconditionally. In *NDSS*, 2019.

[153] O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications.* Cambridge University Press, 2004.

[154] 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, pages 218–229, New York, NY, USA, 1987. ACM.

[155] S. Goldwasser and S. Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 365–377, New York, NY, USA, 1982. Association for Computing Machinery.

[156] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, 1985.

[157] C. Gouert, D. Mouris, and N. G. Tsoutsos. Sok: New insights into fully homomorphic encryption libraries via standardized benchmarks. *Proc. Priv. Enhancing Technol.*, 2023(3):154–172, 2023.

[158] M. Grisafi, M. Ammar, M. Roveri, and B. Crispo. PISTIS: Trusted computing architecture for low-end embedded systems. In *31st USENIX Security Symposium*, pages 3843–3860, Aug. 2022.

[159] J. Groth. Non-interactive zero-knowledge arguments for voting. In J. Ioannidis, A. Keromytis, and M. Yung, editors, *Applied Cryptography and Network Security*, pages 467–482, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[160] C. Hagen, C. Weinert, C. Sendner, A. Dmitrienko, and T. Schneider. All the numbers are us: Large-scale abuse of contact discovery in mobile messengers. *IACR Cryptology ePrint Archive*, page 1119, 2020.

[161] S. Halevi and V. Shoup. An implementation of homomorphic encryption. `https://github.com/shaih/HElib(2013)`. Accessed: 2022-01-31.

[162] P. Hallgren, C. Orlandi, and A. Sabelfeld. Privatepool: Privacy-preserving ridesharing. In *IEEE 30th Computer Security Foundations Symposium (CSF)*, 2017.

[163] M. Z. Hasan, M. S. R. Mahdi, M. N. Sadat, and N. Mohammed. Secure count query on encrypted genomic data. *Journal of Biomedical Informatics*, 81:41–52, 2018.

[164] C. Hazay and Y. Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In *Theory of Cryptography*, pages 155–175, 2008.

[165] C. Hazay and K. Nissim. Efficient set operations in the presence of malicious adversaries. In *Public Key Cryptography – PKC 2010*, pages 312–331, 2010.

[166] S. J. Heerema and C. Dekker. Graphene nanodevices for dna sequencing. *Nature nanotechnology*, 11(2):127, 2016.

[167] B. Hemenway Falk, D. Noble, and R. Ostrovsky. Private set intersection with linear communication from general assumptions. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, WPES'19, pages 14–25. Association for Computing Machinery, 2019.

[168] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *CRYPTO*, pages 339–352, 1995.

[169] M. Hirt, U. Maurer, and C. Lucas. A Dynamic Tradeoff between Active and Passive Corruptions in Secure Multi-Party Computation. In R. Canetti and J. A. Garay, editors, *Advances in cryptology - CRYPTO 2013 : 33rd Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013 : proceedings*, volume 8043 of *Lecture notes in computer science*, pages 203–219, Heidelberg, 2013. Springer.

[170] M. Hirt and D. Tschudi. Efficient general-adversary multi-party computation. In K. Sako and P. Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013*, pages 181–200, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[171] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 1969.

[172] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 720–731. VLDB Endowment, 2004.

[173] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*. ISOC, 2012.

[174] B. Huberman, M. Franklin, and T. Hogg. Enhancing privacy and trust in electronic communities. In *ACM Conference on Electronic Commerce*, 1999.

[175] S. Hwang, E. Ozturk, and G. Tsudik. Source code for evaluation. `https://github.com/sprout-uci/genomic-security-journal-code`, 2022.

[176] S. Hwang, E. Ozturk, and G. Tsudik. Balancing security and privacy in genomic range queries. *ACM Trans. Priv. Secur.*, 26(3), mar 2023.

[177] S. IKEDA. South korea issues fines to facebook, netflix over privacy violations. `https://www.cpomagazine.com/data-protection/south-korea-issues-fines-to-facebook-netflix-over-privacy-violations/`.

[178] M. G. Ilaria Chillotti, Nicolas Gama and M. Izabachène. Tfhe: Fast fully homomorphic encryption library over the torus. `https://github.com/tfhe/tfhe.(2017)`. Accessed: 2022-01-31.

[179] T. Instruments. Msp430 ultra-low-power sensing & measurement mcus. `http://www.ti.com/microcontrollers/msp430-ultra-low-power-mcus/overview.html`.

[180] Intel. Intel Software Guard Extensions (Intel SGX). `https://software.intel.com/en-us/sgx`.

[181] M. Ion, B. Kreuter, A. E. Nergiz, S. Patel, S. Saxena, K. Seth, M. Raykova, D. Shanahan, and M. Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 370–389, 2020.

[182] A. Irfan, A. Cimatti, A. Griggio, M. Roveri, and R. Sebastiani. Verilog2SMV: A tool for word-level verification. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, 2016.

[183] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography with constant computational overhead. In *STOC*, pages 433–442, 2008.

[184] S. Jarecki and X. Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *Theory of Cryptography*, pages 577–594, 2009.

[185] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 216–230, 2008.

[186] M. Jin, R. Jia, and C. J. Spanos. Virtual occupancy sensing: Using smart meters to indicate your presence. *IEEE Transactions on Mobile Computing*, 16(11):3264–3277, 2017.

[187] K. Kajita and G. Ohtake. Private set intersection for viewing history with efficient data matching. In *HCI International 2022 Posters*, pages 498–505. Springer International Publishing, 2022.

[188] M. Kantarcioglu, W. Jiang, Y. Liu, and B. Malin. A cryptographic approach to securely share and query genomic sequences. *IEEE Transactions on information technology in biomedicine*, 12(5):606–617, 2008.

[189] A. Kawachi, K. Tanaka, and K. Xagawa. Multi-bit cryptosystems based on lattice problems. In *Public Key Cryptography – PKC 2007*, pages 315–329. International Workshop on Public Key Cryptography, 04 2007.

[190] R. Kennell and L. H. Jamieson. Establishing the genuinity of remote computer systems. In *USENIX Security Symposium*, 2003.

[191] H. C. Kim Laine and R. Player. Simple encrypted arithmetic library (seal). `https://github.com/microsoft/SEAL.(2017)`. Accessed: 2022-01-31.

[192] L. Kissner and D. Song. Privacy-preserving set operations. In *Advances in Cryptology – CRYPTO 2005*, pages 241–257, 2005.

[193] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *SIGOPS*. ACM, 2009.

[194] G. Klein, K. Elphinstone, G. Heiser, et al. seL4: Formal verification of an OS kernel. In *ACM SIGOPS*, 2009.

[195] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. TrustLite: A security architecture for tiny embedded devices. In *EuroSys*, 2014.

[196] V. Kolesnikov, R. Kumaresan, M. Rosulek, and N. Trieu. Efficient batched oblivious prf with applications to private set intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS, pages 818–829. Association for Computing Machinery, 2016.

[197] V. Kolesnikov, M. Rosulek, and N. Trieu. SWiM: Secure wildcard pattern matching from ot extension. In *Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26 – March 2, 2018, Revised Selected Papers*, pages 222–240. Springer-Verlag, 2018.

[198] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. In *2012 IEEE Symposium on Security and Privacy*, pages 239–253, 2012.

[199] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth. New results for timing-based attestation. 2012.

[200] H. Krawczyk. Cryptographic extraction and key derivation: The hkdf scheme. Cryptology ePrint Archive, Report 2010/264, `https://ia.cr/2010/264`.

[201] J. Kreidler. Ftc says: Amazon didn't protect alexa users' or children's privacy. `https://consumer.ftc.gov/consumer-alerts/2023/05/ftc-says-amazon-didnt-protect-alexa-users-or-childrens-privacy`.

[202] A. Kulshrestha and J. Mayer. Estimating incidental collection in foreign intelligence surveillance: Large-Scale multiparty private set intersection with union and sum. In *31st USENIX Security Symposium*, pages 1705–1722. USENIX Association, 2022.

[203] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler. *JEDI*: Many-to-many end-to-end encryption and key delegation for iot. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1519–1536, 2019.

[204] J. Lampkins and R. Ostrovsky. Communication-efficient mpc for general adversary structures. In M. Abdalla and R. De Prisco, editors, *Security and Cryptography for Networks*, pages 155–174, Cham, 2014. Springer International Publishing.

[205] K. Lauter, A. López-Alt, and M. Naehrig. Private computation on encrypted genomic data. In D. F. Aranha and A. Menezes, editors, *Progress in Cryptology - LATINCRYPT 2014*, pages 3–27, Cham, 2015. Springer International Publishing.

[206] C. Legislature. Title 1.81.5. california consumer privacy act of 2018, 2018.

[207] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 121–132. ACM, 2006.

[208] Y. Li, J. M. McCune, and A. Perrig. Viper: Verifying the integrity of peripherals' firmware. In *CCS*. ACM, 2011.

[209] Y. Li, J. M. McCune, and A. Perrig. VIPER: Verifying the integrity of peripherals' firmware. In *ACM CCS*, 2011.

[210] H. Lin and N. W. Bergmann. Iot privacy and security challenges for smart home environments. *Information*, 7(3):44, 2016.

[211] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. *Journal of Cryptology*, 28(2):312–350, Apr. 2015.

[212] N. Lindsey. Google data collection is more extensive and intrusive than you ever imagined. `https://www.cpomagazine.com/data-privacy/google-data-collection-is-more-extensive-and-intrusive-than-you-ever-i` 2018.

[213] H. Lipmaa. On diophantine complexity and statistical zero-knowledge arguments. In C.-S. Laih, editor, *Advances in Cryptology - ASIACRYPT 2003*, pages 398–415, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[214] H. Lipmaa, N. Asokan, and V. Niemi. Secure vickrey auctions without threshold trust, 2001. Published in Financial Cryptography 2002. helger@tcs.hut.fi 11810 received 13 Nov 2001, last revised 3 May 2002.

[215] M. Loukides. What is devops? OReilly Media, 2012. 7 June.

[216] D. W. Loveland. *Automated Theorem Proving: a logical basis*. Elsevier, 2016.

[217] I. Lutsenko and I. Protasov. Sparse, thin and other subsets of groups. *International Journal of Algebra and Computation*, 19(04):491–510, 2009.

[218] A. López-Alt, E. Tromer, and V. Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 05 2012.

[219] J. P. K. Ma and S. S. M. Chow. Secure-computation-friendly private set intersection from oblivious compact graph evaluation. ASIA CCS '22, 2022.

[220] W. Makalowski. The human genome structure and organization. *Acta biochimica Polonica*, 48:587–98, 02 2001.

[221] W. Mao. Guaranteed correct sharing of integer factorization with off-line shareholders. In *International Workshop on Public Key Cryptography*, pages 60–71. Springer, 1998.

[222] S. K. D. Maram, F. Zhang, L. Wang, A. Low, Y. Zhang, A. Juels, and D. X. Song. Churp: Dynamic-committee proactive secret sharing. *IACR Cryptology ePrint Archive*, 2019:17, 2019.

[223] E. R. Mardis. A decade's perspective on dna sequencing technology. *Nature*, 470(7333):198, 2011.

[224] U. Maurer. Secure multi-party computation made simple. In *Proceedings of the 3rd International Conference on Security in Communication Networks*, SCN'02, pages 14–28, Berlin, Heidelberg, 2003. Springer-Verlag.

[225] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P '10*, 2010.

[226] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328, 2008.

[227] K. L. McMillan. The smv system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[228] C. A. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. *1986 IEEE Symposium on Security and Privacy*, pages 134–134, 1986.

[229] P. Miao, S. Patel, M. Raykova, K. Seth, and M. Yung. Two-sided malicious security for private intersection-sum with cardinality. In *Advances in Cryptology – CRYPTO 2020*, pages 3–33. Springer International Publishing, 2020.

[230] A. Mohammed Yakubu and Y.-P. P. Chen. Ensuring privacy and security of genomic data and functionalities. *Briefings in Bioinformatics*, 21(2):511–526, 02 2019.

[231] E. Morais, T. Koens, C. Wijk, and A. Koren. A survey on zero knowledge range proofs and applications. *SN Applied Sciences*, 1, 08 2019.

[232] D. Naccache and J. Stern. A new public key cryptosystem based on higher residues. In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, CCS '98, pages 59–66, New York, NY, USA, 1998. Association for Computing Machinery.

[233] S. Nagaraja, P. Mittal, C. Hong, M. Caesar, and N. Borisov. BotGrep: Finding bots with structured graph analysis. In *Usenix Security*, 2010.

[234] S. Narain, T. D. Vo-Huu, K. Block, and G. Noubir. Inferring user routes and locations using zero-permission mobile sensors. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 397–413. IEEE, 2016.

[235] M. Naveed, E. Ayday, E. W. Clayton, J. Fellay, C. A. Gunter, J.-P. Hubaux, B. A. Malin, and X. Wang. Privacy in the genomic era. *ACM Computing Surveys (CSUR)*, 48(1):6, 2015.

[236] A. L. M. Neto, A. L. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. Loureiro, D. F. Aranha, H. K. Patil, et al. Aot: Authentication and access control for the entire iot device life-cycle. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*, pages 1–15, 2016.

[237] O. Nevo, N. Trieu, and A. Yanai. Simple, fast malicious multiparty private set intersection. In *CCS '21: ACM SIGSAC Conference on Computer and Communications Security*, pages 1151–1165, 2021.

[238] J. Noorman, J. V. Bulck, J. T. Mühlberg, F. Piessens, P. Maene, B. Preneel, I. Verbauwhede, J. Götzfried, T. Müller, and F. C. Freiling. Sancus 2.0: A low-cost security architecture for iot devices. *ACM Trans. Priv. Secur.*, 20(3):7:1–7:33, 2017.

[239] U. of Cambridge and T. Munich. Isabelle. `https://isabelle.in.tum.de/`.

[240] T. Okamoto and S. Uchiyama. A new public-key cryptosystem as secure as factoring. In K. Nyberg, editor, *Advances in Cryptology — EUROCRYPT'98*, pages 308–318, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[241] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *PODC*, pages 51–59, 1991.

[242] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, pages 223–238. Springer, Springer Berlin Heidelberg, 1999.

[243] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2005.

[244] E. Parliament and Council. Regulation (eu) 2016/679, general data protection regulation, 2016.

[245] T. P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In *Annual international cryptology conference*, pages 129–140. Springer, Springer Berlin Heidelberg, 1991.

[246] H. Perl, Y. Mohammed, M. Brenner, and M. Smith. Fast confidential search for bio-medical data using bloom filters and homomorphic cryptography. In *2012 IEEE 8th International Conference on E-Science (e-Science)*, pages 1–8, Los Alamitos, CA, USA, oct 2012. IEEE Computer Society.

[247] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. Copilot — A coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, 2004.

[248] A. Petz, G. Jurgensen, and P. Alexander. Design and formal verification of a copland-based attestation protocol. In *ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2021.

[249] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. Spot-light: Lightweight private set intersection from sparse ot extension. In *Advances in Cryptology – CRYPTO 2019*, pages 401–431. Springer International Publishing, 2019.

[250] B. Pinkas, M. Rosulek, N. Trieu, and A. Yanai. Psi from paxos: Fast, malicious private set intersection. In *Advances in Cryptology – EUROCRYPT 2020*, pages 739–767. Springer International Publishing, 2020.

[251] B. Pinkas, T. Schneider, G. Segev, and M. Zohner. Phasing: Private set intersection using permutation-based hashing. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 515–530. USENIX Association, 2015.

[252] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. *Asiacrypt*, 2009.

[253] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai. Efficient circuit-based psi with linear communication. In *Advances in Cryptology – EUROCRYPT 2019*, pages 122–153. Springer International Publishing, 2019.

[254] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based psi via cuckoo hashing. In *Advances in Cryptology – EUROCRYPT 2018*, pages 125–157. Springer International Publishing, 2018.

[255] B. Pinkas, T. Schneider, and M. Zohner. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812. USENIX Association, 2014.

[256] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on ot extension. *ACM Transactions on Privacy and Security (TOPS)*, 21:1–35, 2016.

[257] P. Porambage, M. Ylianttila, C. Schmitt, P. Kumar, A. Gurtov, and A. V. Vasilakos. The quest for privacy in the internet of things. *IEEE Cloud Computing*, 3(2):36–45, 2016.

[258] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, and N. Swamy. Verified low-level programming embedded in f*. *Proc. ACM Program. Lang.*, aug 2017.

[259] T. Rabin and M. Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, STOC '89, pages 73–85, New York, NY, USA, 1989. ACM.

[260] S. Ravi, A. Raghunathan, and S. Chakradhar. Tamper resistance mechanisms for secure embedded systems. In *VLSI Design*, 2004.

[261] P. Rindal and P. Schoppmann. Vole-psi: Fast oprf and circuit-psi from vector-ole. In *Advances in Cryptology – EUROCRYPT 2021*, pages 901–930. Springer International Publishing, 2021.

[262] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, Feb. 1978.

[263] M. Russell. Enable the security potential and versatility of sel4 in medical device development. `https://dornerworks.com/blog/sel4-medical-products/`, 2020.

[264] A. Satariano. Google is fined $57 million under europe's data privacy law. `https://www.nytimes.com/2019/01/21/technology/google-europe-gdpr-fine.html`, 2019.

[265] A. Satariano. Meta fined $1.3 billion for violating e.u. data privacy rules. `https://www.nytimes.com/2023/05/22/business/meta-facebook-eu-privacy-fine.html`, 2023.

[266] G. Sathya Narayanan, T. Aishwarya, A. Agrawal, A. Patra, A. Choudhary, and C. Pandu Rangan. Multi Party Distributed Private Matching, Set Disjointness and Cardinality of Set Intersection with Information Theoretic Security. In *CANS*, 2009.

[267] D. Schellekens, B. Wyseur, and B. Preneel. Remote attestation on legacy operating systems with trusted platform modules. *Science of Computer Programming*, 74(1):13–22, 2008.

[268] D. C. Schmidt. Google data collection. https://www.dre.vanderbilt.edu/~schmidt/PDF/google-data-collection.pdf, 2018.

[269] D. Schultz. *Mobile Proactive Secret Sharing*. PhD thesis, Massachusetts Institute of Technology, 2007.

[270] A. Sears. 'felt so violated:' milwaukee couple warns hackers are outsmarting smart homes. https://www.fox6now.com/news/felt-so-violated-milwaukee-couple-warns-hackers-are-outsmarting-smart-

[271] seL4 Team. sel4 supported platforms with verification status. https://docs.sel4.systems/Hardware/.

[272] A. Seshadri, M. Luk, and A. Perrig. SAKE: Software attestation for key establishment in sensor networks. In *DCOSS*. 2008.

[273] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SOSP*, 2005.

[274] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *IEEE Symposium on Research in Security and Privacy (S&P)*, pages 272–282, Oakland, California, USA, 2004. IEEE.

[275] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified os kernel. *ACM SIGPLAN Notices*, 48:471–482, 2013.

[276] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[277] A. Shamir. On the power of commutativity in cryptography. In *Automata, Languages and Programming*, pages 582–595, 1980.

[278] S. Shead. Amazon hit with $887 million fine by european privacy watchdog. https://www.cnbc.com/2021/07/30/amazon-hit-with-fine-by-eu-privacy-watchdog-.html, 2021.

[279] SiFive. Hifive unleashed specifications. https://www.sifive.com/boards/hifive-unleashed.

[280] S. H. Standard. Fips pub 180-2, 2002.

[281] E. Stefanov, E. Shi, and D. Song. Policy-enhanced private set intersection: Sharing information while enforcing privacy policies. In *Public Key Cryptography – PKC*, 2012.

[282] I. H. C. Study and F. P. and Xiaoming Jia and Paul J McLaren and Amalio Telenti and Paul I W de Bakker and Bruce D Walker and Stephan Ripke and Chanson J Brumme and Sara L Pulit and Mary Carrington and Carl M Kadie and Jonathan M Carlson and David Heckerman and Robert R Graham and Robert M Plenge

and Steven G Deeks and Lauren Gianniny and Gabriel Crawford and Jordan Sullivan and Elena Gonzalez and Leela Davies and Amy Camargo and Jamie M Moore and Nicole Beattie and Supriya Gupta and Andrew Crenshaw and Noël P Burtt and Candace Guiducci and Namrata Gupta and Xiaojiang Gao and Ying Qi and Yuko Yuki and Alicja Piechocka-Trocha and Emily Cutrell and Rachel Rosenberg and Kristin L Moss and Paul Lemay and Jessica O'Leary and Todd Schaefer and Pranshu Verma and Ildiko Toth and Brian Block and Brett Baker and Alissa Rothchild and Jeffrey Lian and Jacqueline Proudfoot and Donna Marie L Alvino and Seanna Vine and Marylyn M Addo and Todd M Allen and Marcus Altfeld and Matthew R Henn and Sylvie Le Gall and Hendrik Streeck and David W Haas and Daniel R Kuritzkes and Gregory K Robbins and Robert W Shafer and Roy M Gulick and Cecilia M Shikuma and Richard Haubrich and Sharon Riddler and Paul E Sax and Eric S Daar and Heather J Ribaudo and Brian Agan and Shanu Agarwal and Richard L Ahern and Brady L Allen and Sherly Altidor and Eric L Altschuler and Sujata Ambardar and Kathryn Anastos and Ben Anderson and Val Anderson and Ushan Andrady and Diana Antoniskis and David Bangsberg and Daniel Barbaro and William Barrie and J Bartczak and Simon Barton and Patricia Basden and Nesli Basgoz and Suzane Bazner and Nicholaos C Bellos and Anne M Benson and Judith Berger and Nicole F Bernard and Annette M Bernard and Christopher Birch and Stanley J Bodner and Robert K Bolan and Emilie T Boudreaux and Meg Bradley and James F Braun and Jon E Brndjar and Stephen J Brown and Katherine Brown and Sheldo. The major genetic determinants of hiv-1 control affect hla class i peptide presentation. *Science (New York, NY)*, 330(6010):1551, 2010.

[283] A. S. A. Sukor, A. Zakaria, N. A. Rahim, L. M. Kamarudin, R. Setchi, and H. Nishizaki. A hybrid approach of knowledge-driven and data-driven reasoning for activity recognition in smart homes. *Journal of Intelligent & Fuzzy Systems*, 36(5):4177–4188, 2019.

[284] Z. Sun, B. Feng, L. Lu, and S. Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[285] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in f*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[286] J. Takeshita, R. Karl, A. Mohammed, A. Striegel, and T. Jung. Provably secure contact tracing with conditional private set intersection. In *Security and Privacy in Communication Networks*, pages 352–373. Springer International Publishing, 2021.

[287] Z.-Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur. Dice*: A formally verified implementation of dice measured boot. In *USENIX Security Symposium*, 2021.

[288] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky. Packet-level signa-

tures for smart home devices. In *Network and Distributed Systems Security (NDSS) Symposium*, volume 2020, 2020.

[289] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy preserving error resilient dna searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 519–528. ACM, 2007.

[290] Trusted Computing Group. Trusted platform module (tpm), 2017.

[291] A. Ukil, S. Bandyopadhyay, and A. Pal. Iot-privacy: To be private or not to be private. In *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 123–124. IEEE, 2014.

[292] J. Vaidya and C. Clifton. Secure set intersection cardinality with application to association rule mining. *Journal of Computer Security*, 13(4), 2005.

[293] Z. Wan, J. W. Hazel, E. W. Clayton, Y. Vorobeychik, M. Kantarcioglu, and B. A. Malin. Sociotechnical safeguards for genomic data privacy. *Nature Reviews Genetics*, 23(7):429–445, July 2022.

[294] S. Wang, Y. Zhang, W. Dai, K. Lauter, M. Kim, Y. Tang, H. Xiong, and X. Jiang. HEALER: homomorphic computation of ExAct Logistic rEgRession for secure rare disease variants analysis in GWAS. *Bioinformatics*, 32(2):211–218, 10 2015.

[295] X. S. Wang, Y. Huang, Y. Zhao, H. Tang, X. Wang, and D. Bu. Efficient genome-wide, privacy-preserving similar patient query based on private edit distance. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 492–503, New York, NY, USA, 2015. Association for Computing Machinery.

[296] R. Waugh. Smart TV hackers are filming people having sex on their sofas. `http://metro.co.uk/2016/05/23/smart-tv-hackers-are-filming-people-having-sex-on-their-sofas-and-putt`

[297] R. H. Weber. Internet of things–new security and privacy challenges. *Computer law & security review*, 26(1):23–30, 2010.

[298] W. Wei. Xiaomi cameras connected to google nest expose video feeds from others. `https://thehackernews.com/2020/01/google-nest-xiaomi-camera.html`.

[299] Y. Wen, Z. Gong, Z. Huang, and W. Qiu. A new efficient authorized private set intersection protocol from Schnorr signature and its applications. *Cluster Computing*, 21(1):287–297, Mar. 2018.

[300] T. M. Wong, C. Wang, and J. M. Wing. Verifiable secret redistribution for archive system. In *IEEE Security in Storage Workshop*, pages 94–106, 2002.

[301] A. C. Yao. Protocols for secure computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, FOCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society.

[302] A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[303] L. Zajíček. Differentiability of the distance function and points of multi-valuedness of the metric projection in banach space. *Czechoslovak Mathematical Journal*, 33(2):292–308, 1983.

[304] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi. Atrium: Runtime attestation resilient under memory attacks. In *Proceedings of the 36th International Conference on Computer-Aided Design*, pages 384–391. IEEE Press, 2017.

[305] K. Zetter. Inside the cunning, unprecedented hack of ukraine's power grid. `https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/`, March 2016.

[306] Y. Zhang, M. Blanton, and G. Almashaqbeh. Secure distributed genome analysis for gwas and sequence comparison computation. *BMC Medical Informatics and Decision Making*, 15:S4, 12 2015.

[307] Y. Zhang, W. Dai, X. Jiang, H. Xiong, and S. Wang. Foresee: Fully outsourced secure genome study based on homomorphic encryption. *BMC Medical Informatics and Decision Making*, 15:S5, 12 2015.

[308] Y. Zhao and S. Chow. Are you the one to share? secret transfer with access structure. *Proceedings on Privacy Enhancing Technologies – PETS'17*, 2017.

[309] Y. Zhao and S. S. Chow. Can you find the one for me? In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, WPES'18, pages 54–65. Association for Computing Machinery, 2018.

[310] S. Zheng, N. Apthorpe, M. Chetty, and N. Feamster. User perceptions of smart home iot privacy. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–20, 2018.

[311] L. Zhou, F. B. Schneider, and R. van Renesse. Apss: proactive secret sharing in asynchronous systems. *ACM Trans. Inf. Syst. Secur.*, 8(3):259–286, 2005.

[312] J.-K. Zinzindohoué, K. Bhargavan, J. Protzenko, and B. Beurdouche. Hacl*: A verified modern cryptographic library. In *CCS*, 2017.