MOBILE AGENT SECURITY THROUGH MULTI-AGENT CRYPTOGRAPHIC PROTOCOLS

Ke Xu, B.E.

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

May 2004

APPROVED:

Stephen R. Tate, Major Professor
Armin R. Mikler, Committee Member
Ram Dantu, Committee Member
Krishna Kavi, Chair of the Department of
Computer Science and Engineering
Oscar Garcia, Dean of the College of
Engineering
Sandra Terrell, Interim Dean of the Robert B. Toulouse
School of Graduate Studies

Xu, Ke, Mobile agent security through multi-agent cryptographic protocols. Doctor of Philosophy (Computer Science), May 2004, 141 pp., 6 tables, 7 figures, references, 69 titles.

An increasingly promising and widespread topic of research in distributed computing is the mobile agent paradigm: code travelling and performing computations on remote hosts in an autonomous manner. One of the biggest challenges faced by this new paradigm is security. The issue of protecting sensitive code and data carried by a mobile agent against tampering from a malicious host is particularly hard but important. Based on secure multi-party computation, a recent research direction shows the feasibility of a software-only solution to this problem, which had been deemed impossible by some researchers previously. The best result prior to this dissertation is a single-agent protocol which requires the participation of a trusted third party. Our research employs multi-agent protocols to eliminate the trusted third party, resulting in a protocol with minimum trust assumptions.

This dissertation presents one of the first formal definitions of secure mobile agent computation, in which the privacy and integrity of the agent code and data as well as the data provided by the host are all protected. We present secure protocols for mobile agent computation against static, semi-honest or malicious adversaries without relying on any third party or trusting any specific participant in the system. The security of our protocols is formally proven through standard proof technique and according to our formal definition of security.

Our second result is a more practical agent protocol with strong security against most real-world host attacks. The security features are carefully analyzed, and the practicality is demonstrated through implementation and experimental study on a real-world mobile agent platform. All these protocols rely heavily on well-established cryptographic primitives, such as encrypted circuits, threshold decryption, and oblivious transfer. Our study of these tools yields new contributions to the general field of cryptography. Particularly, we correct a well-known construction of the encrypted circuit and give one of the first provably secure implementations of the encrypted circuit.

ACKNOWLEDGEMENTS

Deep thanks to my advisor, Dr. Steve Tate, for suggesting the topic and continuous guidance throughout the research, for numerous fruitful discussions about some of the most intriguing details in this work, for moral support and encouragement, and for providing ample opportunities for me to interact with the research community. Thanks to the committee members, Dr. Mikler and Dr. Dantu, for the hard work of reviewing this dissertation and providing helpful feedbacks. Furthermore, this research was supported in part by the National Science Foundation, Trusted Computing program, award 0208640.

Every step in my life would have been impossible without the greatest love and support from my parents. Thank you, Mom and Dad, for your indispensable role in this remarkable journey.

CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF TABLES vi

LIST OF FIGURES vii

1 INTRODUCTION 1

1.1 Mobile Agent Security 2

1.1.1 Protecting Agents with Secure Multi-Party Computation Protocols 5

1.1.2 Prior Work 6

1.1.3 Basic Idea and Drawbacks of The ACCK Protocol 7

1.2 Our Research — Secure Computation with Multi-Agents 10

1.2.1 Related Work 13

1.2.2 Organization of This Dissertation 14

1.3 The ACCK Protocol for Secure Mobile Agent Computation 15

1.3.1 The Computational Model 16

1.3.2 The Protocol 18

2 PROVABLY SECURE MOBILE AGENT COMPUTATION 22

2.1 Introduction 22

2.1.1 Definition of Security for Secure Function Evaluation 23

2.1.2 Computational Indistinguishability 28

2.2 Secure Mobile Agent Computation — The Definitions 31

2.2.1 The Ideal Model 33

2.2.2 The Real Model and The Definition of Security 36

2.3 The Universal Composition Theorem 37

2.4 Basic Tool 1: Encrypted Circuits 39

2.4.1 The Beaver-Micali-Rogaway Construction of Encrypted Circuits 40

2.4.2 The Security Flaw of The BMR Construction 42

2.4.3 Using Splitters to Correct The Problem 47

2.4.4 Proving Security for The Enhanced BMR Construction 49

2.5 Basic Tool 2: (Concurrent) Oblivious Threshold Decryption 56

2.5.1 1-out-of-2 Oblivious Transfer 56

2.5.2 Threshold Decryption 59

2.5.3 Oblivious Threshold Decryption 63

2.5.4 Concurrent OTD 67

2.6 Secure Mobile Agent Computation against Static, Semi-Honest Adversaries 69

2.6.1 Basic Tool 3: Public-Key Encryption 69

2.6.2 Secure Mobile Agent Protocol in The Hybrid Model 71

2.6.3 Proof of Security 74

2.6.4 Secure Mobile Agent Protocol in The Real Model 79

2.6.5 Communication Complexity 80

2.7 Protecting Universally Composable Protocols against Malicious Adversaries 81

2.7.1 The Goldreich-Micali-Wigderson Compiler 82

2.7.2 A Universally Composable Compiler 84

2.8 Secure Mobile Agent Computation against Static, Malicious Adversaries 86

2.8.1 The Infeasibility of Effectively Predicting Off-Path Signals 87

2.8.2 Secure Mobile Agent Protocol against Static, Malicious Adversaries 90

2.8.3 Proof of Security 91

2.8.4 Communication Complexity 93

3 A PRACTICAL MOBILE AGENT PROTOCOL 95

3.1 Introduction 95

3.2 Security against Malicious Adversaries 97

3.2.1 Secure Oblivious Threshold Decryption 97

3.2.2 More Fortifications for OTD 102

3.2.3 Protection against Other Attacks 105

3.3 Putting It Altogether 106

3.3.1 Security Definitions 106

3.3.2 The Practical Protocol for Secure Mobile Agent Computation 108

3.3.3 Security Analysis 111

4 EXPERIMENTAL RESULTS 115

4.1 Implementation Overview 115

4.1.1 The JADE System 115

4.1.2 Code Design 118

4.2 Test Configurations 121

4.3 Performance Results 123

5 CONCLUSION 129

5.1 Open Problems 130

BIBLIOGRAPHY 133

LIST OF TABLES

1.1 Comparison of the three secure computation-based approaches 7

3.1 Privacy and integrity achieved by Protocol 3.3.2. 114

4.1 Cryptographic tools used by our implementation. 121

4.2 Test parameters. 123

4.3 Performance of the ACCK protocol. 127

4.4 The initial sizes of the mobile agents in number of bytes, with 32-bit MAX circuit, 256-bit signal size, and 1024-bit keys. 128

LIST OF FIGURES

2.1 A circuit that computes $z = \max(x_1, x_2)$. 43

2.2 The encrypted subcircuit $\mathcal{M}_1$. Wires α_k get the input bits from x_1, and output wires γ_k provide the bits of y_1. 44

2.3 Basic cryptographic tools for secure mobile agent computation. 80

4.1 Implementation of the secure mobile agent computation protocol 120

4.2 Overall running time of Protocol 3.3.2 with 32-bit MAX circuit 123

4.3 Average time an agent spends on each host in Protocol 3.3.2 with 32-bit MAX circuit, and the circuit evaluation time 124

4.4 Average computation overhead per host 126

CHAPTER 1

INTRODUCTION

Mobile agents are goal-directed, autonomous programs capable of migrating from host to host during their execution, using a combination of agent technology and mobile code. Working as an agent for its user, a mobile agent can execute the user's authority and work autonomously toward a goal. It is also mobile in that the agent program can suspend its execution at one host, transfer itself—its code, data, and execution state—to another host, and resume execution there. The combination of autonomy and mobility provides mobile agents enormous potential for application in today's Internet-based, distributed computing environment. Typical application areas, to name a few, include E-commerce, information retrieval, software distribution and administration, and network management. The mobile agent paradigm offers several advantages over traditional network applications. First, mobile agents work independently from their users. The only interaction occurs at the beginning of the agent's lifetime, when it is created and sent out to remote hosts by the user, and in the end when the agent returns back to the user. This makes mobile agents an attractive paradigm for applications such as mobile computing where the communication bandwidth is limited or the user cannot or is not always connected to the network. Second, through migrating to remote hosts, mobile agents are able to bring their execution close to the resource, therefore could potentially save a large amount of communication overhead. Furthermore, compared to the client-server paradigm, the mobile agent paradigm is more adaptive to the outside world. For example, the agent can adjust its itinerary according to its current status.

Along with these notable properties and the bright prospect of a wide range of applications, the mobile agent technology also faces some serious challenges. Like many other new technologies, mobile agents suffer the lack of necessary infrastructure and widely-deployed applications. There are

mobile agent systems developed in both academia and industry, such as Telescript [66], Aglets [5], Agent Tcl [44], Concordia [67], Mole [8], etc, but none of them has achieved commercial success. Yet another even more serious hurdle for mobile agents is security. Awoken by the recent, seemly endless and ever changing security attacks on the Internet, people have realized that security is an inherent requirement for computer software, especially network-based applications. Hence, the mobile agent technology cannot achieve all its promised success without a sound security mechanism. Unfortunately, due to the unique executing scenario of mobile agents, security issues in this paradigm are mostly brand-new, as illustrated in the following section.

1.1 Mobile Agent Security

From the security perspective, a mobile agent system consists of three basic components: the agent, the originator (also known as the home host) from which the agent originates, and the remote hosts[1] that the agent visits. The distinctive feature of the mobile agent paradigm, as far as security is concerned, is that there is little trust among these basic components, except for the agent and its originator. The originator worries about hosts tampering with its agents, while the hosts are concerned about potential damage an agent could do. And for many e-commerce applications such as bidding, the hosts are competing with each other, so fair play is essential in such situation. Considering all these issues, security threats in the mobile agent paradigm can be divided into three categories [40].

- **Agents attacking hosts**. A malicious agent could try to gain unauthorized access to a host's resources or secrets, through exploiting the security holes in the host system or masquerading as an authorized agent. The agent can also launch denial-of-service attacks by consuming excessive amount of resource. So far, threats in this category have been the most studied

[1] From now on, we'll simply use "hosts" when talking about the remote hosts.

security issues in the research community. The main solution is to confine the execution of the agent in a *sandbox*, where fine-grained access control and safe code interpretation are enforced [65, 50]. In addition, code signing is employed for confirming the authenticity, origin, and integrity of the agent code [61, 38, 42, 43]. New techniques are also emerging. For example, state appraisal [33] is an advanced scheme for detecting a malicious agent, while proof carrying code [49] goes even further by allowing the producer of mobile code to attach to the code a formal proof of its safety.

- **Hosts attacking agents**. While not so obvious at the first glance, mobile agents are susceptible to attacks from malicious hosts, too. Protecting mobile agents is equally as important as protecting hosts, but much harder to deal with. First, all the attacks from the agent to the host can be done in the reverse direction. In addition, because the host provides the executing environment for the agent, it has full control of the agent's code, data, and execution. Hence eavesdropping and alteration can be easily done to the agent. For example, the host can analyze the agent's behavior by statically inspecting its code, dynamically monitoring its execution, and even running the agent repeatedly with different inputs. It can also modify the agent's code or data to its favor.

Due to the complete control the host has over the agent, it is not clear at first that anything can be done to prevent attacks against an agent. In fact, some people have gone so far as to say that it is impossible to prevent tampering unless a secure, trusted hardware environment is available at the host [26] (the trusted hardware solution is pursued by, among others, Yee in his Sanctuary project [69]). While this statement may be true when taken literally, we have seen that cryptography can make *meaningful* tampering impossible even in a software-only setting.

Turning to a software-only environment, the difficulty of protecting agents has focused most research on the problem of detecting rather than preventing the host's attacks. Techniques for this purpose include mutual itinerary recording [55], itinerary recording with replication and voting [59], execution tracing [64], and partial result encapsulation [41, 69]. Code obfuscating [39] has been proposed as a solution for protecting the agent's code against analysis or modification by the host. The main drawback, however, is the lack of complexity measures for its strength and algorithms for generating obfuscated code. In fact, recently Barak et al. has proved the impossibility of perfectly obfuscating code in theory [7].

A new direction of research came out around 1998 which applies protocols from secure multi-party computation to the mobile agent paradigm. Such protocols enable multiple participants to jointly compute a function without revealing to each other their secrets, such as each player's input data. The mobile agent paradigm can be treated as multi-party computation. The participants are the originator and the remote hosts. Collaboratively they carry out a computation represented by the mobile code. That said, there are significant differences between the settings of a typical secure multi-party computation and the mobile agent scenario. Also, secure multi-party computation has traditionally been studied as a theoretical problem, resulting many protocols only of theoretical interest. Therefore, much research needs to be done before this approach achieves a fruitful result in the practice-oriented mobile agent paradigm. This dissertation intends to make contributions to this new area.

- **Agents attacking agents**. Many components in a mobile agent platform are also agents themselves, providing directory services, communication services, etc. Some applications require interactions between agents too, such as the negotiation between a buying agent and a selling agent in an E-commerce application. As a result, an agent is also faced with attacks from other, malicious agents. Again, one effective countermeasure is restricting the agent's

behavior by a sandbox. But the sandbox is enforced by the agent platform, not the individual hosts.

1.1.1 Protecting Agents with Secure Multi-Party Computation Protocols

Sander and Tschudin [56] were the first to point out a new direction that protocols for *secure multi-party computation* could be useful in the design of a software-only solution to protect mobile agents against malicious hosts. After all, the purpose of agent computation is for the originator and the hosts together to compute some function. Secure multi-party computation [53, 68, 36] has been studied in the theoretical research community for more than two decades. There are three basic types of secure computation problems.

- **Computing with encrypted functions (CEF)**. Suppose user Alice has a function f and user Bob has some data x. Alice would like Bob to compute $f(x)$ for her, but does not want Bob to learn anything substantial about f.

- **Computing with encrypted data (CED)**. Alice has some private input x and Bob has a function f. Bob computes $f(x)$ for Alice, but learns nothing substantial about x.

- **Secure function evaluation (SFE)**. Alice and Bob both have some private inputs, x and y respectively. Jointly they evaluate a common function g on x and y and learn the output of $g(x, y)$. Neither party should learn anything substantial about the other party's input except for what $g(x, y)$ reveals.

The basic problems can be easily extended to n participants. Of the three, SFE is the most widely studied problem, with general results for evaluating any polynomial-time function. Limited results of CEF are available, such as computing polynomial and rational functions. In most cases, the code of a mobile agent is public. In fact, from the aspect of a host's protection, it is desirable

to have the agent code known and verifiable by the host. On the other hand, the data of the agent requires serious protection. (In this work, we use "agent state" to refer to all the private data of an agent, reflecting the usually dynamic nature of agent data.) Thus, SFE is a promising basis for tackling the agent security problem. Furthermore, notice that SFE protocols protect the privacy of all the participants. This means the data provided by the host to the agent computation is also treated as an object for protection. This important feature gives us the possibility of a complete security solution that protects both the agent and the host, which is a significant advantage over previous approaches in the agent area.

Since there already exist general protocols for securely evaluating any polynomial function, it seems directly applying such a protocol would solve the agent security problem once and for all. However, one big hurdle makes this impossible: in SFE, there is no limit on the rounds of communication among the participants (although protocols with constant round complexity where the constant is greater than one do exist in some situations); while the mobile agent paradigm restricts the communication between the originator and the hosts to exactly one round — sending out the agent and receiving the agent back. Currently, no SFE protocol (such as [36, 15, 25, 11]) satisfies this restriction. As this research shows, non-trivial work is needed to reconcile this difference, and our work is one of the first that seriously investigates this problem. Before introducing our approach, we first review the prior work that leads to the starting point of our work.

1.1.2 Prior Work

In an inspiring paper [56], Sander and Tschudin provided a non-interactive CEF protocol for computing polynomials and rational functions, using a homomorphic encryption scheme. Following work toward a more general function hiding scheme has varied results. Domingo-Ferrer and Herrera-Joancomartì developed a different homomorphic encryption scheme [31], which allowed full field

operations, but is unfortunately not secure against known-plaintext attacks. Loureiro and Molva presented a different function hiding scheme based on error-correcting codes, which is applicable to hiding linear functions [45].

A subsequent paper by Sander et al. [57] gave a non-interactive CED protocol for all NC^1 functions. Based on that, a non-interactive CEF protocol can be implemented by letting Alice's private input be her function f and Bob's function be a universal circuit. However, due to the logarithmic limitation on the depth of the circuit being evaluated, its application is still very limited.

Research by a group of IBM [16, 6] researchers took another approach based on Yao's SFE protocol [68]. Like the previous two, this work is able to protect the agent's code, data, as well as the host's data, but is more powerful in terms of the type of functions it can compute. Cachin et al. [16] presented a non-interactive protocol for securely evaluating any polynomial-time function. Following work by Algesheimer, Cachin, Camenisch, and Karjoth [6] gave a protocol specifically designed for the mobile agent paradigm. We refer to this last protocol as the ACCK protocol, naming it after the authors. Table 1.1 gives a comparison of these three approaches.

	Related theoretical problem	Functions that can be computed	Protecting agent's code and/or data	Protecting host's input
Sander & Tschudin [56]	CEF	Polynomials and rational functions	Code & Data	Yes
Sander et al. [57]	CED	NC^1 functions	Code & Data	Yes
IBM papers [16, 6]	SFE	Polynomial-time functions	Code & Data	Yes

Table 1.1: Comparison of the three secure computation-based approaches

1.1.3 Basic Idea and Drawbacks of The ACCK Protocol

Providing secure agent computation for any polynomial-time function, the ACCK protocol [6] is based on Yao's *encrypted circuit* and two-party SFE protocol [68]. Transformed from a normal

boolean circuit, an encrypted circuit (also known as garbled circuit in some literature) hides the signals on the wires of the "plaintext" circuit. Informally speaking, for each gate in the plaintext circuit, there is a corresponding encrypted gate. While the signals on the input and output wires of the plaintext gate are 0s and 1s, the signals on the wires of the encrypted gate are some encrypted forms of 0s and 1s, typically random binary strings. We use *signals* to specifically refer to such random strings, and *semantics* or *values* to talk about the normal boolean signals 0 and 1. The decryption keys are simply a mapping of the random strings to their corresponding semantics. In addition, for each wire in the encrypted circuit, there is a different encryption (that is, a different pair of random strings). Each encrypted gate has an associated truth table. With one set of input signals (in the form of random strings), one can evaluate the gate with the help of the truth table and get the correct output signal, which is also a binary string. More details of Yao's encrypted circuit are discussed in Chapter 2.

The strength of encrypted circuits is that they can be evaluated by an outsider without leaking any information about the inputs, the outputs, and the internal logic of the circuits. This overcomes the drawback of conventional encryption schemes whose ciphertexts are not executable unless being decrypted first. The tradeoff is that the encryption is done at the gate level, which is inefficient, in terms of time and the size of the result. Therefore, the encryption should only be done to the privacy-critical part of the mobile agent, which in many applications is fairly small and hence makes this approach practical. The rest of the agent should still be in the plaintext form.

On a high level, the ACCK protocol can be described as follows: Suppose there are ℓ remote hosts to be visited by an mobile agent. For each remote host, the originator of the mobile agent constructs an encrypted circuit for computing its function f, resulting in ℓ encrypted circuits, namely $\mathcal{C}^1, \cdots, \mathcal{C}^\ell$. The originator's private input x_0 is encrypted as the input to $\mathcal{C}^1$ — each bit of x_0 is translated into a signal on the input wires. Then a single mobile agent carries these encrypted

circuits and the encrypted x_0 to the first remote host, and executes $\mathcal{C}^1$ there with the encrypted x_0 as well as the host's input y_1. The outputs are a new state of the agent, x_1, and an output to the host, z_1, both in the form of signals. Given the decryption keys for z_1, the host is able to decrypt z_1, while x_1 is forwarded to the next host along with the agent. At the next host, the mobile agent executes $\mathcal{C}^2$ with the input being x_1 and the input from that host. This process repeats until $\mathcal{C}^\ell$ is executed at host ℓ, and the encrypted output x_ℓ is returned to the originator, where the signals are decrypted to get the final result. Of course, the hosts' inputs also need to be in the form of signals in order to be fed into the encrypted circuits. However, the signals are selected by the creator of the encrypted circuits, which is the originator. Obviously, a host cannot directly request the signals for its input from the originator, if it wants to protect its private input from the originator. The ACCK protocol uses a trusted third party T to transfer to each host the signals for its input in an *oblivious transfer* manner. The idea is simple: the originator encrypts all the signals (for both 0 and 1) of the wires for host input, using conventional public-key encryption with T's public key. The agent carries these *encrypted* signals to the hosts. Each host selects one signal for each of its input bit, and asks T to decrypt and return the plain-form signal, i.e., the binary string. Then the host can evaluate the circuit, and the originator, which is not allowed to talk to T, does not learn about the host's input[2].

Under standard cryptographic assumptions, it is computationally intractable for the host, through evaluating the encrypted circuit, to figure out any useful information about the agent's input x and the encrypted output, unless the host also knows the signals for both 0 and 1 for some wires in the circuit. For example, if the host has acquired both signals (for 0 and 1) of all the input wires, nothing can prevent it from evaluating the circuit repeatedly with different inputs to dig out secret information, such as the internal logic of the circuit, and gain unfair advantage. Therefore,

[2]More accurately, the originator learns nothing more about the hosts' inputs than what the final result reveals.

the security of the encrypted circuit relies on the assumption that the host holds only (or at most) one signal for every wire in the circuit, which in turn depends on the security of the signals on the input wires. Based on the description above, the third party T plays the central role in maintaining this security. T is trusted to not collude with any host and decrypt the encrypted signals for both 0 and 1 of any input wire. Meanwhile, for the host's privacy, T would not inform the originator of which signals the host has selected for decryption. This is a fairly strong assumption and makes the protocol less applicable to the real world. First, T is the single point of failure in this protocol. If T is compromised, the whole security will fall apart. Therefore, a significant amount of effort has to be spent in protecting T, and there is no room for error. Second, T is the bottleneck of the system. The designers of this protocol suggests that "the service of T may be offered as a public service for 'secure mobile agent computation' on the Internet" [6], which, unfortunately, could make it a perfect target for denial-of-service attacks. Third, if T is provided as a service, it may require a fee, which restricts its application. Finally, trusting sensitive or private information to a third party is an idea that will always make some people uncomfortable.

1.2 Our Research — Secure Computation with Multi-Agents

We believe the ACCK protocol is in the right direction toward achieving agent security. Particularly, notice that both the agent and the host's privacy concerns are addressed. But it suffers the drawbacks discussed above due to the use of a trusted party. Multi-agent systems[3] have been attracting much attention and been applied to many applications because of their advantages of providing parallelism, scalability, and fault-tolerance. In the security area, it provides the possibility of distributing trust over multiple entities, therefore removing the inevitable disadvantages with centralized trust assumption. This turns out to be the underlying theme of our new approach.

[3] We will use the term "multi-agents" to refer to multi-agent systems.

Based on the ACCK protocol, the primary purpose of this dissertation is to design a multi-agent protocol for secure mobile agent computation, which overcomes the shortcomings of prior work and achieves higher security.

For a higher sense of security, nothing can replace the theoretical approach of proving security. People are all too aware of old security measures fallen to new attacks. Only security that is formally proven, based on widely-accepted mathematical assumptions and according to well-defined models, can hope to stand against attacks for the many years to come. Basing on secure multi-party computation gives us a big jump start in this perspective. As an example, most of the cryptographic tools used by the prior work as well as our research have already been well studied and proved in theory. Of course, much theoretical work still needs to be done when applying these tools to a new paradigm like mobile agents. None of the prior work on mobile agents gave thorough treatment of their results from a theoretical point of view. The first major goal of this research is obtaining a provably secure agent computation protocol. We will show that such a protocol does exist with proven protection for all participants in the agent computation.

Unfortunately, a theoretically sound result usually lacks appealing efficiency in practice. Our protocol is no exception. The current state of the art in cryptography does not provide many provably secure primitives and methodologies that are also efficient. Therefore, the second part of this research aims at studying the practicality of secure agent protocols. A more efficient protocol is designed and its security properties is carefully analyzed. We argue that many security features of the theoretical protocol are maintained even after this improvement in efficiency. Next, we implement this protocol on top of a multi-agent platform, and measure the performance with a simple agent application. We also implement and study the ACCK protocol on the same platform. To our knowledge, currently there is no published experimental study on any of the prior work. We summarize the contributions of this dissertation as follows.

- *Definition of secure mobile agent computation.* One of our theoretical contribution is the definition of security for mobile agent computation. The definition follows the framework of *universally composable* security [20], which harvests the results of over a decade's work on security definition in theory, and has been widely accepted as a model for today's loosely-coupled, distributed network environment, such as the Internet. Another notable feature of our definition is the consideration of security for every participant in agent computation: the originator, the agents, and the hosts.

- *A provably secure, software-only mobile agent protocol with minimum trust assumption.* The ultimate goal of cryptographic protocols is to enable secure computation in an insecure environment. The weaker the trust assumption a protocol has, the more valuable it is. Our result is a protocol that assumes minimum trust. First of all, it does not make use of any special-purpose, trusted hardware, which also makes the protocol ready to be applied to any existing computing environment. Second, we have shown the use of a trusted third party brings quite a few drawbacks that undermines the applicability of the ACCK protocol. Our protocol makes explicit use of multi-agents and eliminates the role of the trusted party. In our protocol, no particular trust is assumed on any participants, and security is preserved as long as enough players, not necessarily any particular ones, are honest. Finally, our protocol is formally proved to achieve the defined security.

- *A practical mobile agent protocol with arguable security against most common attacks.* A practice-oriented variant of our theoretical result preserves the same framework, especially the minimum trust assumption, but improves on the efficiency. This result can be implemented on real-world agent platforms as demonstrated by our experimental study. Meanwhile, it offers effective protection against most real-world attacks, particularly attacks from the hosts.

- *Implementation and experimentation.* Both our practical secure agent protocol and the ACCK protocol have been implemented on the multi-agent system JADE. This involves also one of the first implementations of the encrypted circuit, a newly published threshold decryption scheme, and a widely used oblivious transfer protocol. The experimental results demonstrate the practicality of our research as well as the ACCK protocol, and suggests directions for future improvements.

- *Contribution to cryptography in general.* As part of our work, we have studied some fundamental cryptographic primitives extensively. From our study, we identified and corrected a flaw in a well-known construction for the encrypted circuit. As a result, we have obtained the first provably secure implementation of Yao's encrypted circuit. Our exploration of oblivious transfer and threshold decryption also bring out some new insights, both theoretically and experimentally.

We believe applying cryptographic protocols to mobile agent computation is a promising direction, with solid theoretical foundation and acceptable efficiency. Still, this is a relatively new approach, and interesting research problems exist for further exploration. We list some open problems at the end of this dissertation.

1.2.1 Related Work

Independent from our research, recently Endsuleit and Mie [32] proposed another approach of using multi-agents to protect agent security. From their paper, we can conclude the following differences between their result and ours.

- Endsuleit and Mie treat mobile agent computation as a standard multi-party SFE among the hosts, while we view the process as evaluating a sequence of two-party functions. In [32], at

any moment, a group of agents conduct joint evaluation of a function with the inputs from their current hosts. It is unclear, however, how the input from the originator is treated.

- Protecting the agents from host tampering is the main threat addressed by Endsuleit and Mie. It is not clear how attacks from the agents to the hosts are dealt with. In particular, as the agents take the hosts' inputs and conduct SFE on these inputs, preventing private information being taken away as the agents migrate is a valid concern. In contrast, our results also offer strong privacy protection to the hosts.

- Probably the biggest difference between [32] and the work of this dissertation is the SFE protocol each is based on. While we extend Yao's two-party protocol with encrypted circuits, Endsuleit and Mie use the multi-party protocol of [15]. Like Yao's, the protocol works on a boolean circuit that implement the desired function. Instead of using random strings to hide the boolean values of the signals in the circuit, these values are distributed among the parties through secret sharing. As a result, without enough parties together, these values cannot be uncovered. As the evaluation propagates through a gate, all the parties jointly compute the shares of the output value of the gate, and eventually everyone obtains only its share of this value. Thus, evaluating each gate in the circuit requires interaction among all the participants. This is in sharp contrast with Yao's encrypted circuits whose evaluation incurs no communication at all. Consequently, the communication rounds of [32] is proportional to the size of the agent function. The round complexity of the protocols we present only depends on the number of hosts, the security parameter, and the bit size of the host input.

1.2.2 Organization of This Dissertation

The rest of this Chapter introduces the ACCK protocol in details. Chapter 2 deals with the efforts toward obtaining a theoretically proven secure mobile agent protocol. We begin with the

definition of secure mobile agent computation. Then, we discuss the three building blocks of our protocol: encrypted circuits, oblivious threshold decryption, and public-key encryption. Following that is our first protocol which is proved to be secure against semi-honest adversaries, adversaries that strictly follow the protocol instructions. We then move on to enhance this protocol to stand against malicious adversaries, which may deviate arbitrarily from the prescribed behavior. We introduce a "compiler" that fortifies the original protocol with protections against cheating, and our adjustments to it when it is applied to the mobile agent scenario. Finally, the security of the enhanced protocol is proved. We also analyzed the communication complexity of both results.

Chapter 3 presents our work in the practical front. We discuss the issues that prevent the direct implementation of our theoretical results, followed by our work-arounds. The practical mobile agent protocol is presented next. Although some security are forfeited, our analysis demonstrates that the practical protocol still offers sound protection against most attacks. In Chapter 4, we report the implementation and experimental study of our practical protocol and the ACCK protocol. Chapter 5 summarizes this dissertation and discusses some interesting problems for future research.

1.3 The ACCK Protocol for Secure Mobile Agent Computation

Mobile agent computation can be treated as a special case of multi-party computation. The user, represented by the originator, uses a mobile agent to compute some function at one or more remote hosts. A unique and also appealing feature of mobile agents is that the originator interacts with the remote hosts only at the beginning and the end of the process. Once the agent is sent out, the originator can go offline, leaving all the work to the agent. In essence, the ACCK protocol [6] implements mobile agent computation as secure function evaluation process with only one-round of communication. We review this protocol with some details.

1.3.1 The Computational Model

A key contribution of the work of Algesheimer, Cachin, Camenisch, and Karjoth [6] is a clear definition of the mobile agent model, and the security goals of this model. We review this model here, with notations and definitions taken from the original paper, and a few minor additions and changes that will allow easier transition to our own definition of security. Notice that this model is aimed at capturing the basic scenario, without including methods for protecting its security.

- **Participants**: There are an originator O, a mobile agent $\mathcal{MA}$, and ℓ remote hosts $H_1, ..., H_\ell$. All participants are distrustful of each other, except the originator and the mobile agent. The agent $\mathcal{MA}$ consists of both static code and dynamic data (also called the "state" of the agent), and travels from host to host. At each host, $\mathcal{MA}$ executes the code, taking input from both its own dynamic state and the private input supplied by the host, and producing output for the current host (optional) and an updated state to be brought to the next host. In the end, $\mathcal{MA}$ returns to O, which obtains its intended result from the final state of $\mathcal{MA}$.

- **One round of communication**: Starting from O, the mobile agent $\mathcal{MA}$ visits the ℓ hosts one by one and finally returns to O from H_ℓ. The migration is the only communication among the players. In other words, the only message host H_j receives is from its predecessor H_{j-1}, which delivers $\mathcal{MA}$ to H_j. Similarly, H_j sends out only one message to deliver $\mathcal{MA}$ to H_{j+1}. The originator, it sends a single message to H_1 and receives a single message from H_ℓ.

- **Computation**: Let $\mathcal{X}$ be the set of possible states of the agent, $\mathcal{Y}$ be the set of possible inputs from the host, and $\mathcal{Z}$ be the set of all outputs that are given to the host. The agent, executing on host H_j, computes two functions

$$g_j : \mathcal{X} \times \mathcal{Y} \to \mathcal{X} \qquad \text{and} \qquad h_j : \mathcal{X} \times \mathcal{Y} \to \mathcal{Z} \ ,$$

where g_j is the "state update" function that produces the state for input to the next host the agent visits, and h_j is the function that provides output for H_j. The original state, set by the originator, is denoted by x_0, and x_j is the state that results from the computation at host H_j. Additionally, we let y_j denote the private input of host j, and z_j denote the output produced for host H_j. The functions then satisfy the following requirements:

$$x_j = g_j(x_{j-1}, y_j) \quad \text{and} \quad z_j = h_j(x_{j-1}, y_j) \ .$$

For *secure* computation with encrypted circuit, we replace the state set $\mathcal{X}$ with an encrypted state set $\mathcal{M}$, and supply algorithms for working directly with this encrypted data. The two basic functions are $\mathcal{E} : \mathcal{X} \rightarrow \mathcal{M}$ and $\mathcal{D} : \mathcal{M} \rightarrow \mathcal{X}$, for encryption and decryption, respectively. We let m_j denote the encrypted version of x_j for all $j = 0, 1, \cdots, \ell$, and let $\mathcal{A}_j$ and $\mathcal{B}_j$ denote the encrypted versions of the g_j and h_j functions, respectively, where the output of $\mathcal{A}_j$ is encrypted, and the output of $\mathcal{B}_j$ is plaintext. The functions then operate as follows, where $m_j \in \mathcal{M}$ for all $j = 0, 1, \cdots \ell$:

$$m_0 = \mathcal{E}(x_0) \quad \text{and} \quad m_j = \mathcal{A}_j(m_{j-1}, y_j) \quad \text{and} \quad z_j = \mathcal{B}_j(m_{j-1}, y_j).$$

The originator can retrieve the final output by computing $\xi = x_\ell = \mathcal{D}(m_\ell)$.

Algesheimer et al. listed two basic requirements that a secure computation should meet: correctness and privacy. Their requirement on correctness does not reflect the encrypted functions, and so we modify the correctness requirement here to require that the $\mathcal{A}_j$ functions correctly track the changes that the unencrypted g_j functions would produce:

$$x_j = \mathcal{D}(m_j) \text{ for all } j = 0, 1, \cdots, \ell.$$

Note that by the definition of the z_j values, and that the same z_j values must be produced by the $\mathcal{B}_j$ and h_j functions, the correctness of the local host outputs is also ensured.

For privacy, we require that no principal (originator or host) can learn anything from the protocol other than what follows from its own input and its own computed output. So for instance, host H_j learns nothing other than what follows from knowing y_j and z_j. The originator O should only learn ξ but nothing else about any y_j other than what follows from ξ and x_0.

1.3.2 The Protocol

The above privacy requirement protects the inputs to g_j and h_j, such that both host H_j and the agent learn nothing more about the other's input than what the outputs reveal. Theoretically, this is the secure function evaluation problem introduced by Yao [68], and the ACCK protocol is based on Yao's two-party SFE protocol. Building around an encrypted circuit[4], Yao's protocol can be described as follows. Assume Alice and Bob, with inputs x and y respectively, want to evaluate a function $g(\cdot, \cdot)$ such that in the end Bob learns the output $z = g(x, y)$ and Alice learns nothing. Let C denote a polynomial-sized circuit computing g. Let $(x_1, \ldots, x_{n_x})$, $(y_1, \ldots, y_{n_y})$, $(z_1, \ldots, z_{n_z})$ denote the binary representations of x, y, and z. Yao's protocol consists of three steps.

(1) Using a probabilistic algorithm **construct**(C), Alice constructs an encrypted circuit $\mathcal{C}$ for C. Each wire in $\mathcal{C}$ has two signals with semantics 0 and 1, respectively. Suppose the pairs of signals for the input and output wires are, presented in the order according to the signals' semantics, $\mathcal{L} = (L_{1,0}, L_{1,1}, \ldots, L_{n_x,0}, L_{n_x,1})$, $\mathcal{K} = (K_{1,0}, K_{1,1}, \ldots, K_{n_y,0}, K_{n_y,1})$, $\mathcal{U} = (U_{1,0}, U_{1,1}, \ldots, U_{n_z,0}, U_{n_z,1})$, corresponding to x, y, and z respectively. To evaluate $\mathcal{C}$ with a particular input, one needs the corresponding signals for each bit of the particular x and y. Alice sends to Bob the encrypted circuit $\mathcal{C}$, the signals corresponding to the particular

[4] See Section1.1.3 for a brief introduction, and more detailed description is given later in Chapter 2.

x value, and $\mathcal{U}$ which will be used later for recover z. Without knowing the semantics of the signals, Bob cannot deduce x from the signals.

(2) Alice and Bob engage in an oblivious transfer protocol[5] in order to transfer to Bob the signals corresponding to hiss input y. As a result, Bob receives only the signals for his particular input y and Alice does not know any bit of y.

(3) Bob evaluates $\mathcal{C}$ with the input signals for x and y, and obtains output signals $U'_1, \ldots, U'_{n_z}$. By looking up $\mathcal{U}$, Bob recovers z.

In addition, Bob can send z to Alice so that she also learns the output. Doing so does not affect the privacy of x and y. Applying Yao's scheme to the mobile agent scenario, the originator plays Alice, the hosts play Bob, and $\mathcal{C}$ is the mobile code. Conventional encryption and a trusted third party T is used to indirectly implement step (2), the oblivious transfer.

Let $C^{(j)}$ be the circuit computing $(x_j, z_j) = (g_j(x_{j-1}, y_j), h_j(x_{j-1}, y_j))$, for $1 \leq j \leq l$, with $n_x + n_y$ input bits $x_{j-1,1}, \ldots, x_{j-1,n_x}, y_{j,1}, \ldots, y_{j,n_y}$, and $n_z + n_x$ output bits $z_{j,1}, \ldots, z_{j,n_z}, x_{j,1}, \ldots, x_{j,n_x}$. Assume T has a public-key encryption scheme with the encryption and decryption operations denoted by $E_T(\cdot)$ and $D_T(\cdot)$. Also assume there is a symmetric cryptosystem whose encryption and decryption operations are $E'_\kappa(\cdot)$ and $D'_\kappa(\cdot)$ under key κ. Suppose that all parties communicate over authenticated links.

1. O chooses a string id that uniquely identifies the computation. O invokes **construct**$(C^{(j)})$ and obtains $(\mathcal{C}^{(j)}, \mathcal{L}^{(j)}, \mathcal{K}^{(j)}, \mathcal{U}^{(j)})$ as described above for each host H_j, $j = 1, \ldots, \ell$, with $\mathcal{U}^{(j)}$ consisting of $n_z + n_x$ pairs of signals in total. Let $\mathcal{U}_z^{(j)}$ denote the pairs in $\mathcal{U}^{(j)}$ with indices $1, \ldots, n_z$ and $\mathcal{U}_x^{(j)}$ denoting those with indices $n_z + 1, \ldots, n_z + n_x$. For $i = 1, \ldots, n_y$ and $b \in \{0, 1\}$, O computes

$$\overline{K}_{i,b}^{(j)} = E_T(id \| j \| i \| K_{i,b}^{(j)}).$$

[5]See Chapter 2 for details of oblivious transfer.

Let $\overline{\mathcal{K}}^{(j)}$ denote the list of pairs of all such $\overline{K}_{i,b}^{(j)}$'s. Using $U_{i,0}^{(j-1)}$ and $U_{i,1}^{(j-1)}$ as encryption keys, O also prepares two encryptions

$$E'_{U_{n_z+i,0}^{(j-1)}}(L_{i,0}^{(j)}) \text{ and } E'_{U_{i,1}^{(j-1)}}(L_{n_z+i,1}^{(j)})$$

for each $j \geq 2$ and $i = 1, \ldots, n_x$ and permutes them before assigning them to $V_{i,0}^{(j)}$ and $V_{i,1}^{(j)}$, so that the order does not give away the semantics. Call the list of such pairs $\mathcal{V}^{(j)}$. Then O let $L_i'^{(1)} = L_{i,x_{0,i}}^{(1)}$ for $i = 1, \ldots, n_x$, and sends

$$id, L_1'^{(1)}, \ldots, L_{n_x}'^{(1)}, \mathcal{C}^{(1)}, \overline{\mathcal{K}}^{(1)}, \mathcal{U}_z^{(1)}, \text{ and } \mathcal{C}^{(j)}, \overline{\mathcal{K}}^{(j)}, \mathcal{U}_z^{(j)}, \mathcal{V}^{(j)} \text{ for } j = 2, \ldots, \ell$$

to H_1 in a single message. This message is the mobile agent $\mathcal{MA}$.

2. Each host H_j, for $j = 1, \ldots, \ell$, executes this step after receiving the message from its predecessor. H_j selects $\overline{K}_i'^{(j)} = \overline{K}_{i,y_{j,i}}^{(j)}$, for $i = 1, \ldots, n_y$, as the signals representing its input y_j, and sends them to T along with $id \| j$. T decrypts $\overline{K}_i'^{(j)}$, for $i = 1, \ldots, n_y$, and verifies that the ith decrypted signal contains the identifier $id \| j$ and index i, as well as no other decryption request has been received for $id||j||i$. If all checks are successful, T returns the decrypted signals $K_1'^{(j)}, \ldots, K_{n_y}'^{(j)}$ to H_j.

3. The first host H_1 evaluates $\mathcal{C}^{(1)}$, with input $L_1'^{(1)}, \ldots, L_{n_x}'^{(1)}, K_1'^{(1)}, \ldots, K_{n_y}'^{(1)}$ and obtains $U_1'^{(1)}$, $\ldots, U_{n_z+n_x}'^{(1)}$. H_1 determines the value of output $z_1 = (z_{1,1}, \ldots, z_{1,n_z})$ such that $z_{1,i} \in \{0, 1\}$ and $U_i'^{(1)} = U_{i,z_{1,i}}^{(1)}$ for $i = 1, \ldots, n_z$, and forwards the remaining $U_{n_z+1}'^{(1)}, \ldots, U_{n_z+n_x}'^{(1)}$ as well as $\mathcal{C}^{(j)}$, $\overline{\mathcal{K}}^{(j)}$, $\mathcal{U}_z^{(j)}$, $\mathcal{V}^{(j)}$ for $j = 2, \ldots, \ell$ to H_2.

4. Every other host H_j, for $j = 2, \ldots, \ell$, interprets each of the $U_1'^{(j-1)}, \ldots, U_{n_x}'^{(j-1)}$ it has received from H_{j-1} as a symmetric key to E', determines which one of the ciphertext $V_{i,0}^{(j)}$ and $V_{i,1}^{(j)}$ the key can decrypt, and decrypts it. This yields $L_i'^{(j)}$ for $i = 1, \ldots, n_x$. Then H_j evaluates

$\mathcal{C}^{(j)}$, gets the output z_j, and forwards $U'^{(j)}_{n_z+1}, \ldots, U'^{(j)}_{n_z+n_x}$ and all the other data it received from H_{j-1} to H_{j+1}. The last host H_ℓ returns $U'^{(\ell)}_{n_z+1}, \ldots, U'^{(\ell)}_{n_z+n_x}$ to O.

5. O determines the result $\xi = (\xi_1, \ldots, \xi_{n_x})$ such that $\xi_i \in \{0, 1\}$ and $U'^{(\ell)}_i = U^{(\ell)}_{n_z+i,\xi_i}$ for $i = 1, \ldots, n_x$.

Notice that E' is not necessary. The signals outputed by the circuit $\mathcal{C}^{(j)}$, $U'^{(j)}_{n_z+1}, \ldots, U'^{(j)}_{n_z+n_x}$, can be directly feed into $\mathcal{C}^{(j+1)}$, if during the circuit construction, $\mathcal{L}^{(j+1)} = (L^{(j+1)}_{1,0}, L^{(j+1)}_{1,1}, \ldots, L^{(j+1)}_{n_x,0}, L^{(j+1)}_{n_x,1})$ is set equal to $\mathcal{U}^{(j)}_x = (U^{(j)}_{n_z+1,0}, U^{(j)}_{n_z+1,1}, \ldots, U^{(j)}_{n_z+n_x,0}, U^{(j)}_{n_z+n_x,1})$.

In a nutshell, our new secure mobile agent protocol keeps a similar framework as the ACCK protocol. But the service provided by the trusted party is replaced by distributed, threshold decryption, which is carried out by a group of mobile agents. Additional measures such as oblivious transfer are used so that no trust is assumed on any particular agent.

CHAPTER 2

PROVABLY SECURE MOBILE AGENT COMPUTATION

The appeal of provable security is obvious. The resulting guarantee of safety is not based on heuristics, which often turns out to be faulty in the security business, but on well-thought mathematical definitions and rigorous proving. Nor is security proved because all known attacks can be defeated, but because no successful attack including unforeseeable future attacks is feasible unless some fundamental complexity assumptions are broken. We intend to bring provable security to the mobile agent paradigm, and instead of tackling isolated threats, we set out for a complete solution that protects all facets of the mobile agent computation.

2.1 Introduction

Traditionally, research in secure agent computation has been done in two separate areas — protecting the hosts against malicious agents or vice versa. Consequently, the results usually address only one type of security threat with (negative) implications on the other. For instance, any technique that helps the host monitor the mobile agent's behavior is certainly beneficial to the host's security. But it also could aid a dishonest host in invading the agent's privacy. Our approach has the potential of addressing both categories of security issues at the same time. Generally speaking, we treat mobile agent computing as multi-party secure function evaluation (SFE), which has been well studied in theory. In multi-party SFE, there are n participants, collaboratively computing a function f that takes private input from every party. It is assumed that any particular party may become malicious and launch arbitrary attacks, and malicious parties may collude with each other. The goal of multi-party SFE is to protect privacy and integrity for the honest participants. Specifically, an SFE protocol tries to emulate an ideal process where all parties simply send their

inputs securely to a trusted third party who then secretly and honestly evaluates the function on these inputs and returns back to each party only its entitled output. In the ideal model, these secrecy and honest will never be compromised. In summary, treating mobile agent computation as an SFE problem incorporates all possible security threats, as no particular entity in the system is trusted; and achieves the best security possible, as the goal is to mimic the ideal model.

There may be malicious parties in the ideal model, too. However, this does affect the desired privacy and integrity. By definition, a malicious party in the ideal model is not able to peep into the internal data of any honest party or the trusted party no matter what, nor is it able to see the communication between them. Having malicious parties in the ideal model reflects the fact that some dishonest behaviors are impossible to prevent in the real world. But these behaviors are all known and their impact is so limited that they may not necessarily be regarded as attacks. First, a malicious party can substitute its original input with something else, but only before the protocol starts. Second, a party can refuse to participant in the computation, or it can abort the process at any time. Since the only job for every party is sending its input to the trusted party and then waiting for the output, aborting does little harm to others except causing the process to be incomplete and others not to receive their outputs. Third, nothing can prevent a party from analyzing the output it receives. This is incorporated in the concept that privacy means no party can learn anything more than their outputs reflect.

2.1.1 Definition of Security for Secure Function Evaluation

The first step toward addressing a security problem is, obviously, defining what *security* means. And from the above discussion, security definition is actually defining the desired ideal model and the meaning of emulating that model. Over the years, satisfactory security definitions for *stand-alone* secure function evaluation have been developed, mainly due to the work of Micali and Rogaway [46],

Beaver [9, 10], and Canetti [19]. However, all these results assume the real-world SFE protocol to be executed in an isolated environment, where there is no other computations going on. In 2001, a new paradigm was introduced aimed at studying security in a more complex environment with many different protocols (identical, related, or unrelated) being executed concurrently[1] This so-called **universally composable** (UC) security [20] is strictly stronger than the previous notions of security and it better reflects today's Internet-based computing environment[2]. The most important result is a composition theorem that allows studying a protocol in isolation but guarantees security when it is executed with others.

Our work considers security under the UC framework. The goal is to obtain secure, UC protocols for mobile agent computation against various kinds of adversaries. In the following, we informally summarize the UC security paradigm.

- **Participants in the UC framework**. First, there are the parties that want to jointly evaluate a function. Since multiple parties can become malicious and malicious parties may also collude with each other, it is instructive to imagine a new entity in the system called *the adversary* who can corrupts any party and so make that party malicious, and coordinates the behavior of the corrupted parties. In fact, the corrupted parties can be thought of as dummy parties that simply carry out instructions from the adversary. So in the following we describe the behaviors of corrupted parties through the description of the adversary, and ignore mentioning the corrupted parties unless necessary. Unlike the stand-alone model, in the UC scenario, there is another (adversarial) entity $\mathcal{Z}$ called the *environment*, that represents everything outside of the protocol in question, such as the upper layer that invokes the protocol as a subprocess, or another irrelevant protocol which is concurrently going on. The

[1] There are some other works on sequential or concurrent composition of protocols. The notion introduced here seems to be the most comprehensive one and has been widely accepted.

[2] Works have shown that protocols that are secure by itself may be vulnerable to attacks when executed with other protocols, known as protocol composition, with parallel composition the most troublesome [52, 19].

environment provides the inputs to all the parties and the adversary, and reads their outputs[3]. However, $\mathcal{Z}$ does not see the "internals" of the protocol execution; i.e. the internal states of the parties as well as the communication between the parties. On the other hand, $\mathcal{Z}$ can freely communicate with the adversary during the whole process. All the parties including the adversary and the environment are formalized as interactive Turing machines, which is a Turing machine with multiple tapes: an identity tape, a security parameter tape, a random tape, an input tape and an output tape, an incoming and an outgoing communication tape, a work tape, and an activation tape. In the following, we use *party* to refer to a participant in the system other than the adversary, the environment, and the later introduced ideal functionality.

- **The communication model**. To better reflect the real world, communication channels in the UC framework are open and asynchronous without guaranteed delivery of messages. Thus, everybody including the adversary is given the power to see the contents of all the messages in the system. More important, the adversary controls the message delivery. The only exception is the communication in the ideal model between the ideal functionality and the parties. On the other hand, we do assume authenticated communication lines, meaning the adversary cannot modify or forge messages sent by honest parties, nor can it deliver such a message more than once. (This is a simplification adopted from [24] as the basic model in [20] actually considers unauthenticated communication channels. See the discussion of $\mathcal{F}_{\text{AUTH}}$ in [20] for details.)

- **The ideal process**. For each SFE problem, an ideal model is defined that captures the desired security goals for that problem. In this model, there is yet one more entity $\mathcal{F}$ known as the ideal functionality, who receives the private input from every party in the computation,

[3] The input and output of the adversary can be anything.

computes the function on the inputs, and returns to each party its corresponding output. The ideal functionality is assumed incorruptible, and since it does all the work, the parties in the ideal model are dummy parties that only forward their inputs and outputs between the environment and the ideal functionality. The communication between a party and the ideal functionality is ideally authenticated and is kept perfectly secret from other players in the system including the adversary, denoted by $\mathcal{S}$ in the ideal model. However, the adversary still controls the delivery of messages from $\mathcal{F}$ to the parties. That is, although $\mathcal{S}$ cannot see the contents of the message, it does know the destination (i.e., which party to receive this message) and it may opt to not deliver the message to that party. We stress $\mathcal{S}$ has no control on the messages from a party to $\mathcal{F}$ unless it has corrupted that party. Since the ideal functionality is incorruptible, the impact of the adversary is very limited. Indeed, the only effective adversarial behaviors are substituting the original inputs of corrupted parties with arbitrary inputs, letting the corrupted parties abort the computation prematurely, not delivering a message from $\mathcal{F}$ to a party, or in the case of an adaptive adversary, corrupting an honest party and obtaining its history of inputs and outputs. As said before, these are the behaviors that are impossible to protect from in the real world. Therefore, the ideal process captures the best security one can hope for.

- **The real model**. The real model is where the cryptographic protocol for secure function evaluation is executed. The communication channels are **point-to-point**, **open**, **asynchronous**, and **authenticated** for simplicity, without guarantee of message delivery. The adversary, denoted by $\mathcal{A}$ in the real model, can be either *semi-honest* (also called passive or honest-but-curious in some literature) who follows the protocol but may try to deduce more from the information it collects, or *malicious* (or active) who can deviate arbitrarily from the protocol. $\mathcal{A}$ can statically corrupt a fixed set of parties before the protocol starts or adaptively corrupts

honest parties as the protocol execution proceeds. Both the adversary and the environment are assumed to be polynomial-time and so are the parties. There may be an upper bound on the number of corrupted parties as well as some set-up assumptions such as the availability of a common reference string.

- **The execution**. In both the ideal model and the real model, the execution consists of a sequence of activations. The sequence always begins with the activation of the environment and ends with an environment activation in which $\mathcal{Z}$ does not provide input to any party in the system and the adversary. If $\mathcal{Z}$ writes an input on some party's or the adversary's input tape, that entity is activated next and will proceed as its program specifies. Communication between the parties, the adversary, and the ideal functionality may lead to further activations of the receiving entities. Eventually, activation comes back to $\mathcal{Z}$ which may start another round by providing a new input. Therefore, the UC model also captures the scenario of reactive systems. Note that at any time, only one party is activated.

- **The security definition.** After the execution is over, $\mathcal{Z}$ outputs a single bit. A protocol is deemed secure or referred to as securely realizing an ideal functionality if for every real-model adversary $\mathcal{A}$, there exists a corresponding ideal-model adversary $\mathcal{S}$ such that the running time of $\mathcal{S}$ is polynomially related to the running time of $\mathcal{A}$ and the *ensembles* of the outputs of $\mathcal{Z}$ in these two models are *indistinguishable*. In other words, no environment can tell whether it is interacting with the ideal model and adversary $\mathcal{S}$ or the real model and adversary $\mathcal{A}$. Because the ideal-model adversary's behavior is restricted to only those that are impossible to protect, by showing the ideal adversary can *simulate* the real model computation, it is proved that at the very best, what the real-model adversary can do to the SFE protocol execution amounts to the attacks its corresponding ideal-model adversary is able to do. Hence, the security of

the protocol follows[4].

- **The impact of no guarantee of message delivery**. In both the ideal and the real model, if a message is not delivered, a party could wait for that message forever and never generates an output. Therefore, the security definition concerns only the case when a protocol generates output. If in the real model a party P hangs and never generates output, as long as in the ideal model P also hangs (because the ideal adversary does not deliver the message from $\mathcal{F}$ to P), it is considered secure. As a corollary, a protocol that sends no messages and generates no output, securely realizes any ideal functionality. A *non-trivial protocol* has the property that if the real-life adversary delivers all messages and does not corrupt any party, the ideal adversary has to deliver all messages in order to simulate the real-model execution. We only consider non-trivial protocols in our work.

- **Synchronization in obtaining inputs**. In traditional multi-party computation, every party is assumed to have its input ready when the protocol starts. In the UC framework, the environment provides input to one party at a time. As later shown in the security proof, it is important to have all hosts receive their inputs before the originator sends out the agents. To this end, in our real-model protocol every host notifies the originator after it has received input.

2.1.2 Computational Indistinguishability

In essence, the environment $\mathcal{Z}$ serves as an (interactive) distinguisher to the real model execution of a UC protocol and its ideal model simulation. However, $\mathcal{Z}$ is much more powerful (and hence much less likely to be fooled) than the distinguisher in the traditional, stand-alone notion of security. Rather than being a passive bystander, $\mathcal{Z}$ takes an active part in the protocol execution by providing

[4] In the previous introduction, security is described as for a protocol to emulate the ideal model. The two concepts are equivalent.

inputs to the protocol and exchanging information freely with the adversary. In fact, one can think of $\mathcal{Z}$ as the actual adversary while adversary $\mathcal{A}$ does nothing other than carrying out $\mathcal{Z}$'s commands. However, due to $\mathcal{Z}$'s role as the distinguisher, the ideal adversary $\mathcal{S}$ which tries to simulate the real model has only limited access to $\mathcal{Z}$. In particular, two crucial points are 1) $\mathcal{S}$ has only black-box access to $\mathcal{Z}$ and 2) $\mathcal{Z}$ cannot be rewound[5]. Intuitively, this stronger notion of indistinguishably ensures the security in a complex, concurrent execution environment. With this in mind, we formally define the notion of computational indistinguishability.

Definition 2.1.1 *Probability Ensemble (Definition 3.2.1 of [35])*

Let I be a countable index set. ***An ensemble indexed by*** *I is a sequence of random variables indexed by I. Namely, any $X = \{X_i\}_{i \in I}$, where each $X(i)$ is a random variable, is an ensemble indexed by I.*

In this paper, we consider random variables of binary strings. A random variable X_n ranges over strings of length $poly(n)$[6], for an integer n.

Definition 2.1.2 *Polynomial-Time Indistinguishability (Definition 3.2.2 of [35])*

Two ensembles, $X \stackrel{\text{def}}{=} \{X_n\}_{n \in \mathbf{N}}$ and $Y \stackrel{\text{def}}{=} \{Y_n\}_{n \in \mathbf{N}}$, are ***indistinguishable in polynomial time*** *if for every probabilistic polynomial-time algorithm D, every positive polynomial $p(\cdot)$, and all sufficiently large n's,*

$$|\Pr[D(X_n, 1^n) = 1] - \Pr[D(Y_n, 1^n) = 1]| < \frac{1}{p(n)}$$

[5] A powerful technique in security proof is for a simulator to rewind its simulation back to a previous point and try another way. Traditionally, rewinding is allowed in black-box simulation, and has been widely used in problems such as zero-knowledge proofs (e.g., [52]). However, the use of rewinding may cause problems in proving a general composition theorem [18, 21].

[6] $poly(x)$ refers to some polynomial of x.

Definition 2.1.3 *Indistinguishability by Repeated Sampling (Definition 3.2.4 of [35])*

Two ensembles, $X \stackrel{\text{def}}{=} \{X_n\}_{n\in\mathbf{N}}$ *and* $Y \stackrel{\text{def}}{=} \{Y_n\}_{n\in\mathbf{N}}$ *are* ***indistinguishable by polynomial-time sampling*** *if for every probabilistic polynomial-time algorithm* D*, every two positive polynomials* $m(\cdot)$ *and* $p(\cdot)$*, and all sufficiently large* n*'s,*

$$|\Pr[D(X_n^{(1)},\ldots,X_n^{(m(n))})=1]-\Pr[D(Y_n^{(1)},\ldots,Y_n^{(m(n))})=1]|<\frac{1}{p(n)}$$

where $X_n^{(1)}$ *through* $X_n^{(m(n))}$ *and* $Y_n^{(1)}$ *through* $Y_n^{(m(n))}$ *are independent random variables, with each* $X_n^{(i)}$ *identical to* X_n *and each* $Y_n^{(i)}$ *identical to* Y_n*.*

An ensemble is *efficiently constructible* if its sampling can be done within polynomial time. The following theorem asserts that for two efficiently constructible ensembles, indistinguishability by a single sample implies indistinguishability by repeated sampling.

Theorem 2.1.4 *(Theorem 3.2.6 of [35])*

Let $X \stackrel{\text{def}}{=} \{X_n\}_{n\in\mathbf{N}}$ *and* $Y \stackrel{\text{def}}{=} \{Y_n\}_{n\in\mathbf{N}}$ *be two polynomial-time-constructible ensembles, and suppose* X *and* Y *are indistinguishable in polynomial time (as in Definition 2.1.2). Then* X *and* Y *are indistinguishable by polynomial-time sampling (as in Definition 2.1.3).*

As a special case, the distributions of the outputs of the environment constitute a *binary distribution ensemble*, $X = \{X_{(k,a)}\}_{k\in\mathbf{N},a\in\{0,1\}^*}$, which only consists of distributions over $\{0,1\}$. In the index (k,a), k corresponds to the security parameter, and a represents the input.

Definition 2.1.5 *Indistinguishability of Two Binary Distribution Ensembles [20]*

Two binary distribution ensembles X *and* Y *are* ***indistinguishable*** *(written* $X \stackrel{c}{\approx} Y$*) if for any* $c \in \mathbf{N}$ *there exists* $k_0 \in \mathbf{N}$ *such that for all* $k > k_0$ *and for all* a *we have*

$$|\Pr(X_{(k,a)}=1)-\Pr(Y_{(k,a)}=1)|<\frac{1}{k^c}.$$

In this work, we consider static, semi-honest and malicious adversaries as our threat model. We begin with defining the ideal agent computation, as well as the real model with either a static, semi-honest adversary or a static, malicious adversary. Then security in these two cases are defined. After laying down the ground, the rest of this chapter presents and proves a secure protocol for agent computation in each of the two cases.

2.2 Secure Mobile Agent Computation — The Definitions

The purpose of a mobile agent's visit to a host is to perform some computation on the input provided by the host. The result is taken by the agent to the next hop and so on, eventually back to the originator. In many applications, the computation also takes as input the result(s) from previous host(s). For example, at each host, a shopping agent compares the lowest price it has collected so far with the price offer from the current host, and result is a new best offer. In such cases, the originator usually sets an initial value for this input. Furthermore, the host may receive output from the agent computation, too. As an example, a shopping agent may even make buying decisions on behalf of the originator. Accounting for all these situations, we model the agent computation as the process of computing a set of two-party (i.e., two-inputs) functions, one function per host. The two inputs are the one from the host and the one from the agent which we refer to as the agent's current *state*. Likewise, there are two outputs, one to the host and one to the agent which will be its updated state. The agent functions can be any polynomial-time computable functions (with regard to some security parameter). We impose an additional requirement that the agent function can be implemented in multi-agents. That is, the overall task can be accomplished by the collaboration of multiple mobile agents, each computing a sequence of polynomial-time two-party functions. In practice, this usually means there is a way for the originator to combine the results of

the multiple agents into one final result. In order to achieve the desired security, the protocols we are going to present require static number of agents. Each agent's itinerary is determined before the protocols start, which implies all the hosts are predetermined, and there should be no overlap between any two agent's itineraries. We believe these restrictions can be easily accommodated by many mobile agent applications, especially e-commerce applications. (For instance, one can execute a setup process to locate and partition the hosts.) Assuming all the above conditions are met, we have the following notations regarding the identities of the hosts and the functions to be computed at each host.

Definition 2.2.1 *Host Identities*

Assume there are a total of ℓ hosts and n agents in the system, where $n \leq \ell$. The hosts are partitioned into n disjoint subgroups, each group to be visited by exactly one agent. We use the combination of the group id i, for $i = 1, \ldots, n$, and the host id j within that group, for $j = 1, \ldots, m_i$ and $\sum_{i=1}^{n} m_i = \ell$, to identify a particular host $H_{(i,j)}$.

The agent functions are two-party polynomial-time functions whose output is also divided into two parts. The inputs are the current agent state and input from the current host. The function outputs a new agent state and an output to the current host. Furthermore, as implied by the interactive Turing machine model, there are two additional inputs to the agent functions. One is the security parameter, and the other is a random input (i.e., a random binary string) to be used by the randomized operations in the computation. In practice, the security parameter is fixed and reflected in the choice of key size, etc.; and the random input is included in the input from the agent and the host. They are singled out here for the purpose of analysis.

Definition 2.2.2 *Agent Functions*

Let $f^{(i,j)} : 1^* \times \{0,1\}^* \times \{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m_i$ as defined above, denote the agent function for host $H_{(i,j)}$. The first parameter is the security

parameter and is represented in unary, while the last parameter is the random input. The second parameter is the current agent state, and the third parameter is the input from the host. The function outputs an updated agent state as well as a local output to the host. We are interested in functions that are computable in time polynomial to the length of the security parameter. Also, the lengths of the inputs and outputs are polynomially bounded by the security parameter. Let $f_1^{(i,j)}(1^k, x, y, r)$ *and* $f_2^{(i,j)}(1^k, x, y, r)$ *denote the first (i.e., to the agent) and the second outputs of function* $f^{(i,j)}$ *on inputs* 1^k, x, y, r. *Furthermore, let* $x_{(i,0)}$ *for* $i = 1, \ldots, n$ *denote the initial state of agent* MA_i, $y_{(i,j)}$ *be the input from host* $H_{(i,j)}$, *and* $r_{(i,j)}$ *be the random input. Agent* MA_i, *visiting hosts* $H_{(i,1)}, \ldots, H_{(i,m_i)}$, *computes function* $f^{(i,j)}$ *at host* $H_{(i,j)}$ *(for* $j = 1, \ldots, m_i$*) on the current agent state and the host's input, and obtains the following outputs.*

$$
\begin{aligned}
x_{(i,j)} &= f_1^{(i,j)}(1^k, x_{(i,j-1)}, y_{(i,j)}, r_{(i,j)}) \quad \text{for } 1 \leq j \leq m_i \\
z_{(i,j)} &= f_2^{(i,j)}(1^k, x_{(i,j-1)}, y_{(i,j)}, r_{(i,j)}) \quad \text{for } 1 \leq j \leq m_i
\end{aligned}
$$

$x_{(i,j)}$ *is the updated agent state, and* $z_{(i,j)}$ *is the local output to host* $H_{(i,j)}$.

Finally, all ℓ *agents return to the originator* O *with final states* $x_{(i,m_i)}$, *for* $i = 1, \ldots, n$, *which are combined to the final output to* O, *denoted by* $\xi = g(1^k, x_{(1,m_1)}, \ldots, x_{(n,m_n)}, r)$, *where* $g(\cdots)$ *is the combination function.*

2.2.1 The Ideal Model

The participants in the ideal-model agent computation are an originator O, ℓ hosts, the ideal functionality $\mathcal{F}_{\mathrm{MA}}$, an adversary $\mathcal{S}$, and the environment $\mathcal{Z}$. We assume as a pre-stage, the hosts are partitioned into n disjoint subsets. The partition is reflected by the hosts' ids. All participants are modeled as interactive Turing machines as introduced before. It is the ideal functionality that computes the desired functions which are defined in Definition 2.2.2. The originator and the

hosts do nothing other than forwarding inputs and outputs between the environment and the ideal functionality. $\mathcal{S}$ controls the delivery of the messages from $\mathcal{F}_{\mathrm{MA}}$ to O and the hosts, and can corrupt any party including O at its will. A corrupted party essentially becomes silent and is no longer activated, while the adversary takes part in the process on behalf of it. Our work assumes a static $\mathcal{S}$, which corrupts a pre-determined set of parties before the protocol starts and no more thereafter. After controlling some parties, $\mathcal{S}$ can substitute their inputs from the environment with arbitrary inputs. $\mathcal{S}$ can also not forward the inputs to the ideal functionality. Furthermore, $\mathcal{S}$ has its own communication with both $\mathcal{Z}$ and $\mathcal{F}_{\mathrm{MA}}$. The behavior of the ideal functionality is defined as follows.

Definition 2.2.3 *Ideal Functionality* $\mathcal{F}_{\mathrm{MA}}$

$\mathcal{F}_{\mathrm{MA}}$ *proceeds as follows, interacting with originator* O*,* ℓ *hosts* $H_{(i,j)}$*, with* i *and* j *as in Definition 2.2.1, and adversary* $\mathcal{S}$*.*

1. *Upon receiving the initial agent states* $x_{(1,0)}, \ldots, x_{(n,0)}$ *with session id sid from* O*, store them in a buffer. Send a notification* $(O-\mathrm{input}, sid)$ *to* $\mathcal{S}$*, where "*$O-\mathrm{input}$*" is just a text message containing no information about the input value. From now on, only respond to messages with the same session id as sid.*

2. *Upon receiving input* $y_{(i,j)}$ *from host* $H_{(i,j)}$*, send notification* $(H_{(i,j)}-\mathrm{input}, sid)$ *to* $\mathcal{S}$*. If there does not exist* $x_{(i,j-1)}$*, buffer* $y_{(i,j)}$*. Otherwise, compute* $x_{(i,j)}$ *and* $z_{(i,j)}$ *with uniformly chosen random input* $r_{(i,j)}$*, store* $x_{(i,j)}$ *and return* $z_{(i,j)}$ *to* $H_{(i,j)}$*. In addition, if there are buffered* $y_{(i,k)}$ *for* $k = j+1, j+2, \ldots$*, continue computing the corresponding* $x_{(i,k)}$ *and* $z_{(i,k)}$*, and return* $z_{(i,k)}$ *to host* $H_{(i,k)}$*, until either the computation cannot proceed or there are no more buffered inputs.*

3. *Upon receiving a request to reveal intermediate result* $x_{(i,j-1)}$ *from both* O *and host* $H_{(i,j)}$*, send* $x_{(i,j-1)}$ *to* $\mathcal{S}$*.*

4. *Finally, combine the final agent states* $x_{(1,m_1)}, x_{(2,m_2)}, \ldots, x_{(n,m_n)}$ *with uniformly random input* r *as defined in Definition 2.2.2, and return the result* ξ *to* O*. If having received a request from* O *to reveal the final agent states, send all the final agent states to* $\mathcal{S}$*. Halt after this step is done.*

5. *Each host* $H_{(i,j)}$ *outputs* $z_{(i,j)}$*, and* O *outputs* ξ*.*

Recall that the environment $\mathcal{Z}$ provides inputs to all the parties and reads every party's output. It may also exchange information with $\mathcal{S}$. Eventually, $\mathcal{Z}$ outputs a single bit (potentially based on all the information it has collected, which is referred to as its *view* of the process). Let $\text{IDEAL}_{\mathcal{F}_{\text{MA}},\mathcal{S},\mathcal{Z}}(k,a)$ denote the binary distribution of $\mathcal{Z}$'s output induced by the random choices of $\mathcal{F}_{\text{MA}}$, $\mathcal{S}$, and $\mathcal{Z}$ and indexed by k the security parameter and a the inputs. Let $\text{IDEAL}_{\mathcal{F}_{\text{MA}},\mathcal{S},\mathcal{Z}}$ denote the ensemble $\{\text{IDEAL}_{\mathcal{F}_{\text{MA}},\mathcal{S},\mathcal{Z}}(k,a)\}_{k\in\mathbf{N},a\in\{0,1\}^*}$.

A few comments about the ideal functionality. We stress that the communication channels are ideally authenticated. As a result, unless corrupting a party, $\mathcal{S}$ cannot send messages to $\mathcal{F}_{\text{MA}}$ on behalf of that party. The UC framework assumes potentially multiple agent computations as well as other unrelated computations going on concurrently. It is the session id that identifies each computation. $\mathcal{F}_{\text{MA}}$ ignores all incoming messages other than those defined. Furthermore, because it halts after the session is completed, a new instance of $\mathcal{F}_{\text{MA}}$ must be invoked for another agent computation. The returning of some intermediate agent states to the adversary reflects the fact that currently the best agent protocol cannot hide the intermediate agent states from an adversary if it corrupts both the originator and the next host the agent is going to visit. Similar reason for returning the final states of all the agents to the adversary. We leave as an interesting open problem to come up with an agent protocol that removes both shortcomings.

2.2.2 The Real Model and The Definition of Security

In the real model, there is no ideal functionality available. Rather, the originator and the hosts execute an agent protocol π (to be presented later), with the goal to mimic the ideal-model process. Like before, we assume the hosts are partitioned into n disjoint subgroups and the partition is known by everyone. In addition to the standard communication model of the UC framework, we stress the unique communication restriction of the mobile agent paradigm. Unlike a typical SFE problem in which every party may communicate with others at any time during the protocol, the mobile agent paradigm restricts the originator to communicate with the hosts only at the beginning and the end of the process.

The adversary in the real model, denoted by $\mathcal{A}$, controls all corrupted parties and coordinates their behaviors. We distinguish two types of adversaries. A *semi-honest* adversary let the corrupted parties follow the instructions of protocol π, while a *malicious* adversary may have the parties deviate arbitrarily from the protocol. Again, we consider only the *static* variants of both types of adversaries. That is, $\mathcal{A}$ only corrupts parties before π starts. As a consequence, $\mathcal{A}$'s decision of whom to corrupt does not depends on the protocol execution. The behavior of the environment is exactly the same as in the ideal model. Furthermore, both the adversary and the environment are polynomial-time bounded with regard to the security parameter. Let $\mathrm{REAL}_{\pi,\mathcal{A},\mathcal{Z}}$ denote the binary distribution ensemble of $\mathcal{Z}$'s output in the real model.

Definition 2.2.4 *Secure Mobile Agent Computation against Static Adversaries*

*Mobile agent protocol π is said to be **secure against static, semi-honest (or malicious) adversaries under condition** t, if for any such adversary $\mathcal{A}$ interacting with the real-model execution of π and corrupting parties subject to condition t, there exists a corresponding adversary $\mathcal{S}$ corrupting the same set of parties in the ideal model, such that the running time of $\mathcal{S}$ is polynomial*

in the running time of $\mathcal{A}$ and for all environment $\mathcal{Z}$ we have

$$\text{REAL}_{\pi,\mathcal{A},\mathcal{Z}} \overset{c}{\approx} \text{IDEAL}_{\mathcal{F}_{\text{MA}},\mathcal{S},\mathcal{Z}}. \tag{2.1}$$

In the following, we present a secure mobile agent protocol in each of the two adversarial cases and prove their security according the above definitions. Our protocols rely heavily on several fundamental cryptographic primitives, and the proofs utilize an important universal composition theorem that enables us to consider the security of the primitives individually and then prove the security of the whole protocol on top of them. We introduce this composition in the next section.

2.3 The Universal Composition Theorem

The virtue of the notion of universally composable security is that the obtained security is maintained when the single protocol instance is composed, sequentially or concurrently, with arbitrarily many other protocol executions. This is guaranteed by a universal composition theorem [20], and consequently we can have a divide-and-conquer approach to the design of the secure mobile agent protocols. For easy reference, we summarize the universal composition theorem of [20] as follows, tailored to suit the agent scenario.

Definition 2.3.1 *Real-Model Execution with Concurrent Subprotocols*

The real-model execution of protocol π with a set of concurrent subprotocols $\hat{\rho}$ is defined the same as the standard real model. During the process there can be multiple instances of protocols in $\hat{\rho}$ executed concurrently by subsets of the parties as required by π. Denote by $\text{REAL}_{\pi^{\hat{\rho}},\mathcal{A},\mathcal{Z}}$ the ensemble of the environment's output.

Definition 2.3.2 *Hybrid-Model Execution with Ideal Functionalities*

In the hybrid-model execution of protocol π *with access to a set of ideal functionalities* $\hat{\mathcal{F}}$ *that computes a group of subfunctions* $\hat{f}$*, the parties and the adversary (denoted by* $\mathcal{H}$ *in the hybrid model) behaves the same as in the real model with the following exception. When a subset of parties need to evaluate a function in* $\hat{f}$*, the parties start an ideal-model evaluation with a copy of the corresponding ideal functionality from* $\hat{\mathcal{F}}$*. We stress there can be multiple ideal processes going on concurrently. Denote by* $\mathrm{HYB}^{\hat{\mathcal{F}}}_{\pi,\mathcal{H},\mathcal{Z}}$ *the ensemble of the environment's output.*

Theorem 2.3.3 *The Universal Composition Theorem [20]*

Let C *be a class of real-model adversaries. Let* π *be an* n*-party protocol with calls to a set of subfunctions* $\hat{f}$*. Let* $\hat{\mathcal{F}}$ *be the corresponding group of ideal functionalities that computes* $\hat{f}$*, and* $\hat{\rho}$ *be the set of universally composable protocols that securely realizes the ideal functionalities in* $\hat{\mathcal{F}}$ *with respect to* C *in the real model. Then for any real-model adversary* $\mathcal{A} \in C$ *there exists a hybrid-model adversary* $\mathcal{H} \in C$ *such that for all environment* $\mathcal{Z}$ *it holds that*

$$\mathrm{REAL}_{\pi^{\hat{\rho}},\mathcal{A},\mathcal{Z}} \stackrel{c}{\approx} \mathrm{HYB}^{\hat{\mathcal{F}}}_{\pi,\mathcal{H},\mathcal{Z}}. \tag{2.2}$$

Due to the composition theorem, our first agent protocol, in the case with static, semi-honest adversaries, is presented in three steps. We begin with three building blocks: Yao's encrypted circuit, a UC subprotocol for *Concurrent Oblivious Threshold Decryption* (COTD), and a UC subprotocol for public key encryption (PKE). Next, we present an agent protocol in a hybrid model where calls to COTD and PKE are directed to their respective ideal functionalities, $\mathcal{F}_{\mathrm{COTD}}$ and $\mathcal{F}_{\mathrm{PKE}}$, and prove its security by showing that for any hybrid-model adversary $\mathcal{H}$, there exists

an ideal-model adversary $\mathcal{S}$ such that for all environment we have

$$\mathrm{HYB}_{\pi,\mathcal{H},\mathcal{Z}}^{\mathcal{F}_{\mathrm{COTD}},\mathcal{F}_{\mathrm{PKE}}} \stackrel{c}{\approx} \mathrm{IDEAL}_{\mathcal{F}_{\mathrm{MA}},\mathcal{S},\mathcal{Z}}. \tag{2.3}$$

Finally, applying the composition theorem and combining equations (2.2) and (2.3), we prove the security of the real-model agent protocol.

2.4 Basic Tool 1: Encrypted Circuits

Encrypted circuits[7] were introduced by Yao [68] as the core component of his classic two-party SFE protocol. Simply put, an encrypted circuit can be obliviously evaluated without revealing any information — the input, the internal state, and the output — to the evaluator. This nice property makes it a natural choice as the form of mobile agents, which are to be evaluated in untrusted environment. Yao's paper contains no details on how such circuits are constructed, and subsequent authors have given concrete constructions of encrypted circuits. The representative work include those of Goldreich, Micali, and Wigderson [36], Naor, Pinkas, and Sumner [47], and Beaver, Micali, and Rogaway [11]. All these work achieve the same security goal (but with different assumptions). We adopt the construction of Beaver, Micali, and Rogaway, which we'll refer to as the *BMR construction*, because of its simplicity. Furthermore, Rogaway has formally proved the security of the BMR construction in his dissertation [54]. However, we have discovered a flaw in the BMR construction which could seriously undermine its security in some cases. In the following, we describe the original BMR construction, our correction of the flaw, and prove the security of the corrected construction.

[7] Also known as garbled circuits or scrambled circuits in some literature.

2.4.1 The Beaver-Micali-Rogaway Construction of Encrypted Circuits

The main result of the Beaver-Micali-Rogaway paper [11] is a constant round protocol for multi-party secure function evaluation. The encrypted circuit plays an important role in this protocol. The setting of the BMR paper is quite different from the mobile agent scenario (but the construction is general for our use). Hence, details relating to their setting are discarded in our description of the BMR construction. Furthermore, the original construction was designed as an n-party protocol, where the encrypted circuit is constructed based on the contribution from all parties, and no single party has more impact on the circuit construction than others. In contrast, the encrypted circuit in our mobile agent protocol is created solely by the originator. Therefore, we tailor the original construction so as to suit our case. For simplicity, we assume that every gate in the circuit has at most two inputs but may have unbounded fan-out. Gates with more inputs can be easily transformed into these types.

As summarized in [34], the basic idea behind all the aforementioned constructions for encrypted circuits is to hide the boolean semantics (i.e., 0 or 1) of the signals on the circuit wires. Instead of letting the signals be 0 or 1 directly, random binary strings are used as the signals, and the strings are also the keys to some kind of encryption. Accordingly the truth table of an encrypted gate consists of encryptions of the string signals on the output wire, presented at random order, such that holding a pair of input signals one can decrypt exactly one entry in the truth table and obtain the correct output signal. For example, suppose the two input string signals to an OR gate correspond to the boolean values 0 and 1, respectively. Then, through decrypting one entry in the truth table, one will obtain the output signal corresponding to boolean value 1. Each wire in an encrypted circuit has a unique pair of string signals corresponding to the two possible boolean values. The correspondence, which we'll refer to as the *semantics* of the string signals, is hidden from the evaluator of the circuit. However, holding a set of input signals and even without knowing

their semantics, one is able to correctly propagate through the circuit with the help of the truth table at each gate. Meanwhile, without directly knowing the semantics of the string signals, there is no way for the evaluator to figure out the value of either the input or the output, because 1) the strings are randomly chosen and so is their correspondence to the boolean values; 2) the process of decrypting an entry in the truth table does not reveal any information on the semantics of the input strings and the output string; 3) the security of the encryption scheme ensures only one entry can be decrypted with a given pair of input strings. In the following, we use *signals* to refer to the random strings.

In the BMR construction, the signals are random binary strings whose lengths depend on the security parameter. For each wire, the semantics of the two signals on that wire is also randomly determined. We adopt the convention from [54] as to append a tag bit to a signal. So for the two signals of a wire, one has tag '0' and the other has tag '1'. We stress a signal's tag has no connection with its semantics. The only purpose of the tags is for locating the corresponding entry in the truth table. Suppose the security parameter is k. Then each signal is a k-bit string plus the tag bit. Furthermore, the BMR construction makes use of a pseudorandom generator G that stretches k bits to $2k+2$ bits. Let s be a k-bit seed, and define $G_0(s)$ and $G_1(s)$ by $G(s) = G_0(s)G_1(s)$, for $|G_0(s)| = |G_1(s)| = k+1$. In other words, $G_0(s)$ and $G_1(s)$ are the first and the second half of the pseudorandom string generated by G with seed s. For an encrypted gate g, let the left input wire be α, and let the two possible signals on α be $s_{\alpha 0}$ with tag bit '0' and $s_{\alpha 1}$ with tag bit '1'. The semantics of $s_{\alpha 0}$ and $s_{\alpha 1}$ are λ_α and $\overline{\lambda_\alpha}$, respectively, where $\lambda_\alpha \in \{0,1\}$. Similarly, define the right input wire as β, the two possible signals as $s_{\beta 0}$ and $s_{\beta 1}$, whose semantics are λ_β and $\overline{\lambda_\beta}$, respectively. The output wire is γ, with signals $s_{\gamma 0}$ and $s_{\gamma 1}$, and semantics λ_γ and $\overline{\lambda_\gamma}$. The function that g computes is denoted by $\otimes$, and $\oplus$ denotes bit-wise XOR. In addition, let $s'_{\alpha 0}$ denote the part of $s_{\alpha 0}$ without the tag bit, and likewise for the other signals. Then the truth table of g is defined

as follows.

$$
\begin{aligned}
A^g_{00} &= G_0(s'_{\alpha 0}) \oplus G_0(s'_{\beta 0}) \oplus \begin{cases} s_{\gamma 0} & \text{if } \lambda_\alpha \otimes \lambda_\beta = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A^g_{01} &= G_1(s'_{\alpha 0}) \oplus G_0(s'_{\beta 1}) \oplus \begin{cases} s_{\gamma 0} & \text{if } \lambda_\alpha \otimes \overline{\lambda_\beta} = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A^g_{10} &= G_0(s'_{\alpha 1}) \oplus G_1(s'_{\beta 0}) \oplus \begin{cases} s_{\gamma 0} & \text{if } \overline{\lambda_\alpha} \otimes \lambda_\beta = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A^g_{11} &= G_1(s'_{\alpha 1}) \oplus G_1(s'_{\beta 1}) \oplus \begin{cases} s_{\gamma 0} & \text{if } \overline{\lambda_\alpha} \otimes \overline{\lambda_\beta} = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise.} \end{cases}
\end{aligned}
\tag{2.4}
$$

An evaluator, holding input signals $s_{\alpha a}$ and $s_{\beta b}$ where $a, b \in \{0, 1\}$, computes the output signal as $G_b(s'_{\alpha a}) \oplus G_a(s'_{\beta b}) \oplus A^g_{ab}$. It's easy to verify that this evaluation produces the correct output signal and does not reveal any information about the semantics of all the signals involved[8]. This fact can be extended straightforwardly to evaluating the whole circuit. To recover the value of the output, one needs to know the semantics of the output signals. If the output can be made public, then the semantics of the output signals can simply be defined the same as their tags. Creating an encrypted circuit by a single party, as is the case for the agent protocol, is just a matter of choosing random strings as the signals, defining the semantics, and computing the truth tables for each gate.

2.4.2 The Security Flaw of The BMR Construction

The BMR construction has a clean and simple design (Equation (2.4)) which leads to efficient implementation and clear analysis of its security. Unfortunately, for circuits with certain topologies, the security may be compromised. Specifically, any circuit that contains gates with fan-out greater than one may be vulnerable to attacks, as illustrated by the following example.

[8]Looking ahead, we remark that this original construction has a flaw that could, in some cases, let the evaluator find out the semantics of the signals.

Figure 2.1: A circuit that computes $z = \max(x_1, x_2)$.

Suppose two players Alice and Bob each hold ℓ-bit numbers, x_1 and x_2 respectively, and want to find out the larger number of the two. After joint computation, the two players should learn the larger number, but no information about the smaller number should be revealed. Hence, for the player that has the larger number, he should essentially gain no new information after the computation. This is a typical application of Yao's two party SFE protocol (introduced in the Section 1.3.2 of Chapter 1). Assume Alice creates an encrypted circuit for comparing and outputting the larger one of two input numbers, using the BMR construction. The circuit is evaluated by Bob. Suppose $x_1 < x_2$. As a result, the output of the encrypted circuit is Bob's number and the assumed security prevents Bob from finding out anything Alice's number other than the fact that it's less than his own number. However, we show that even a semi-honest Bob is able to learn Alice's number x_1 with high probability, as long as x_1 has a nontrivial number of zero bits.

Figure 2.1 shows a circuit that computes $z = \max(x_1, x_2)$. Each rectangle represents a subcircuit. If $x_1 \geq x_2$, the comparator outputs 1, otherwise 0. This bit serves as a selector to the mask circuits M_1 and M_2, so that if the bit is 1, then $y_1 = x_1$ and $y_2 = 0$; otherwise, $y_1 = 0$ and $y_2 = x_2$. The "combiner" subcircuit is just a bitwise OR of y_1 and y_2. Let's focus on subcircuit M_1. M_1 simply ANDs each bit from x_1 with the selector bit (the output from the comparator),

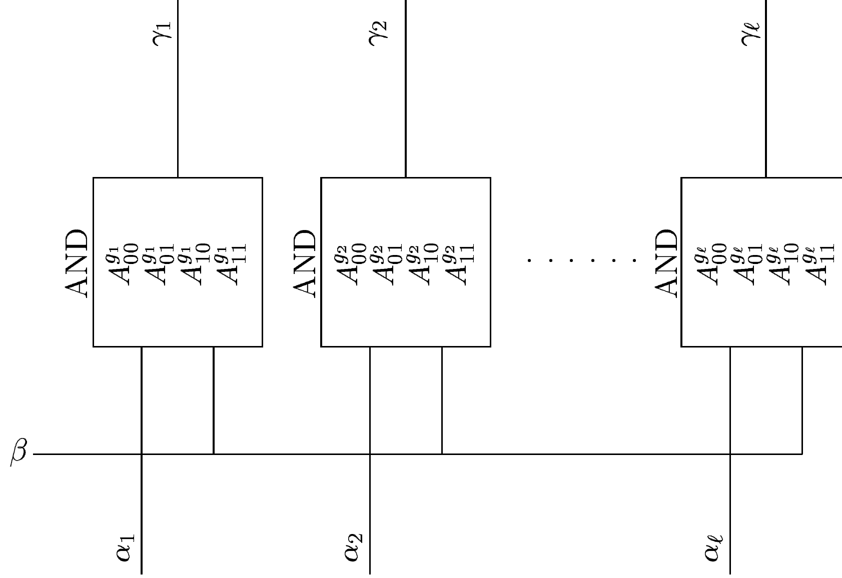

Figure 2.2: The encrypted subcircuit $\mathcal{M}_1$. Wires α_k get the input bits from x_1, and output wires γ_k provide the bits of y_1.

and the encrypted version of this subcircuit is shown in Figure 2.2. Suppose the input wires of M_1 are $\alpha_1, \ldots, \alpha_\ell$ for x_1 and β for the selector bit, the output wires are $\gamma_1, \ldots, \gamma_\ell$ for y_1.

According to equation (2.4), the truth table of gate g_i in $\mathcal{M}_1$, for $1 \le i \le \ell$, is

$$\begin{aligned}
A_{00}^{g_i} &= G_0(s'_{\alpha_i 0}) \oplus G_0(s'_{\beta 0}) \oplus \begin{cases} s_{\gamma_i 0} & \text{if } \lambda_{\alpha_i} \wedge \lambda_\beta = \lambda_{\gamma_i}, \\ s_{\gamma_i 1} & \text{otherwise;} \end{cases} \\
A_{01}^{g_i} &= G_1(s'_{\alpha_i 0}) \oplus G_0(s'_{\beta 1}) \oplus \begin{cases} s_{\gamma_i 0} & \text{if } \lambda_{\alpha_i} \wedge \overline{\lambda_\beta} = \lambda_{\gamma_i}, \\ s_{\gamma_i 1} & \text{otherwise;} \end{cases} \\
A_{10}^{g_i} &= G_0(s'_{\alpha_i 1}) \oplus G_1(s'_{\beta 0}) \oplus \begin{cases} s_{\gamma_i 0} & \text{if } \overline{\lambda_{\alpha_i}} \wedge \lambda_\beta = \lambda_{\gamma_i}, \\ s_{\gamma_i 1} & \text{otherwise;} \end{cases} \\
A_{11}^{g_i} &= G_1(s'_{\alpha_i 1}) \oplus G_1(s'_{\beta 1}) \oplus \begin{cases} s_{\gamma_i 0} & \text{if } \overline{\lambda_{\alpha_i}} \wedge \overline{\lambda_\beta} = \lambda_{\gamma_i}, \\ s_{\gamma_i 1} & \text{otherwise.} \end{cases}
\end{aligned} \tag{2.5}$$

Refer the pseudorandom strings generated by G as the "contribution" of the corresponding input

wire to the truth table. Therefore, the contribution of wire β to the truth tables of all the gates $g_1, \ldots, g_\ell$ comes from a limited set of four possible values consisting of $G_0(s'_{\beta 0})$, $G_0(s'_{\beta 1})$, $G_1(s'_{\beta 0})$, and $G_1(s'_{\beta 1})$. This can be exploited by an adversary as follows.

Let b_β denote the plaintext bit on wire β, that is, the semantics of the signal on β. Assume $x_1 < x_2$, so that $b_\beta = 0$ which in turn makes all the output wires of M_1 0 and so masks out x_1. However, this does not prevent the evaluator Bob from learning x_1. Essentially Bob is able to deduce what the output of M_1 would be if b_β were 1, from analyzing its encrypted version $\mathcal{M}_1$. Let σ_{α_i} denote the signal on input wire α_i. In the following, we play as Bob. First, observe that these α signals do not change when b_β changes from 0 to 1. So we can locate the correct truth table entry at each gate when b_β is 1, by simply flipping the second bit of the index to the truth table when $b_\beta = 0$ (which we know legally from the evaluation of the circuit). Second, since the signals on the α wires is known, we can compute their contributions to the output value of each gate after β is changed. (Again, this requires only flipping the pseudorandom generator from G_0 to G_1 or vice versa.) Next, we make a (likely wrong) guess that all bits of x_1 are 0. If this were true, evaluating the gates with $b_\beta = 1$ would give the same result as when $b_\beta = 0$, so we in fact already know the output signals because we computed them when the circuit was evaluated with the legally obtained inputs, x_1 and x_2 in the form of signals (which led to $b_\beta = 0$). Let σ_{γ_i}, for $1 \leq i \leq \ell$, denote the output signal of gate g_i when $b_\beta = 0$. Therefore, of the four basic parts of Equation (2.5) we know what three of them would be when $b_\beta = 1$: the proper truth table entry, the contribution of the input σ_{α_i}, and the output signal σ_{γ_i}. Without loss of generality, suppose that $\lambda_\beta = 1$, then the tag of the signal on β is 0 when $b_\beta = 1$. For gate g_i,

- If the tag of the signal on wire α_i is also 0, (in other words, the signal is $s_{\alpha_i 0}$,) then $A^{g_i}_{00}$ is the correct truth table entry to use when $b_\beta = 1$. Compute $\mu_i = \sigma_{\gamma_i} \oplus G_0(s'_{\alpha_i 0}) \oplus A^{g_i}_{00}$. If our guess was right, which means the ith bit of x_1 is actually 0, (not to be confused

with the tag 0) then σ_{γ_i} is the correct output signal, and so according to equation (2.5) $A^{g_i}_{00} = G_0(s'_{\alpha_i 0}) \oplus G_0(s'_{\beta 0}) \oplus \sigma_{\gamma_i}$. Thus $\mu_i = G_0(s'_{\beta 0})$. Otherwise (if our guess was not correct), then following the equations we see that $\mu_i = s_{\gamma_i 0} \oplus s_{\gamma_i 1} \oplus G_0(s'_{\beta 0})$, and since the signals are chosen randomly and independently, μ_i is a random value in this case.

- If the signal on α_i is $s_{\alpha_i 1}$, compute $\mu_i = \sigma_{\gamma_i} \oplus G_0(s'_{\alpha_i 1}) \oplus A^{g_i}_{10}$. If our guess was right, then $\mu_i = G_1(s'_{\beta 0})$. Otherwise, like before it is just a random string.

Therefore, all the resulting μ_i's can be divided into 3 groups. In each of the first 2 groups, the μ_i's are equal to each other, since they are equal to either $G_0(s'_{\beta 0})$ or $G_1(s'_{\beta 0})$. These μ_i's correspond to correct guesses of the output bit, or in other words the 0 bits of x_1. Note that we have no way of knowing these values ahead of time, since we do not know the proper signal on β for boolean value 1, but we *can* recognize subsets with equal μ_i values. The third group will be just some random strings, corresponding to the wrong guesses or in other word the 1 bits of x_1, and the probability that these values are equal to the values produced by the first two sets, or any other μ_i value, is very small. Thus by identifying repeated values produced by this computation we have determined x_1 with high probability. We summarize this attack in the following theorem.

Theorem 2.4.1 *Let $\mathcal{M}$ be the encrypted version of the mask circuit as in the previous discussion that correctly masks out an ℓ-bit value x that has z zero bits, for $z \geq 3$. If the wire signals each has k bits, then the attack described above correctly recovers x with probability at least $1 - \frac{z}{2^{z-1}} - \varepsilon$, for any constant $\varepsilon > 0$ and sufficiently large k.*

Proof: Consider the cases where this attack could fail. There could be two types of failure — either the adversary misses identifying a 0 bit in x or it mistakes a 1 bit in x as a 0 bit. For the first case to happen, it must be the case that of the α wires in our example corresponding to the 0 bits, all except one have the same semantic definition for their signals. As a result, the signals for all the 0

bits have the same tag except the one with different definition. Hence, the μ_i of that special "0" signal won't equal to the μ_i's of any other "0" signals, and so that bit cannot be identified. Since the semantic definition of each wire is chosen randomly and independently, the probability for $z-1$ wires to have the same semantics definition is $2 \cdot \binom{z}{z-1} \cdot (\frac{1}{2})^{(z-1)} \cdot \frac{1}{2} = \frac{z}{2^{z-1}}$.

For the adversary to misinterpret a 1 bit in x as a 0 bit, either the μ_i of that "1" signal happens to equal to the μ_i of the group of "0" signals with the same tag, or there are at least two "1" signals which have the same tag and the same μ_i's. Let z_0 and z_1 denote the number of 0 bits in x whose signals' tags are 0 and 1 respectively. So $z_0 + z_1 = z$. Similarly, let $\bar{z}_0$ and $\bar{z}_1$ denote the number of 1 bits with tags 0 and 1 respectively. An equivalent way to consider these two cases of mistaking a 1 bit as a 0 bit is that there is a conflict within the $\bar{z}_0$ μ_i values corresponding to the $\bar{z}_0$ 1 bits plus the common μ_i value of the z_0 0 bits, or a conflict within the $\bar{z}_1$ μ_i's of the $\bar{z}_1$ 1 bits plus the common μ_i value of the z_1 0 bits. Observe that for any "1 bit" wire α_i, its μ_i is the result of XORing together the two possible signals on the output wire and three pseudorandom strings. A wire signal is a truly random string. It follows that for a "1 bit" wire α_i, the corresponding μ_i is a truly random string, too. The probability of a conflict in the $(\bar{z}_0 + 1)$ μ_i's is $1 - (\frac{2^k-1}{2^k})(\frac{2^k-2}{2^k}) \cdots (\frac{2^k-\bar{z}_0}{2^k}) < 1 - (\frac{2^k-\bar{z}_0}{2^k})^{\bar{z}_0}$. Similarly, the probability of a conflict in the $(\bar{z}_1 + 1)$ μ_i's is less than $1 - (\frac{2^k-\bar{z}_1}{2^k})^{\bar{z}_1}$. For sufficiently large k, the sum of these two probabilities can be upper bounded by any constant $\varepsilon > 0$.

Combining all these cases and assuming sufficiently large k, the probability for a failed attack is less than $\frac{z}{2^{z-1}} + \varepsilon$. Hence, the probability for a successful attack is at least $1 - \frac{z}{2^{z-1}} - \varepsilon$ for sufficiently large k. ■

2.4.3 Using Splitters to Correct The Problem

While our example attacks specific masking circuits, it's obvious that the same strategy could apply to many other situations where multiple gates share a common input. The source of this security

flaw is the correlation in the entries of the truth tables of any gates that share a common input wire. Hence, the correction is straightforward: wherever a wire drives two or more gates, we split it into multiple wires each having its own signals and semantic definition. Each of these new spawned wire is fed into a gate that previously shared the "root" wire with other gates. As a result, no two gates will share an input wire. For this purpose, we add a "splitter" gate which has a single input and two outputs. The two outputs will have independent pairs of signals, effectively decorrelating the gates which use these output signals. We build up trees of splitters to accommodate arbitrary fanout.

Logically, a splitter simply copies the value from the input to each output. In the encrypted version, the truth table has four entries just as other gates. Let the input wire be α, the two output wires be γ and δ. Then the truth table of a "splitter" g is defined as follows:

$$\begin{aligned}
A^g_{00} &= G_0(s'_{\alpha 0}) \oplus \begin{cases} s_{\gamma 0} & \text{if } \lambda_\alpha = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A^g_{01} &= G_0(s'_{\alpha 1}) \oplus \begin{cases} s_{\gamma 0} & \text{if } \overline{\lambda_\alpha} = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A^g_{10} &= G_1(s'_{\alpha 0}) \oplus \begin{cases} s_{\delta 0} & \text{if } \lambda_\alpha = \lambda_\delta, \\ s_{\delta 1} & \text{otherwise;} \end{cases} \\
A^g_{11} &= G_1(s'_{\alpha 1}) \oplus \begin{cases} s_{\delta 0} & \text{if } \overline{\lambda_\alpha} = \lambda_\delta, \\ s_{\delta 1} & \text{otherwise.} \end{cases}
\end{aligned} \tag{2.6}$$

Then in the evaluation phase, for input signal $s_{\alpha a}$ where $a \in \{0, 1\}$, the two output signals are calculated as follows:

$$\sigma_\gamma = G_0(s'_{\alpha a}) \oplus A^g_{0a}$$

and

$$\sigma_\delta = G_1(s'_{\alpha a}) \oplus A^g_{1a}$$

It is not difficult to see that this properly reflects the logic of splitting the input wire, and results in two independent codings of the input value as signals on the output wires. We refer to the BMR construction with splitters added to break all shared wires as the *enhanced BMR construction*.

Lemma 2.4.2 *There are no two gates in an encrypted circuit constructed according to the enhanced BMR construction that share a common input wire.*

2.4.4 Proving Security for The Enhanced BMR Construction

With the correction described in the last section, we now have a provably secure construction of encrypted circuits. This section proves the security formally. To our knowledge, this is the only correct security proof of an encrypted circuit construction currently available, and it can be used generally in any protocol that utilizes encrypted circuits. Notice that encrypted circuits are always used as a component in a larger protocol, but we only concentrate on the security property provided by the encrypted circuit, which is only concerned with the evaluation of the circuit. Thus we assume an encrypted circuit created according to the enhance BMR construction and exactly one set of input signals are obtained by the evaluator. Depending on the larger protocol, the evaluator may be entitled to learn the output or part of it. In such cases, it is assumed that the evaluator has also obtained the semantic definitions of the output wires that he is entitled to but nothing else. These assumptions should be guaranteed by the outer protocol. Our goal is to show that under these assumptions, evaluating a *polynomial-size* encrypted circuit does not reveal any information to a polynomial-time bounded evaluator about the input and the output, or if the output is made public, then the information revealed is restricted to whatever can be deduced from the output. Following the standard technique, this is demonstrated by showing that the evaluator, with only

the knowledge he is entitled to after a particular evaluation, is able to create a *fake* circuit which is computationally distinguishable from the real encrypted circuit. Hence, the information revealed by the real circuit is no more than the information that is embodied in the fake circuit.

Our proof is extracted from the proof presented in [54]. The original proof is concerned with a multi-party SFE protocol. We extract out the part of the proof that is relevant to the encrypted circuit evaluation, and make changes to accommodate the splitters[9].

2.4.4.1 Fake Encrypted Circuits

The evaluator of a polynomial-size encrypted circuit can create a fake encrypted circuit and a set of fake input signals with only the knowledge of the "plaintext" circuit and (the part of) the output he is entitled to learn from the given evaluation. Like in a real encrypted circuit, random signals with tags are assigned to all wires, but only one signal per wire instead of two. If (part of) the output is known, define the semantics of the signals assigned to the corresponding output wires to match the known output. All non-output signals have unknown semantics (that is, they are not defined). For any gate, the truth table is constructed by finding the one entry that corresponds to the tags of the input signals to this gate[10] and correctly setting that entry to produce the already assigned output signal. In other words, the entry is the result of XORing altogether the pseudorandom strings generated from the two input signals as well as the output signal. The other three entries are set to random values. The correct gate entries are called the *on-path* entries, and other than making a consistent path through the circuit they have no real meaning (notice that the definition does not even depend on the functionality of the gate or the semantics of the signals). Slightly abusing the term, we use "circuit" to refer to its gates plus an instance of input signals. It is clear that this fake circuit reveals no information about private inputs or intermediate values of the

[9] The original proof is in fact invalid with the flawed construction. Although the proving strategy is correct, the flawed construction does not satisfy the claims made by the proof.

[10] Since the tags are randomly chosen, the gate entry selected will also be random.

computation (since none exist!). Next, we show that the ensemble of the distributions of these fake encrypted circuits, induced by the random choices of signals and indexed by the security parameter, the plaintext circuit, and the output known to the evaluator, is computationally indistinguishable from the ensemble of real encrypted circuits, as long as the circuit size is polynomially bounded.

Theorem 2.4.3 *Real encrypted circuits constructed according to the enhanced BMR construction are computationally indistinguishable from fake encrypted circuits described above by an polynomial-time evaluator, who knows signals for at most one instance of input and the semantics of the signals for the output it is entitled to learn (if any).*

2.4.4.2 Proving Indistinguishability

Denote the ensembles of fake encrypted circuits and the real encrypted circuits as $\tilde{\mathcal{C}}$ and $\mathcal{C}$ respectively. To prove their indistinguishability, first notice the signals (input, output, and internal) and the semantic definitions in both circuits are identically distributed, as they are all truly random. Furthermore, both circuits produce the same output value to the evaluator in the case that the evaluator learns part of or the whole output. The only difference is the truth tables of the gates. We make a sequence of hybrid encrypted circuit distributions as

$$\tilde{\mathcal{C}} = \mathcal{C}_0 \Rightarrow \mathcal{C}_1 \Rightarrow \cdots \Rightarrow \mathcal{C}_{\Gamma-1} \Rightarrow \mathcal{C}_\Gamma = \mathcal{C}.$$

where Γ is the total number of gates in the circuit. Because the size of the circuit is polynomially bounded, so is Γ. Suppose the gates are ordered according to a valid topological sorting, so the gates providing input always precede the gates receiving that input. For $\mathcal{C}_i$, gates numbered $1, \ldots, \Gamma - i$ are fake gates (only the one on-path entry in the truth table is correctly computed from pseudorandom strings, while the remaining entries are truly random strings), and the remaining gates are real

encrypted gates (all 4 entries are correctly computed[11]). It is obvious that at the endpoints of this sequence, $\mathcal{C}_0 = \tilde{\mathcal{C}}$ and $\mathcal{C}_\Gamma = \mathcal{C}$.

Assume for the sake of contradiction that $\tilde{\mathcal{C}}$ and $\mathcal{C}$ are distinguishable by a probabilistic polynomial-time algorithm $\mathcal{D}$. Since Γ is bounded by a polynomial, it immediately follows that somewhere in the above sequence there exists a non-negligible "jump" in the distinguishing probabilities, and so $\mathcal{D}$ is also able to distinguish the neighboring hybrid circuits. Intuitively, we can think of the distinguishing probability of $\tilde{\mathcal{C}}$ and $\mathcal{C}$ as the "sum" of the distinguishing probabilities of the neighboring circuits. Since there are only Γ pairs of neighboring circuits and Γ is polynomially bounded, without a non-negligible "jump", the distinguish probability of the two ends would not be non-negligible, either. So we can make use of this distinguisher to construct another polynomial-time distinguisher which is able to distinguish a truly random string from a pseudorandom string. This contradicts the assumption that the pseudorandom generator G is secure.

Definition 2.4.4 *We define two probability distributions:*

- U_{2k+2} *is the uniform distribution on length* $2k+2$ *binary strings;*
- $G(U_k)$ *is the distribution of length* $2k+2$ *binary strings that are produced by pseudorandom generator* G *on uniformly selected seeds of length* k*.*

Based on the definition of pseudorandom generator [35] and Theorem 2.1.4, we immediately get the following lemma.

Lemma 2.4.5 *Assuming the existence of a secure pseudorandom generator, the ensembles of distributions* U_{2k+2} *and* $G(U_k)$*, indexed by* k*, are computationally indistinguishable, even by repeated sampling.*

[11] This, of course, requires defining two signals and their semantics for each wire involved.

Consider two independent $(2k+2)$-bit strings, X and Y, which either both come from U_{2k+2} or both come from $G(U_k)$. Our goal is to create a distinguisher for the ensembles of these distributions. Define four substrings $X_0 = X[1 : k+1]$, $X_1 = X[k+2 : 2k+2]$, $Y_0 = Y[1 : k+1]$, and $Y_1 = Y[k+2 : 2k+2]$ — note that if X and Y are pseudorandom these correspond to precisely the pseudorandom substrings produced by the G_0 and G_1 functions, which were defined earlier.

For $i \in [1..\Gamma]$ and let $j = \Gamma - i + 1$, we use X and Y to define a new circuit distribution $\mathcal{C}_i^{X,Y}$ which involves constructing as in $\mathcal{C}_i$ (in which gates $g_j, \ldots, g_\Gamma$ are real), but using strings X and Y for the contributions of the input wires to the truth table of gate g_j and other gates that share an input wire with g_j if any. Like before, suppose the two input wires to g_j are α and β, and the output wire is γ. Without loss of generality, suppose the on-path entry is $A_{00}^{g_j}$, which means the on-path input signals are $s_{\alpha 0}$ and $s_{\beta 0}$. We substitute the pseudorandom strings in the truth table due to the off-path signals of α and β with X_0, X_1, Y_0, Y_1 as follows.

$$
\begin{aligned}
A_{00}^{g_j} &= G_0(s'_{\alpha 0}) \oplus G_0(s'_{\beta 0}) &&\oplus \begin{cases} s_{\gamma 0} & \text{if } \lambda_\alpha \otimes \lambda_\beta = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A_{01}^{g_j} &= G_1(s'_{\alpha 0}) \oplus Y_0 &&\oplus \begin{cases} s_{\gamma 0} & \text{if } \lambda_\alpha \otimes \overline{\lambda_\beta} = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A_{10}^{g_j} &= X_0 \oplus G_1(s'_{\beta 0}) &&\oplus \begin{cases} s_{\gamma 0} & \text{if } \overline{\lambda_\alpha} \otimes \lambda_\beta = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise;} \end{cases} \\
A_{11}^{g_j} &= X_1 \oplus Y_1 &&\oplus \begin{cases} s_{\gamma 0} & \text{if } \overline{\lambda_\alpha} \otimes \overline{\lambda_\beta} = \lambda_\gamma, \\ s_{\gamma 1} & \text{otherwise.} \end{cases}
\end{aligned} \tag{2.7}
$$

In summary, we substitute the pseudorandom strings $G_0(s'_{\alpha 1})$, $G_1(s'_{\alpha 1})$, $G_0(s'_{\beta 1})$, $G_1(s'_{\beta 1})$ with X_0, X_1, Y_0, Y_1. If X and Y are $G(U_k)$-distributed, then off-path entries in the truth table of gate g_j are made from pseudorandom strings just as they would be in a real gate. On the other hand, if

X and Y are U_{2k+2}-distributed then each of the off-path entries is made by XORing a truly random string from X or Y with some other strings, so the off-path entries should be random just as they would be in a fake gate. This leads to the following two claims.

Claim 2.4.6 *If X and Y are $G(U_k)$-distributed, then $\mathcal{C}_i^{X,Y} = \mathcal{C}_i$, for any $i \in [1..\Gamma]$.*

Claim 2.4.7 *If X and Y are U_{2k+2}-distributed, then $\mathcal{C}_i^{X,Y} = \mathcal{C}_{i-1}$, for any $i \in [1..\Gamma]$.*

Proof: It's obvious that gate g_j where $j = \Gamma - i + 1$ is a real gate if X and Y are $G(U_k)$-distributed, and a fake gate if X and Y are U_{2k+2}-distributed. Furthermore, no other gate in the circuit needs to be changed. If X and Y are pseudorandom, the substitution means the off-path signals on α and β are changed (to the seeds that generate X and Y, respectively). In a real encrypted circuit, this incurs two types of corresponding update in other gates.

First, there should be changes in the two gates that output α and β respectively. For the gate that outputs α, the entry or entries in the truth table that produce the off-path signal $s_{\alpha 1}$ should be modified so that they produce the seed of X plus tag 1. The same change is required for the gate that outputs β. However, in the hybrid circuit $\mathcal{C}_i$, all gates preceding gate g_j in the topological sort order are fake gates. So the off-path entries in the truth table of these gates are simply random strings with no connection with the output signals they are supposed to produce. Therefore, no change is required for the gates that precede g_j. Second, if there are any other gates that also use α or β as input wire, then the truth tables of those gates should also be modified. According to Lemma 2.4.2, there is no wire sharing in the whole circuit. So no update is needed for this reason, either. Thus, according to the definition of hybrid circuits, Claims 2.4.6 and 2.4.7 are correct[12]. ■

Now we construct another probabilistic polynomial-time algorithm $\mathcal{D}'$ based on $\mathcal{D}$ to distinguish samples taken from U_{2k+2} and $G(U_k)$. All the information needed for constructing a fake circuit

[12]In contrast, for the original BMR construction, updating other gates is required due to these two reasons, and the updating makes the two claims invalid. See our paper [62].

(i.e., the "plaintext" circuit, the output known to the evaluator, and the security parameter) is supplied to $\mathcal{D}'$ as an auxiliary input. $\mathcal{D}'$ randomly picks $i \in [1..\Gamma]$ and constructs hybrid circuit $\mathcal{C}_i^{X,Y}$. Then it outputs the output of $\mathcal{D}$ on $\mathcal{C}_i^{X,Y}$.

Lemma 2.4.8 *If ensembles $\tilde{\mathcal{C}}$ and $\mathcal{C}$ are distinguishable by a probabilistic polynomial-time algorithm $\mathcal{D}$, then the ensembles of distributions U_{2k+2} and $G(U_k)$ are distinguishable by probabilistic polynomial-time algorithm $\mathcal{D}'$.*

Proof: By definition, if $\mathcal{D}$ can distinguish $\tilde{\mathcal{C}}$ and $\mathcal{C}$, then there exists a polynomial $p(\cdot)$ such that, for infinitely many n's where n is the index of the ensembles, it holds that

$$\triangle(n) \stackrel{\text{def}}{=} |\Pr[D(\tilde{\mathcal{C}^n}, 1^n) = 1] - \Pr[D(\mathcal{C}^n, 1^n) = 1]| > \frac{1}{p(n)}$$

Slightly abusing the terms, let U_{2k+2} and $G(U_k)$ also denote the ensembles of the corresponding distributions, indexed by k which equals the security parameter encoded in n. Then we have

$$\Pr[D'(U_{2k+2}, 1^k) = 1] = \frac{1}{\Gamma}\sum_{i=1}^{\Gamma} \Pr[D(\mathcal{C}_{i-1}^n, 1^n) = 1]$$

and

$$\Pr[D'(G(U_k), 1^k) = 1] = \frac{1}{\Gamma}\sum_{i=1}^{\Gamma} \Pr[D(\mathcal{C}_i^n, 1^n) = 1]$$

Therefore,

$$\begin{aligned}
|\Pr[D'(U_{2k+2}, 1^k) = 1] - \Pr[D'(G(U_k), 1^k) = 1]| &= \frac{1}{\Gamma} \cdot |\Pr[D(\mathcal{C}_0^n, 1^n) = 1] - \Pr[D(\mathcal{C}_\Gamma^n, 1^n) = 1]| \\
&= \frac{1}{\Gamma} \cdot |\Pr[D(\tilde{\mathcal{C}^n}, 1^n) = 1] - \Pr[D(\mathcal{C}^n, 1^n) = 1]| \\
&= \frac{\triangle(n)}{\Gamma} > \frac{1}{\Gamma \cdot p(n)}
\end{aligned}$$

Hence, $\mathcal{D}'$ distinguishes U_{2k+2} and $G(U_k)$. ■

Lemma 2.4.5 and Lemma 2.4.8 together lead to Theorem 2.4.3.

2.5 Basic Tool 2: (Concurrent) Oblivious Threshold Decryption

Oblivious Threshold Decryption (OTD) combines the functionalities of threshold decryption with oblivious transfer. This new cryptographic primitive plays an essential role in our efforts to remove the trusted third party required by the prior work. This section presents a universally composable protocol for OTD whose instances can be executed concurrently.

2.5.1 1-out-of-2 Oblivious Transfer

Through 1-out-of-2 oblivious transfer, the sender Alice transfers exactly one out of the two values she possesses to the receiver Bob, such that Alice does not learn which value Bob receives and Bob does not learn anything about the other value. This functionality is captured by the following definition of ideal functionality $\mathcal{F}_{\mathrm{OT}}$.

Definition 2.5.1 *Ideal Functionality* $\mathcal{F}_{\mathrm{OT}}$[13]

$\mathcal{F}_{\mathrm{OT}}$ *proceeds as follows, running with an oblivious transfer sender* T*, a receiver* R *and an adversary* $\mathcal{S}$.

1. *Upon receiving a message* $(\text{Sender}, sid, x_1, x_2)$ *from* T*, where each* $x_i \in \{0,1\}^n$ *and sid is the session identifier, record the pair* (x_1, x_2).

2. *Upon receiving a message* $(\text{Receiver}, sid, i)$ *from* R*, where* $i \in \{0,1\}$*, send* (sid, x_i) *to* R *and* (sid) *to* $\mathcal{S}$*, and halt. (If no* $(\text{sender}, \ldots)$ *message was previously sent, then send nothing to* R*.)*

[13] Adjusted from a more general definition in [24].

For realizing 1-out-of-2 OT in the real model, the basic idea is to transfer the two n-bit strings, where n could be arbitrary polynomially-bounded number, through an "OT-channel" [13]. This is done in two stages. First, a shorter ℓ-bit seed, where $n \leq p(\ell)$ for some polynomial $p(\cdot)$, is transfered to the receiver through ℓ instances of bit-wise oblivious transfer. Then both n-bit strings are sent to the receiver in an encrypted form, and the seed received in the previous stage is fed into a pseudorandom generator to obtain the key for decrypting one of the two n-bit strings.

Protocol 2.5.2 *Bit-Wise OT*

(The following bit-wise 1-out-of-2 OT is taken from [24], which in turn is due to [36, 34].)

1. *Given input* $(\text{Sender}, sid, x_0, x_1)$, *where* $x_0, x_1 \in \{0,1\}$, *the sender* T *chooses a trapdoor permutation* f *over* $\{0,1\}^k$, *for security parameter* k, *together with its inverse* f^{-1}, *and sends* (sid, f) *to the receiver* R. *(The permutation* f *is chosen uniformly from a given family of trapdoor permutations.)*

2. *Given input* $(\text{Receiver}, sid, i)$ *where* $i \in \{0,1\}$, *and having received* (sid, f) *from* T, *receiver* R *randomly chooses* $y_{1-i}, r \in \{0,1\}^k$, *computes* $y_i = f(r)$, *and sends* (sid, y_0, y_1) *to* T.

3. *Having received* (sid, y_0, y_1) *from* R, T *sends* $(sid, x_0 \oplus B(f^{-1}(y_0)), x_1 \oplus B(f^{-1}(y_1)))$ *to* R, *where* $B(\cdot)$ *is a hard-core predicate for* f.

4. *Having received* (sid, b_0, b_1) *from* T, R *outputs* $(sid, b_i \oplus B(r))$.

The following lemma about the security of Protocol 2.5.2 is proved in [24].

Lemma 2.5.3 *Assuming that* f *is a trapdoor permutation, Protocol 2.5.2 securely realizes* $\mathcal{F}_{\text{OT}}$ *for the case of* $n = 1$ *and in the presence of static, semi-honest adversaries.*

Protocol 2.5.4 *String OT*

To transfer an n*-bit string through OT,*

1. *Given input* (Sender, sid, x_0, x_1)*, the sender T randomly chooses two ℓ-bit strings s_0 and s_1, where $n \leq p(\ell)$ for some polynomial $p(\cdot)$.*

2. *Given input* (Receiver, sid, i) *and through ℓ (concurrent) copies of bit-wise OT (Protocol 2.5.2), the receiver R receives s_i.*

3. *T computes $y_0 = G(s_0) \oplus x_0$ and $y_1 = G(s_1) \oplus x_1$, where G is a pseudorandom generator that generates an n-bit string from an ℓ-bit seed. T sends (sid, y_0, y_1) to R.*

4. *R outputs $(sid, G(s_i) \oplus y_i)$.*

Lemma 2.5.5 *Assuming that f is a trapdoor permutation and G is a pseudorandom generator (which can be constructed based on an one-way permutation), Protocol 2.5.4 securely realizes $\mathcal{F}_{\text{OT}}$ for an arbitrary polynomially-bounded n and in the presence of static, semi-honest adversaries.*

Proof (Sketch): Due to the UC composition theorem 2.3.3 and Lemma 2.5.3, execution of Protocol 2.5.4 in the real model is computationally indistinguishable from the hybrid model where bit-wise OT is implemented by the ideal functionality $\mathcal{F}_{\text{OT}}$. Hence, what's left is to prove that the hybrid model with ideal bit-wise OT is indistinguishable from the ideal string OT by any environment.

For any static, semi-honest adversary $\mathcal{H}$ in the hybrid model, we construct a corresponding static, semi-honest adversary $\mathcal{S}$ in the ideal model. Basically, $\mathcal{S}$ runs a copy of $\mathcal{H}$ and forwards the communication between the environment and $\mathcal{H}$. Steps 1 and 2 of Protocol 2.5.4 are just calls to the ideal bit-wise OT functionality. If the sender T is the only party corrupted, no simulation is required for these two steps, as T receives nothing from the ideal functionality. If the receiver R is the only party corrupted, $\mathcal{S}$ randomly generates and feeds the internally-run $\mathcal{H}$ an ℓ-bit string s. For steps 3 and 4, if T is the only party corrupted, again no simulation is needed. If R is the

only party corrupted, $\mathcal{S}$, knowing R's input i and output x_i from the ideal string OT functionality, feeds $\mathcal{H}$ with $y'_i = G(s) \oplus x_i$ and $y'_{1-i} = r$, where r is a random n-bit string. y'_i is distributed identically as y_i in the hybrid model, while y'_{1-i} is computationally indistinguishable from y_{1-i} due to the security of the pseudorandom generator. Finally, if neither party is corrupted, in the hybrid model $\mathcal{H}$ only sees the messages sent in step 3. Thus, $\mathcal{S}$ simply generates two messages each with a random n-bit string. Again, they are indistinguishable from the real messages y_0 and y_1 due to the security of the pseudorandom generator. If both parties are corrupted, everything is controlled by $\mathcal{H}$ and no simulation is required. In summary, $\mathcal{S}$ is able to simulate all the information $\mathcal{H}$ may collect in the hybrid model. Hence, the internally-run $\mathcal{H}$ behaves exactly the same as the actual $\mathcal{H}$ in the hybrid model. We conclude that no environment can tell between the hybrid model execution and its simulation in the ideal model with non-negligible probability[14]. ■

2.5.2 Threshold Decryption

A threshold public-key cryptosystem [29] is a scheme whose encryption is the same as a conventional public-key encryption, but the decryption key is shared by a number of decryption servers and decryption cannot be successfully performed unless m (the threshold value) or more servers participate. To avoid unnecessary confusion, we still use "threshold decryption scheme" to refer to a threshold public-key cryptosystem. A threshold decryption scheme is *t-secure* if ciphertexts of different messages are indistinguishable to a coalition of up to t semi-honest servers. In other words, the decryption reveals no knowledge to any group of up to t servers. It is *t-robust* if the same is true for up to t malicious servers, plus the malicious servers cannot disrupt a legal decryption process. Obviously, $t < m$. What we are interested in is a universally composable, non-interactive threshold decryption. That is, the decryption process consists of only one round of communication between the client and the m servers. Unlike the other basic tools used in this work, to our knowledge

[14] More formal argument can be found in the proof of Claim 4.1 in [24].

currently there is no threshold decryption scheme that is proven to be universally composable. The best candidate is the scheme of Canetti and Goldwasser [22]. Proving the universal composability of the CG scheme or some other scheme is left as an open question for future research. In light of this lack of provable realization, we only present the ideal functionality of a threshold decryption scheme, and summarize the Canetti-Goldwasser scheme as a potential real-model realization without a proof.

In addition to the threshold properties, a threshold decryption scheme should meet the same security measures for conventional public-key encryptions, such as CCA (Chosen Ciphertext Attack) security, non-malleability, and plaintext awareness. Hence, the ideal functionality is defined accordingly as a trusted encryption/decryption server that also provides these security properties. Notice that the ideal adversary $\mathcal{S}$ is given enough power as to choosing the public key, the value of each ciphertext, and the output of decryptions on ciphertexts that were not legitimately generated by using the ideal functionality. The purpose is to ensure security of the legitimately encrypted plaintexts even with such a powerful adversary.

Definition 2.5.6 *Ideal Functionality* $\mathcal{F}_{\mathrm{TD}}$[15]

$\mathcal{F}_{\mathrm{TD}}$ proceeds as follows, parameterized with a threshold m, a security parameter k, and a message domain ensemble $\mathcal{D} = \{D_k\}_{k \in \mathrm{N}}$, and running with encryption user E, decryption user D, decryption servers $P_1, \ldots, P_n$ for $n \geq m$, and an adversary $\mathcal{S}$.

***Key Generation**: In the first activation, expect to receive a message* (KeyGen, sid) *from E. Then do:*

1. *Hand* (KeyGen, sid) *to the adversary.*
2. *Receive a value e from $\mathcal{S}$ and record e.*
3. *Hand e to E.*

[15] Definition based on a similar notion in [22] with slight improvement from [23].

***Encryption**: Upon receiving a message* (Encrypt, sid, e', w) *from E (and E only), proceed as follows:*

1. *If $w \notin D_k$ then return an error message to E.*
2. *If $w \in D_k$ then hand* (Encrypt, sid, e') *to $\mathcal{S}$. (If $e' \neq e$ or e is not yet defined then hand also the entire value w to $\mathcal{S}$.)*
3. *Receive a tag c from $\mathcal{S}$ and hand c to E. If $e' = e$ then record the pair (c, w). (If c already appears in a previously recorded pair then return an error message to E.)*

***Decryption**: Upon receiving a message* (Decrypt, sid, c) *from D, record c and proceed as follows:*

1. *Upon receiving a message* (Decrypt, sid, c') *from decryption server P_i and $c' = c$, record this request as* (Decrypt, c, P_i) *if there is no same request from P_i previously recorded for this session.*
2. *Once there are at least m requests with the same tag c as requested by D, and there is a recorded pair (c, w), then hand w to D.*
3. *If there are m or more requests, but there does not exist a recorded pair (c, w), hand* (Decrypt, sid, c) *to $\mathcal{S}$. When receiving a value w from $\mathcal{S}$, hand w to D.*

In a nutshell, the Canetti-Goldwasser threshold decryption scheme [22] is based on the public-key encryption scheme of Cramer and Shoup [28] and transforms the CS scheme into a distributed, threshold scheme. The CS scheme is secure against adaptive CCA. The transformation maintains this property and is geared toward security in a UC-like framework[16]. Unfortunately, the security of the CG scheme under the UC model has yet to be formally proved. Both schemes attained their security goals assuming the *Decisional Diffie-Hellman* (DDH) problem is intractable. The

[16]This was actually one of the motivations for the development of the UC paradigm [17].

DDH problem is: Given a tuple that is either of the form (g^x, g^y, g^{xy}) or (g^x, g^y, g^z), where g is the generator of a cyclic group G of prime order q and $x, y, z \in Z_q$ are random, determine which is the case. In the following we summarize the basic CG scheme that is (presumably) t-secure (i.e., secure against up to t semi-honest decryption servers) under the UC notion, but leave the proof of its security as an open question. The scheme requires $m = 2t + 1$ correct shares to complete a decryption operation. Furthermore, the public/private key pair have to be refreshed after a certain number of decryptions.

Algorithm 2.5.7 *The Canetti-Goldwasser Threshold Decryption Scheme*

Given security parameter k, let p be a k-bit prime, and g_1, g_2 the generators of a subgroup of Z_p of a large prime order q. H is a hash function chosen from a collision-resistant hash function family. Also, let L denote the number of decryptions after which the public/private key pair has to be refreshed, and let t be the maximum number of servers that the adversary can corrupt.

Key Generation: *Suppose a trusted dealer for this stage, which can be implemented by a multi-party SFE protocol by the decryption servers. Call a polynomial $P(\xi) = \sum_{i=0}^{d} a_i \xi^i (\text{mod } q)$ a* random degree d polynomial for *a if $a_0 = a$ and $a_1, \ldots, a_d \stackrel{R}{\leftarrow} Z_q$, where $\stackrel{R}{\leftarrow}$ means randomly choosing. The dealer generates:*

- *$x_1, x_2, y_1, y_2, z \stackrel{R}{\leftarrow} Z_q$, and random degree t polynomials $P^{x_1}(), P^{x_2}(), P^{y_1}(), P^{y_2}(), P^z()$ for x_1, x_2, y_1, y_2, z, respectively.*
- *L values of $s_1, \ldots, s_L \stackrel{R}{\leftarrow} Z_q$ and random degree t polynomials $P^{s_1}(), \ldots, P^{s_L}()$ for them.*
- *L random degree $2t$ polynomials $P^{o_1}(), \ldots, P^{o_L}()$ for the value 0.*

Let $x_{j,i} = P^{x_j}(i)$, and define $y_{j,i}, z_i, s_{l,i}, o_{l,i}$ similarly. The public key $\boldsymbol{pk} = (p, q, g_1, g_2, c, d, h)$ where $c = g_1^{x_1} g_2^{x_2}$, $d = g_1^{y_1} g_2^{y_2}$, and $h = g_1^z$. Decryption server P_i holds secret key share $\boldsymbol{sk}_i = (p, q, g_1, g_2, x_{1,i}, x_{2,i}, y_{1,i}, y_{2,i}, z_i, s_{1,i}, \ldots, s_{L.i}, o_{1,i}, \ldots, o_{L,i})$.

Encryption: *To encrypt a message w encoded as an element in Z_q, E chooses $r \stackrel{R}{\leftarrow} Z_q$, and*

computes $\text{ENC}_{\text{pk}}(w, r) = (g_1^r, g_2^r, h^r w,\ c^r d^{r\alpha})$*, where* $\alpha = H(g_1^r, g_2^r, h^r w)$*.*

Decryption: *To decrypt the* l*th ciphertext* (u_1, u_2, e, v)*,* $l \leq L$*, each server* P_i *first computes a share* v_i' *of* v*, as* $v_i' = u_1^{x_{1,i}+y_{1,i}\alpha} u_2^{x_{2,i}+y_{2,i}\alpha}$*. Then it computes a share* $u_1^{z_i}$ *of* u_1^z *and a share* $g_1^{o_{l,i}}$ *of value 1. Next* P_i *computes the partial decryption* $f_i = u_1^{z_i} \cdot (v/v_i')^{s_{l,i}} \cdot g_1^{o_{l,i}}$*. The user* D *collects (at least)* $m = 2t+1$ *partial decryptions* $f_0, \ldots, f_m$ *and computes* $f_0 = \prod_{i=1}^m f_i^{\lambda_i}$*, where the* λ_i*'s are the appropriate Lagrange interpolation coefficients that satisfy for any degree* $2t$ *polynomial* $P()$ *over* Z_q *it holds that* $P(0) = \sum_{i=1}^m \lambda_i P(i)$*.* D *recovers the message as* $w = e/f_0$*.*

The scheme intentionally avoids specifying how the user D "collects" the partial decryptions. This is left to the outer protocol. However, one obvious requirement is that the collecting process does not leak the shares to the adversary. For example, in the UC framework, the communication channels are public. Therefore, secure message transmission [20] must be employed if the servers directly send the shares to D.

2.5.3 Oblivious Threshold Decryption

Combining non-interactive threshold decryption and 1-out-of-2 oblivious transfer, we get oblivious threshold decryption (OTD) in which a user requests decrypting two ciphertexts but receives only one decryption, while the decryption servers do not learn which decryption the user actually receives. Again, since currently there is no provable realization of UC threshold decryption, we present only an "abstract" realization of OTD. In particular, we assume a non-interactive threshold decryption scheme TD which consists of four operations:

- Key Generation: This is done by a trusted dealer or through a multi-party protocol by the decryption servers. As a result, the public key is made known to the encryption user, and each server gets its share of the decryption key secretly.

- Encryption: The encryption user runs the encryption algorithm on the plaintext and the encryption key. This is a process involving only the encryption user.

- Decryption: A decryption server applies the decryption algorithm with its share of the decryption key on a ciphertext. The result is a decryption share. Obtaining a decryption share involves only the current server and requires no interaction with the decryption user or any other server.

- Share Combination: After collecting m decryption shares, the decryption user combines them using the combination algorithm and obtains the plaintext. We require no interaction with the servers in this stage.

Thus, if we leave the part of getting inputs — distributing the keys, obtaining a plaintext or a ciphertext, and collecting decryption shares — to the upper layer, then no communication is required to perform the core functionalities of the threshold decryption scheme. This is essential for the implementation of OTD. It's easy to verify that the Canetti-Goldwasser cryptosystem has all these properties.

Definition 2.5.8 *Ideal Functionality* $\mathcal{F}_{\mathrm{OTD}}$

Running with: encryption user E, decryption user D, decryption servers $P_1, \ldots, P_n$, adversary $\mathcal{S}$.

Parameters: threshold m, security parameter k, message domain ensemble $\mathcal{D} = \{D_k\}_{k \in \mathbf{N}}$.

Key Generation: *In the first activation, expect to receive a message* (KeyGen, sid) *from E. Then do:*

1. *Hand* (KeyGen, sid) *to the adversary.*

2. *Receive a value e from $\mathcal{S}$ and record e.*

3. *Hand e to E.*

Encryption: *Upon receiving a message* (Encrypt, sid, e', w) *from* E *(and* E *only), proceed as follows:*

1. *If* $w \notin D_k$ *then return an error message to* E.

2. *If* $w \in D_k$ *then hand* (Encrypt, sid, e') *to* $\mathcal{S}$. *(If* $e' \neq e$ *or* e *is not yet defined then hand also the entire value* w *to* $\mathcal{S}$.*)*

3. *Receive a tag* c *from* $\mathcal{S}$ *and hand* c *to* E. *If* $e' = e$ *then record the pair* (c, w). *(If* c *already appears in a previously recorded pair then return an error message to* E.*)*

Oblivious Decryption: *Upon receiving a message* (Decrypt, $sid, c1, c2, b, \mathcal{ID}$) *from* D*, where* $b \in \{0,1\}$ *and* $\mathcal{ID}$ *is a set identities of at least* m *decryption server, record* $(c1, c2, b, \mathcal{ID})$ *and proceed as follows:*

1. *Upon receiving a value* (Decrypt, sid, c'_1, c'_2) *from decryption server* P_i*, and* $c'_1 = c_1$*,* $c'_1 = c_1$*, and* $P_i \in \mathcal{ID}$*, record this request as* (Decrypt, $(c_1, c_2), P_i$) *if there is no same request from* P_i *previously recorded.*

2. *Once there are at least* m *requests from the servers in* $\mathcal{ID}$ *with the same pair* (c_1, c_2) *as requested by* D*, and there are recorded pairs* (c_1, w_1) *and* (c_2, w_2)*, hand* w_b *to* D.

3. *If there are* m *or more requests from the server and a corresponding request from* D*, but there does not exist either a recorded* (c_1, w_1) *or* (c_2, w_2)*, hand the value* (Decrypt, sid, c_1, c_2) *to* $\mathcal{S}$. *When receiving two value* w_1*,* w_2 *from* $\mathcal{S}$*, hand* w_b *to* D.

The key point of OTD is that the user receives exactly one plaintext from a pair of ciphertexts, and the decryption servers do not learn which that plaintext is. Meanwhile, the definition also maintains the standard features of a threshold decryption scheme. Notice we do not specify any

requirements on the pair of ciphertexts requested, this is left to the upper layer of the outer protocol. (For example, a decryption server can impose some rules on which ciphertext pairs it agrees to decrypt.)

Assuming the existence of a UC non-interactive threshold decryption scheme whose operations can be abstracted as described above, implementing $\mathcal{F}_{\mathrm{OTD}}$ is basically a combination of the threshold decryption scheme and string-wise OT, where the decryption shares are returned to the user through oblivious transfer. Due to the composition theorem 2.3.3, we first describe a hybrid protocol assuming the existence of ideal functionality $\mathcal{F}_{\mathrm{OT}}$.

Protocol 2.5.9 *Oblivious Threshold Decryption in The $\mathcal{F}_{\mathrm{OT}}$-Hybrid Model*

1. *Key Generation: run the* Key Generation *phase of TD.*
2. *Encryption: run the* Encryption *operation of TD.*
3. *Oblivious Decryption: the decryption user D receives a bit b, a pair of ciphertexts (c_0, c_1), and the identities of (at least) m decryption servers from the environment; while the same m servers receive the pair of ciphertexts (c_0, c_1). Each of the m servers P_i obtains its decryption shares (c_0^i, c_1^i) through the* Decryption *operation of TD. Then D and each P_i calls $\mathcal{F}_{\mathrm{OT}}$ such that D receives c_b^i. Applying* Share Combination *of TD, D combines the m shares it has received to obtain the plaintext of c_b.*

Lemma 2.5.10 *If non-interactive threshold decryption scheme TD securely realizes ideal functionality $\mathcal{F}_{\mathrm{TD}}$ in the presence of a static, semi-honest adversary that corrupts up to t decryption servers, then Protocol 2.5.9 securely realizes $\mathcal{F}_{\mathrm{OTD}}$ against a static, semi-honest adversary that corrupts up to t servers.*

Proof (Sketch): The goal is to prove for any environment, the ensembles $\mathrm{HYB}^{\mathcal{F}_{\mathrm{OT}}}_{\pi,\mathcal{H},\mathcal{Z}} \overset{c}{\approx} \mathrm{IDEAL}_{\mathcal{F}_{\mathrm{OTD}},\mathcal{S},\mathcal{Z}}$, where π refers to Protocol 2.5.9. Due to the assumption, there exists an ideal adversary $\mathcal{S}_{\mathrm{TD}}$ running

with $\mathcal{F}_{\mathrm{TD}}$ that simulates protocol TD. Since the Key Generation and Encryption of Protocol 2.5.9 are the same as TD, and their counterparts in the corresponding ideal functionalities, $\mathcal{F}_{\mathrm{TD}}$ and $\mathcal{F}_{\mathrm{OTD}}$ respectively, are also the same, $\mathcal{S}_{\mathrm{TD}}$ can be directly used to simulate these two operations. For Oblivious Decryption, first notice share collecting is done through ideal oblivious transfer. Hence, it does not reveal to the adversary, which has no access to the user D, any information about the shares from the honest servers. The difference from TD is each server decrypt two ciphertexts and the user only receives one plaintext. Because of the semantic security of TD, there is no distinguishable relation between the two ciphertexts even if their corresponding plaintexts are related. Thus, the simulation is basically run the corresponding part of $\mathcal{S}_{\mathrm{TD}}$ twice to simulate two separate ciphertexts. ■

Replacing each of the ideal-model oblivious transfers occurred in Protocol 2.5.9 with the real-model realization — Protocol 2.5.4, we get the real-model OTD protocol.

Protocol 2.5.11 *Oblivious Threshold Decryption in The Real Model*

Run Protocol 2.5.9 with the exception that whenever a call to $\mathcal{F}_{\mathrm{OT}}$ *is required, the involved parties execute Protocol 2.5.4.*

With Lemmas 2.5.10 and 2.5.5, applying the composition theorem (Theorem 2.3.3) leads to:

Theorem 2.5.12 *Protocol 2.5.11 securely realizes* $\mathcal{F}_{\mathrm{OTD}}$ *in the real model against a static, semi-honest adversary that corrupts up to* t *decryption servers, where* t *is the maximum number of corrupted servers the underlying non-interactive threshold decryption scheme can stand against.*

2.5.4 Concurrent OTD

The definition of OTD in the last section does not fully capture the requirements of the agent protocol. Specifically, multiple decryption operations are executed concurrently in the agent protocol. At first glance, this is easily dealt with by applying the UC composition theorem to compose

concurrently multiple OTD instances together. However, there is a subtle difference between the composition provided by the UC theorem and the one needed by the agent case. The UC composition theorem works at the ideal functionality level. That is, the theorem guarantees security when composing protocol instances each realizing a separate copy of the corresponding ideal functionality. Referring to Definition 2.5.8, this means the concurrent decryptions must each have a different set of encryption/decryption keys. Obviously, what's going on in the agent protocol is concurrent decryptions of ciphertexts encrypted by the same key[17].

Notice that Definition 2.5.8 does allow multiple encryptions and decryptions with the same keys. Hence, the issue lies mainly in the realization. We formalize this notion of concurrent decryptions with the same key as another ideal functionality.

Definition 2.5.13 *Ideal Functionality* $\mathcal{F}_{\mathrm{COTD}}$

$\mathcal{F}_{\mathrm{COTD}}$ *behaves exactly like* $\mathcal{F}_{\mathrm{OTD}}$*, but there could be multiple decryption users running with* $\mathcal{F}_{\mathrm{COTD}}$*.*

We are interested in a realization of $\mathcal{F}_{\mathrm{COTD}}$ where some or all of decryptions are done concurrently by possibly different groups of user+servers.

Protocol 2.5.14 *OTD with Concurrent Decryptions*

Run Protocol 2.5.11 with concurrent executions of Oblivious Decryption by the appropriate groups of parties.

Proof of the security of Protocol 2.5.14 can be sketched as follows. The Oblivious Decryption of Protocol 2.5.11 consists of three phases. The servers each independently obtain two decryption shares and obliviously transfer one share to the user, then the user combines the shares into the decryption result. The first and the last phases are both non-interactive, involving no communication, and therefore cause no problem when executed concurrently. The only interactive phase is the

[17]Different keys are possible but certainly incur undesired overhead.

oblivious transfer. Due to its universal composability, this phase is also concurrently executable.

Theorem 2.5.15 *Protocol 2.5.14 securely realizes ideal functionality $\mathcal{F}_{\mathrm{COTD}}$ in the real model against a static, semi-honest adversary that corrupts up to t decryption servers, where t is the maximum number of corrupted servers the underlying non-interactive threshold decryption scheme can stand against.*

2.6 Secure Mobile Agent Computation against Static, Semi-Honest Adversaries

In this section, we present a protocol for executing mobile agent applications with universally composable security when the adversary is static and semi-honest. We assume the agent application is implemented in a multi-agent setting with the restrictions discussed at the beginning of Section 2.2. For the sake of the security proof, like before we first present the protocol in a hybrid model where the participants have access to the ideal functionality of concurrent oblivious threshold decryption and public-key encryption (PKE) (to be introduced below). A proof of security for this hybrid version of the agent protocol follows. The real-model protocol is obtained by replacing the ideal functionality of OTD and PKE with their real-model realizations, and the security follows due to the composition theorem.

2.6.1 Basic Tool 3: Public-Key Encryption

The last primitive used in the agent protocol is public-key encryption (PKE). PKE is a well-studied cryptographic primitive, and as a result, there are ready-to-use PKE solutions [28, 30, 12, 48, 51]. This section summarizes the results that are used by our protocol, beginning with the definition of the ideal functionality for PKE.

Definition 2.6.1 *Ideal Functionality $\mathcal{F}_{\text{PKE}}$ [23]*[18]

$\mathcal{F}_{\text{PKE}}$ proceeds as follows, when parameterized by message domain ensemble $\mathcal{D} = \{D_k\}_{k\in \mathrm{N}}$ and security parameter k, and interacting with an adversary $\mathcal{S}$, and parties $P_1, \ldots, P_n$.

Key Generation: *In the first activation, expect to receive a message* (KeyGen, sid) *from some party P_i. Then do:*

1. *Hand* (KeyGen, sid) *to the adversary.*

2. *Receive a value e from $\mathcal{S}$ and record e and the identity of P_i.*

3. *Hand e to P_i.*

Encryption: *Upon receiving a message* $(\text{Encrypt}, sid, e', m)$ *from some party P_j, proceed as follows:*

1. *If $m \notin D_k$ then return an error message to P_j.*

2. *If $m \in D_k$ then hand* $(\text{Encrypt}, sid, e', P_j)$ *to $\mathcal{S}$. (If $e' \neq e$ or e is not yet defined then hand also the entire value m to $\mathcal{S}$.)*

3. *Receive a tag c from $\mathcal{S}$ and hand c to P_j. If $e' = e$ then record the pair (c, m). (If c already appears in a previously recorded pair then return an error message to P_j.)*

Decryption: *Upon receiving a message* $(\text{Decrypt}, sid, c)$ *from P_i, the party which initiated Key Generation, (and P_i only), proceed as follows:*

1. *If there is a recorded pair (c, m) then hand m to P_i.*

2. *Otherwise, hand the value* $(\text{Decrypt}, sid, c)$ *to the adversary. When receiving a value m from the adversary, hand m to P_i.*

[18] With slight modification from the original definition.

Canetti et. al. [23] showed that the above definition of ideal functionality for PKE is equivalent to CCA security when the adversary is non-adaptive. As an important corollary, a CCA-secure PKE scheme [28, 30] can be trivially turned into a realization of $\mathcal{F}_{\mathrm{PKE}}$, with the three operations of the PKE scheme (Key Generation, Encryption, and Decryption) each mapped to its counterpart in $\mathcal{F}_{\mathrm{PKE}}$.

2.6.2 Secure Mobile Agent Protocol in The Hybrid Model

Some notations are in order. Host identities and the agent functions are defined in Definitions 2.2.1 and 2.2.2. Without loss of generality, assume the agent state x is n_x bits, the host input y is always n_y bits, and the output to the host is n_z bits. The agent code is a collection of encrypted circuits that compute the agent functions, each to be evaluated at a host the agent visits. The encrypted circuits are each represented by a tuple $(\mathcal{C}, \mathcal{L}, \mathcal{K}, \mathcal{U})$, where $\mathcal{C}$ is the encrypted form of the circuit, $\mathcal{L} = ((L_{1,0}, L_{1,1}), \ldots, (L_{n_x,0}, L_{n_x,1}))$ and $\mathcal{K} = ((K_{1,0}, K_{1,1}), \ldots, (K_{n_y,0}, K_{n_y,1}))$ are the lists of the input signals for the agent state and the host input, respectively, and $\mathcal{U} = ((U_{1,0}, U_{1,1}), \ldots, (U_{n_z+n_x,0}, U_{n_z+n_x,1}))$ is the list of the output signals to the host as well as the signals for the updated agent state. Each pair of signals are associated with one bit of input or output, with the first one of the pair encoding 0 and the second one encoding 1. Therefore, the position of each signal in its pair indicates the semantics of that signal. (Notice this correspondence is purely for the simplicity of notation. Alternatively, there could be a table that maps each signal to its semantics.) Notice that the updated agent state, $((U_{n_z+1,0}, U_{n_z+1,1}), \ldots, (U_{n_z+n_x,0}, U_{n_z+n_x,1}))$, equals the $\mathcal{L}$ of the encrypted circuit for the next host. If there is also random input to the agent function, both $\mathcal{L}$ and $\mathcal{K}$ include random coins that will be independently provided by the originator and the hosts, and the random input is the XORs of these coins.

Protocol 2.6.2 *Mobile Agent Computation in The $\mathcal{F}_{\rm COTD}$, $\mathcal{F}_{\rm PKE}$-Hybrid Model*

All messages sent are attached with the session id which is provided by the environment as part of the input to every party. As a setup stage, the hosts are partitioned into n subgroups, and the partition is known by every party.

1. *Each host, upon receiving its input from the environment, sends a notification of this event to the originator O.*

2. *Every host invokes a copy of ideal functionality $\mathcal{F}_{\rm PKE}$ to generate a public key e, and broadcasts this key to all other hosts and O*[19]*. Later, we use the host id (i, j) to identify its corresponding copy of $\mathcal{F}_{\rm PKE}$.*

3. *Having received its input $(x_{(1,0)}, \ldots, x_{(n,0)})$ from the environment, as well as the input notifications and the public keys from all hosts, O first invokes the* Key Generation *operation of ideal functionality $\mathcal{F}_{\rm COTD}$.*

4. *For each host $H_{(i,j)}$, using the enhanced BMR construction, O creates an encrypted circuit $(\mathcal{C}^{(i,j)}, \mathcal{L}^{(i,j)}, \mathcal{K}^{(i,j)}, \mathcal{U}^{(i,j)})$ to compute function $f^{(i,j)}$. In addition, O invokes ideal functionality $\mathcal{F}_{\rm COTD}$ to encrypt $\mathcal{K}^{(i,j)}$, the signals for the host's input. Denote the encrypted signals as $\overline{\mathcal{K}}^{(i,j)} = ((\overline{K}^{(i,j)}_{1,0}, \overline{K}^{(i,j)}_{1,1}), \ldots, (\overline{K}^{(i,j)}_{n_y,0}, \overline{K}^{(i,j)}_{n_y,1}))$.*

5. *Next, O creates n mobile agents. For agent $\mathcal{MA}_i$, $i = 1, \ldots, n$, its code includes the encrypted circuits $\mathcal{C}^{(i,j)}$, the encrypted signals $\overline{\mathcal{K}}^{(i,j)}$, and $\mathcal{U}^{(i,j)}_z = ((U^{(i,j)}_{1,0}, U^{(i,j)}_{1,1}), \ldots, (U^{(i,j)}_{n_z,0}, U^{(i,j)}_{n_z,1}))$, the signals of the output wires corresponding to the local output to the host, for all the hosts in subset i. O also sets $L'^{(i,1)}_k = L^{(i,1)}_{k,x_{(i,0),k}}$ where $x_{(i,0),k}$ is the kth bit of $x_{(i,0)}$, for $k = 1, \ldots, n_x$, as the initial state of $\mathcal{MA}_i$. Following our convention, denote this list of signals by $\mathcal{L}'^{(i,1)}$.*

[19] Since the UC framework does not assume the existence of a broadcast channel, broadcasting is implemented as sending the message separately to each party. Meanwhile, an authenticated broadcasting protocol such as the one introduced later is not necessary in the semi-honest adversarial model.

Then O invokes ideal functionality $\mathcal{F}_{\mathrm{PKE}}^{(i,1)}$ to encrypt the initial agent state. Denote by $\overline{\mathcal{L}}'^{(i,1)}$ the encrypted agent state.

6. *Finally, O sends out $\mathcal{MA}_i$ with its encrypted state to $H_{(i,1)}$, the first host of subset i, for $i = 1, \ldots, n$.*

7. *Every host $H_{(i,j)}$, upon receiving agent $\mathcal{MA}_i$, sends the encrypted agent state $\overline{\mathcal{L}}'^{(i,j)}$ to $\mathcal{F}_{\mathrm{PKE}}^{(i,j)}$ and receives the decryption $\mathcal{L}'^{(i,j)}$.*

8. *Next, $H_{(i,j)}$ sends the encrypted input signals $\overline{\mathcal{K}}^{(i,j)}$ to m agents including $\mathcal{MA}_i$. Then $H_{(i,j)}$ and the m agents engage with $\mathcal{F}_{\mathrm{COTD}}$ to obliviously decrypt one signal from each pair in $\overline{\mathcal{K}}^{(i,j)}$. Specifically, $H_{(i,j)}$ is the decryption user and the m agents are the m servers. For each* Oblivious Decryption, *the pair of ciphertexts is $(\overline{K}_{k,0}^{(i,j)}, \overline{K}_{k,1}^{(i,j)})$, for $k = 1, \ldots, n_y$, and $H_{(i,j)}$'s input bit, the selection bit that determines which ciphertext $H_{(i,j)}$ will actually decrypt, is $y_{(i,j),k}$.*

9. *From $\mathcal{F}_{\mathrm{COTD}}$, $H_{(i,j)}$ receives the signals corresponding to its input $y_{(i,j)}$. Denote this list of host input signals as $\mathcal{K}'^{(i,j)}$.*

10. *$H_{(i,j)}$ evaluates the encrypted circuit $\mathcal{C}^{(i,j)}$ using the input signals $\mathcal{L}'^{(i,j)}$ and $\mathcal{K}'^{(i,j)}$, recovers the value of its own private output using the corresponding $\mathcal{U}_z^{(i,j)}$ vector, and sends the agent (consisting of the encrypted circuits and signals for the following hosts, as well as its updated state, which is in the form of signals and encrypted by $\mathcal{F}_{\mathrm{PKE}}^{(i,j+1)}$ using host $H_{(i,j+1)}$'s public key) to the next host in the subset. If this is the last host in the subset, then the agent is returned back to the originator with its state not encrypted. If the transfer was to a different host, that host repeats steps 7 – 10.*

11. *After all n agents return back to O with their final states. O recovers the values they represent*

and uses function g to combine the results into the final result.

12. *Every host outputs its local output from the circuit evaluation, and O outputs the result from function g.*

2.6.3 Proof of Security

Recall that for now we are considering only security against a static and semi-honest adversary. Consequently the analysis is relatively simple, as one can observe that the only damage such an adversary can do is analyzing the information it sees (known as the adversary's *view*) when strictly following the protocol instructions. While the adversary can apply all kinds of computation on its view, the computation has to be polynomial-time bounded, and it cannot involve any honest party. Hence, the best the adversary can hope for is extracting as much *knowledge* as possible from the view, where by knowledge we mean anything that the polynomial-time adversary is not able to produce by itself. The strategy of the security proof is to show that the adversary's view does not contain any knowledge it should not learn, and this is done by presenting an simulator that can generate a view indistinguishable from the real view using only what the adversary is entitled to get. Put into the UC context, this amounts to demonstrate an ideal adversary that can simulate the execution of Protocol 2.6.2 in the hybrid model.

Lemma 2.6.3 *For any static, semi-honest adversary $\mathcal{H}$ interacting with Protocol 2.6.2 in the hybrid model with access to ideal functionality $\mathcal{F}_{\mathrm{COTD}}$ and $\mathcal{F}_{\mathrm{PKE}}$, and corrupting parties subject to the following restrictions: if $\mathcal{H}$ corrupts hosts only, then it is limited to corrupt hosts from up to $m-1$ out of the n subsets of hosts where m is the threshold parameter of $\mathcal{F}_{\mathrm{COTD}}$, or the adversary can corrupt the originator and any number of hosts; there exists an ideal-model adversary $\mathcal{S}$ such that no environment can distinguish the ideal-model computation from the hybrid-model computation.*

Proof: First, in the UC framework, the environment $\mathcal{Z}$ has very limited access to the parties in all three models (ideal, hybrid, and real). That is, $\mathcal{Z}$ only provides inputs to and reads outputs from the parties. It's easy to verify that Protocol 2.6.2 correctly realizes the functionality of $\mathcal{F}_{\text{MA}}$ when everyone follows the instructions, which is the case with a semi-honest adversary. Hence, $\mathcal{Z}$ cannot tell the difference based on its interaction with the parties.

As a result, the main task of this proof is to construct such a $\mathcal{S}$ that mimics $\mathcal{H}$ in its interaction with $\mathcal{Z}$. Since $\mathcal{Z}$ can interact with $\mathcal{H}$ freely, it is impossible to predict what information they would exchange. Rather, given a particular $\mathcal{H}$, $\mathcal{S}$ can be constructed as follows. $\mathcal{S}$ runs a copy of $\mathcal{H}$ internally and forwards the communication from $\mathcal{Z}$ to this internal $\mathcal{H}$. In the meantime, $\mathcal{S}$ simulates to $\mathcal{H}$ its view in the hybrid model. Thus, the internal $\mathcal{H}$ produces the same communication to $\mathcal{Z}$ as it does in the hybrid model, which $\mathcal{S}$ forwards back to $\mathcal{Z}$. However, $\mathcal{S}$ only has *black box* access to $\mathcal{H}$. Therefore, the centerpiece is the simulation of $\mathcal{H}$'s view.

The UC framework assumes open but authorized communication channels between the parties. Therefore, the view of $\mathcal{H}$ consists of the inputs and outputs of the corrupted parties and all the messages sent in the system except those between the ideal functionalities and the honest parties. By corrupting the same set of parties in the ideal model, it's trivial for $\mathcal{S}$ to provide the corrupted parties' inputs and outputs. The messages seen by $\mathcal{H}$ are the hosts' input notifications and public keys, all the agents travelling from one host to another, the encrypted input signals $\overline{\mathcal{K}}^{(i,j)}$ sent by a host to m agents for decryption which are already included in the agents, and the messages from the ideal PKE and OTD to the corrupted parties. Simulating these messages depends on what parties $\mathcal{H}$ corrupts.

- First of all, no matter what parties $\mathcal{H}$ corrupts, the input notifications from the hosts to the originator (Step 1 of Protocol 2.6.2) is trivial to simulate. $\mathcal{S}$ simulates a notification to the originator which does not contain any private information, once a "$H_{(i,j)}-$input" notification

is received from $\mathcal{F}_{\rm MA}$. $\mathcal{S}$ simulates the hosts' public keys by playing as the ideal functionality $\mathcal{F}_{\rm PKE}$ and interacting with $\mathcal{H}$ to generate the keys.

- If only the hosts are corrupted, in the ideal model $\mathcal{S}$ first submits inputs of the corrupted hosts to $\mathcal{F}_{\rm MA}$, and receives the outputs to these hosts. This is possible because the honest parties automatically forward their inputs to the ideal functionality. Thus $\mathcal{F}_{\rm MA}$ is guaranteed to be ready to process the inputs from the corrupted parties. In fact, the ideal-model computation can be completely over before $\mathcal{S}$ begins to simulate Step 3 of Protocol 2.6.2. However, $\mathcal{S}$ can hold the results from $\mathcal{F}_{\rm MA}$ and only delivers them to a party in the ideal model at a time when the same party obtains output in Protocol 2.6.2. Because $\mathcal{H}$ only corrupts hosts from up to $m-1$ different subsets, $\mathcal{H}$ is not able to hold enough agents for it to invoke $\mathcal{F}_{\rm COTD}$ to decrypt the encrypted input signals. Hence, in the hybrid model, for any encrypted circuit contained in an agent, adversary $\mathcal{H}$ knows only one instance of input to that circuit if the circuit is for a corrupted host, and no input at all if the circuit is for an honest host. Furthermore, only the semantics of the output signals to the hosts are known to $\mathcal{H}$, and the agent state, encrypted or not, makes no sense to $\mathcal{H}$. Therefore, applying Theorem 2.4.3, $\mathcal{S}$ can construct fake agents indistinguishable to $\mathcal{H}$ based on the outputs of the corrupted hosts.

At any time travelling among the hosts, an agent can be divided into four parts. The encrypted circuits $\mathcal{C}$ for the hosts yet to be visited by this agent, the encrypted current agent state $\overline{\mathcal{L}}'$, the encrypted input signals $\overline{\mathcal{K}}$ for the following hosts, and the signals of the host output $\mathcal{U}_z$ which provides the semantic definitions of the output wires to the host. Beginning with the encrypted circuits, basically for a corrupted host, $\mathcal{S}$ constructs a fake encrypted circuit based on the host's output in the way described in Section 2.4.4. For an uncorrupted host, $\mathcal{S}$ constructs a totally random encrypted circuit, in which the truth tables of all the gates contain truly random strings without any meaning. For the current agent state, since it's

encrypted by the ideal PKE, it is simply a group of truly random strings. Likewise, the encrypted input signals $\overline{\mathcal{K}}$ are set to be any truly random strings. And finally, for corrupted hosts, the output signals $\mathcal{U}_z$ are the ones determined by the fake circuit construction which would result in the correct outputs to those hosts, and for uncorrupted hosts they are just random strings with no meaning.

The last issue is the messages from the ideal functionalities ($\mathcal{F}_{\text{COTD}}$ and $\mathcal{F}_{\text{PKE}}$) to the corrupted hosts. Essentially, $\mathcal{S}$ simulates the behavior of the ideal functionalities and interacts with $\mathcal{H}$. The requests to various copies of $\mathcal{F}_{\text{PKE}}$ to encrypt the current agent state are answered with random strings. The decrypted input signals returned by $\mathcal{F}_{\text{COTD}}$ are set to be the ones determined by the fake circuit construction, which are the *on-path* signals selected for the fake circuit. The decrypted agent state returned by a copy of $\mathcal{F}_{\text{PKE}}$ is still in the form of signals, and without knowing their semantics, $\mathcal{H}$ cannot tell them from any group of truly random strings. The only consistency requirement is, if the previous host is also corrupted, the decrypted state should agree with the one encrypted by the previous host. Fortunately, $\mathcal{S}$ already knows that when the previous host asks a copy of $\mathcal{F}_{\text{PKE}}$ to encrypt the state. Similarly, the final agent state, which is not encrypted when sent back to O, is perfectly simulated by truly random strings with the consistency problem taken care of if necessary.

- If the originator is corrupted, then the agent construction is done by $\mathcal{H}$ itself, and no simulation with fake circuits is necessary. Furthermore, it is meaningless to impose a restriction on corrupted hosts in order to maintain the threshold condition, because the adversary already controls O which generates and distributes the secret keys for threshold decryption. Instead, $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{PKE}}$ and $\mathcal{F}_{\text{COTD}}$, interacting with the corrupted O. The job is basically recording the plaintexts from O and returning random strings as the ciphertexts. Next, $\mathcal{S}$ simulates the agents being sent from one host to another. The static part of the agents (i.e.,

the circuits and the lists of encrypted signals and output signals) is already created by $\mathcal{H}$, so no real work is needed. For the agent state, again, since it's encrypted with the next host's public key, it's a set of random strings.

The decrypted input signals returned to a corrupted host is just the plaintexts recorded by $\mathcal{S}$ when O sends the signals to $\mathcal{F}_{\text{COTD}}$ for encryption. On the other hand, since it was $\mathcal{H}$ that selected the semantics for all the signals, the decrypted agent state returned to a corrupted host, although in the form of signals, cannot simply be any random signals. The signals have to represent the correct agent state. This is done in two steps. First, by corrupting O in the ideal model, $\mathcal{S}$ can send requests to $\mathcal{F}_{\text{MA}}$ on behalf of O. Taking this advantage, in the name of O as well as the corrupted host, $\mathcal{S}$ requests to $\mathcal{F}_{\text{MA}}$ and receives the current state of the agent when it visits the corrupted host. Next, $\mathcal{S}$ translates the agent state into signals. This turns out to be a non-trivial work, as the signals for representing agent states (i.e., the $\mathcal{L}$ lists) are never explicitly carried by the agents. Therefore, $\mathcal{S}$ has to build exactly the same encrypted circuits (not fake ones) as $\mathcal{H}$ did in order to learn the signals on those wires corresponding to the agent states. This is possible because of two facts. First, $\mathcal{H}$ is a semi-honest adversary and always follows the protocol. Second, running $\mathcal{H}$ as a subroutine, it is $\mathcal{S}$ that provides the inputs including the random input to $\mathcal{H}$. Thus, $\mathcal{S}$ knows every step of $\mathcal{H}$ in its building of the encrypted circuits and every input bit $\mathcal{H}$ used. Therefore, $\mathcal{S}$ can build the same encrypted circuits and in turn obtain the correct signals to represent the agent state.

For the rest of the simulation issues, $\mathcal{S}$ plays like $\mathcal{F}_{\text{PKE}}$ and returns random strings to the corrupted hosts' requests of encrypting the updated agent state. When simulating the agents coming back to O, $\mathcal{S}$ uses the final states it received from $\mathcal{F}_{\text{MA}}$ and translates them into signals just like the cases of intermediate agent states. ■

2.6.4 Secure Mobile Agent Protocol in The Real Model

By replacing ideal functionalities $\mathcal{F}_{\text{COTD}}$ and $\mathcal{F}_{\text{PKE}}$ with their real-model realizations in Protocol 2.6.2, we obtain an agent computation protocol for the real model.

Protocol 2.6.4 *Secure Mobile Agent Computation (I)*

Run Protocol 2.6.2 with every call to an ideal functionality replaced by executing the corresponding real-model protocol that securely realizes the ideal functionality.

The security of Protocol 2.6.4 is obvious by combining Lemma 2.6.3 and Theorem 2.3.3. However, if the threshold decryption on which the realization of COTD is based is t-secure, the adversary is allowed to corrupt up to t subsets of hosts instead of $m-1$ if it does not corrupt the originator[20]. The last issue to clarify is the security assumption of this protocol. This is the common denominator of the assumptions of the the basic tools as shown in figure 2.3. The security of encrypted circuits is based on the existence of pseudorandom generators, which is equivalent to the existence of one-way functions [35]. Our implementation of secure 1-out-of-2 oblivious transfer is based on a trapdoor permutation and a pseudorandom generator. The Canetti-Goldwasser scheme, the candidate cryptosystem to implement threshold decryption, is based on the Decisional Diffie-Hellman (DDH) problem, whose intractability implies that the *discrete logarithm problem* is an one-way function. Lastly, DDH is the direct basis for one of the CCA-secure PKE schemes [28]. Hence, we have the following theorem.

Theorem 2.6.5 *Assuming that the Decisional Diffie-Hellman problem is intractable, that trapdoor permutations exist, and that there exists a non-interactive threshold decryption scheme such as the Canetti-Goldwasser scheme which securely realizes ideal functionality $\mathcal{F}_{\text{TD}}$, Protocol 2.6.4 is secure against static, semi-honest adversaries under the condition that the adversary corrupts hosts from*

[20] Therefore, the optimal relation between the two is $t = m - 1$.

up to t out of the n subsets of hosts due to the underlying t-secure threshold decryption scheme but not the originator, or any number of hosts and the originator.

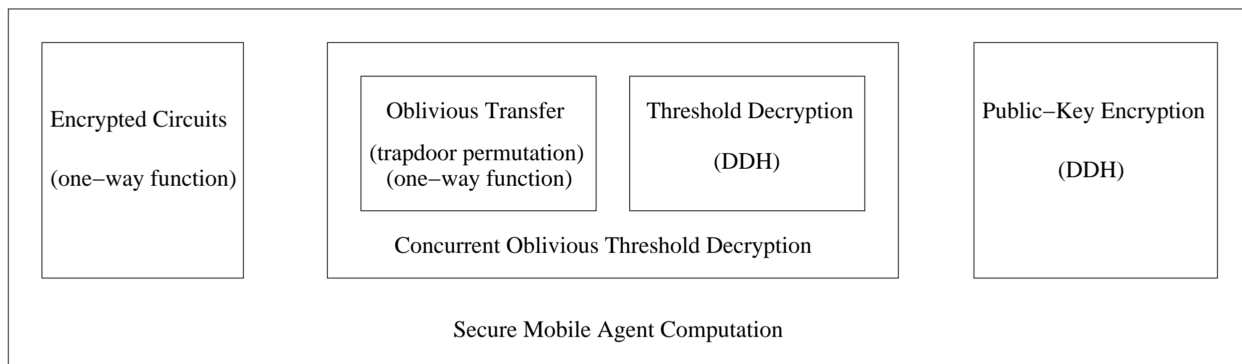

Figure 2.3: Basic cryptographic tools for secure mobile agent computation.

2.6.5 Communication Complexity

One crucial measurement of performance for distributed computing is the round complexity of the communication. This is particularly important in a loosely coupled environment such as the Internet. For Protocol 2.6.4, assume ℓ hosts and n agents in total, where $n \leq \ell$. There are basically four types of communications. The protocol begins with each host notifying the originator that its input is ready, which are ℓ messages. Next, each host broadcasts its public key, resulting in $O(\ell^2)$ messages in a point-to-point network. The agents travelling from one host to another, that is another $O(\ell)$ messages. Finally, for each host, the number of messages for oblivious threshold decryption is $(m-1)+(m-1)\times(2k+1)$ per input bit, where $m \leq n$ is the minimum number of decryption servers required, and k is the number of bit-wise OT in the string OT that transfers a decryption share, which is the length of the seeds of the pseudorandom strings that mask the decryption shares, and so is equal to the security parameter. Thus, the total number of OTD-related messages is $O(n_y km\ell)$. So the round complexity of Protocol 2.6.4 is $O(n_y km\ell + \ell^2)$. In other words, the number of communication rounds is not affected by the size of the agent function,

but is rather determined by the host input size, the number of hosts, and the strength of security we want to achieve.

Furthermore, there is a noticeable property about the communication pattern of Protocol 2.6.4. Other than exchanging public keys with others, each host participates in the computation and communicates with other hosts (who also are currently hosting agents) only when an agent is on-site. In practice, locating other agents is a service provided by underlying multi-agent platform, and the agent itinerary is included in the agent itself so the host does not need to know who is the next host in advance. Furthermore, the hosts' public keys can be broadcasted in a separate, general-purpose operation. Thus, each host does not have to be aware of the whereabouts of other hosts before the protocol execution.

2.7 Protecting Universally Composable Protocols against Malicious Adversaries

Strictly more powerful than its semi-honest counterpart, a malicious adversary can instruct the corrupted parties to deviate arbitrarily from the protocol. The protocols we have presented so far may no longer be secure if the adversary is not restricted to follow the protocol. For example, in bit-wise oblivious transfer (Protocol 2.5.2), a corrupted receiver T can set $y_{1-i} = f(r')$ for a random r' chosen by T. Thus, T can easily discover the other value x_{1-i}. As the malicious adversary may attack a protocol in any way it wants, the defense strategy is for the honest parties to detect such attacks and refuse to play along once an attack is detected. Messages are the only interaction between the parties. So the detection concentrates on verifying the validity of a message received from another party.

A valid message sent by a party is the one that is generated following the protocol instructions. More precisely, each message specified in a protocol can be viewed as the result of a function whose code is the protocol instructions. Depending on the message, the input to this function may include

the party's input to the protocol, some random bits, and the messages the party has received so far. Hence, a message is deemed valid if it was generated from exactly the corresponding protocol instructions and with the genuine data it should depend on.

2.7.1 The Goldreich-Micali-Wigderson Compiler

Given a secure protocol with regard to semi-honest adversaries, Goldreich, Micali, and Wigderson [36, 34] have described a so called *compiler* that transforms such a protocol into one secure against malicious attackers. Basically, what the GMW compiler does is adding a verification on every message received. The GMW compiler assumes the communication model to consist of only a single authenticated broadcast channel (emulatable by a point-to-point network via an authenticated Byzantine Agreement protocol). As a result, every message is received by all the parties regardless of the intended receiver. Each message is verified by all the honest parties. If the verification by any honest party fails, that party refuses to go on with the protocol and notifies all other parties about the rejection, which leads to aborting by all honest parties. The whole protocol instructions are known to all parties, but the data used to generate a message are not public. In fact, they should not be since they are the private data and state of the parties that the protocol intends to protect. To work around this, the GMW compiler consists of three phases. The first two are setup phases before the actual protocol starts. Each party begins with committing its input, and as a result everyone holds the commitments of all other parties' inputs. Second, an augmented coin-tossing protocol is executed so that every party receives as its random tape a distinct, uniformly distributed random string jointly generated by all parties, and everyone else holds a commitment of that string. Note the random string itself is known only to its user.

After these two steps, the parties execute the protocol which is secure against semi-honest adversaries, with the following additional steps. Whenever a party sends out a message, it also

gives an zero-knowledge (ZK) proof about the message's validity. As explained before, a message is the result of applying some protocol instructions probably on the party's input, its random input, and/or the messages it received so far. As the whole protocol is polynomial-time bounded, so is this message generation. Therefore, the assertion of the message's validity is an NP statement, and so a zero-knowledge proof of this assertion exists ([35] Theorem 4.4.11). As an example, define all valid ones of a particular message sent by party P_i as an NP language $L \stackrel{\text{def}}{=} \{m : \exists\alpha, \text{such that } m = f(\alpha)\}$, where f is the protocol instructions for generating this message and α is the union of the party's input, random input, and all received messages. Knowing α, a polynomial-time P_i is able to give a ZK proof showing its message $m \in L$. However, this only answers half of the question, because what we want to ensure is the honesty of the witness α — P_i's input and random input are the ones it has committed to, and the messages are the real messages it has received. To this end, the NP language L is augmented to $L' \stackrel{\text{def}}{=} \{(m, u) : \exists\alpha, \text{such that } m = f(\alpha) \text{ and } u = h(\alpha)\}$, where $h(\alpha) \stackrel{\sqcap\{}{=} (u_1, u_2, u_3)$, and u_1 is the commitment of P_i's input, u_2 is the commitment of P_i's random input, and u_3 is the messages so far received by P_i. Recall that every party holds the commitments of all other parties' inputs and random inputs. Furthermore, all the messages sent in the system are known to everyone due to the single broadcast channel. Hence, all other parties already know the correct $h(\alpha)$, denoted by β. P_i sends out (m, u) followed by a *zero-knowledge strong-proof-of-knowledge* ([35] Section 4.7.6 and Theorem 4.7.15 for existence of such ZK proof of knowledge for any NP language) showing $(m, u) \in L'$ and P_i knows the witness α. Every party verifies the validity of the proof as well as if $u = \beta$. If either fails, that party catches P_i's cheating.

Assuming trapdoor permutations exist, the three components of the GMW compiler — input commitment, augmented coin-tossing, and message validation (or authenticated computation as called in [34]) — can all be securely implemented in the presence of a static and malicious adversary [34]. Essentially, the effect of the compiler is forcing a malicious adversary to follow the

protocol, or it is bound to be detected by the honest parties. However, all the results are obtained under the traditional, stand-alone context.

2.7.2 A Universally Composable Compiler

Canetti et. al. [24] presents a universally composable compiler analogous to the GMW compiler. Deriving a UC compiler turned out to be not as simple as just replacing each of the three components of the GMW compiler with its UC version, as the receiver of a UC commitment learns only a formal "receipt" assuring a value was committed to, but no information that uniquely determines the committed value is received. Hence, later the receiver has no information against which it can check to ensure the witness of the ZK proof is the one that an honest party should use. As a result, the UC compiler takes a slightly different strategy and combines all three components in the GMW compiler into one ideal functionality known as Commit and Prove ($\mathcal{F}_{CP}$). Naturally, the commit phase of $\mathcal{F}_{\mathrm{CP}}$ is called upon to execute input and random tape commitment, while the prove phase is used to verify each message received.

The basic idea of $\mathcal{F}_{\mathrm{CP}}$, which also describes how it can be realized, is as follows. Assume the existence of ideal zero-knowledge proof $\mathcal{F}_{\mathrm{ZK}}$, which receives a statement x and a witness w from the prover, and if relation $R(x, w)$ holds then forwards x to the verifier, for some predetermined relation R. In the commit phase, a committer commits its value w with some perfectly (or computationally) binding, non-UC commitment scheme C. Denote the result by c. Then the committer proves to the receiver using $\mathcal{F}_{\mathrm{ZK}}$ that c is a valid commitment, in which c is the statement, w is the witness, and the relation is the scheme C. This results in the receiver learning the commitment c but not w. In the prove phase, where the committer wishes to prove that some value x stands in an appropriate relation R with the committed value w, the committer sends (x, c) as the statement and w as witness to another $\mathcal{F}_{\mathrm{ZK}}$ which verifies c is the commitment of w and $R(x, w)$ holds. Consequently

the receiver gets (x, c). The receiver accepts x if c is the one it previously received in the commit phase. In the UC compiler, the committer is the sender of a message. x is the message and is generated from w — the combination of the sender's private inputs, random inputs, and all the messages it received so far.

Following the notations in [24], realizing the commit-and-prove functionality in a multi-party scenario, denoted as $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ for one-to-many commit and prove, requires a common (random) reference string shared by all parties formalized as $\mathcal{F}_{\mathrm{CRS}}$, and an authenticated broadcast channel for any party to send the same message to all others, formalized as $\mathcal{F}_{\mathrm{BC}}$ and realized by Goldwasser and Lindell [37] on top of authenticated point-to-point channels. Assuming that both exist as well as a trapdoor permutation, realization of the one-to-many UC zero-knowledge proof formalized as $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$ is given in [24]. In particular, in the case with a static adversary, $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$ is simply one-to-many extension of the non-interactive ZK proof of De Santis el. al. [58]. Finally, $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ is realized in the $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$-hybrid model with the basic idea outlined above.

What happens when cheating is caught? Obviously, when an honest party detects the adversary's cheating in a message, it should at least not respond to that message as if it were an honest message. The GMW compiler lets the honest party treat a cheating message as the indication that the sender has aborted the protocol. The UC compiler is a little more relaxing, letting the honest party to simply ignore the cheating message. Therefore, the protocol may end up hanging. As discussed before, the case when there is no output produced is regarded as secure. In summary, the UC compiler forces a malicious adversary to behave semi-honestly or one or more honest parties will not generate output.

We conclude this section with the following theorem concerning the UC compiler.

Theorem 2.7.1 *Let* Π *be a non-trivial multi-party protocol that securely realizes a well-formed ideal functionality* $\mathcal{F}$ *which does not utilize its knowledge of the identities of corrupted parties, in*

the presence of static, semi-honest adversaries. Let Comp(Π) *be the protocol obtained by applying the multi-party compiler described in [24]. Then* Comp(Π) *securely realizes* $\mathcal{F}$ *in the presence of static, malicious adversaries. The necessary assumptions are that trapdoor permutations exist and the parties shared a common reference string.*

The theorem is essentially the combination of multiple results from [24], including Proposition 9.6, the results concerning the realization of $\mathcal{F}_{\mathrm{CP}}^{1:\mathrm{M}}$ in the the static adversarial model, and the universal composition with joint state state theorem (Theorem 3.4).

2.8 Secure Mobile Agent Computation against Static, Malicious Adversaries

Implied in the UC compiler is the requirement that all (honest) participants in the protocol receive and check every message sent in the system, regardless of who is the intended receiver. Otherwise, a party won't have a global view of the protocol execution when it wants to check the message intended for itself. This is in conflict with the unique setting of the mobile agent paradigm, where the originator is not required to be online after the agents are sent out. Observe that the compiler is a general technique that can be applied to any protocol. Consequently, it does not utilize any specific features of the protocol, which fortunately in the agent case already provides some degree of protection against malicious behaviors and can indeed free the originator from the always-online restriction.

The key observation is that the encrypted circuit provides strong resistance against tampering. Intuitively, without a valid set of input signals, it is infeasible to obtain a valid set of output signals for an encrypted circuit. Since the only messages sent to the originator are the agents returning home with their final states, and the agent states are just a bunch of output signals, by comparing these signals against the valid signals, O is able to catch any tampering with the agents. We begin with formally proving an important security property of the encrypted circuit.

2.8.1 The Infeasibility of Effectively Predicting Off-Path Signals

Let's first look at a single encrypted gate. Referring to the introduction in Section 2.4.1, consider a two-input encrypted gate g whose input and output wires are denoted α, β, and γ, respectively, and whose truth table is defined by Equation (2.4). Suppose one holds a pair of input signals $s_{\alpha a}$ and $s_{\beta b}$, where a and $b \in \{0,1\}$. Thus, one can evaluate gate g by computing $G_b(s'_{\alpha a}) \oplus G_a(s'_{\beta b}) \oplus A^g_{ab}$. (Recall that $s'_{\alpha a}$ is the binary string $s_{\alpha a}$ without the last tag bit a.) Let $s_{\gamma c}$ denote the resulting output signal. We call $s_{\alpha a}$, $s_{\beta b}$, $s_{\gamma c}$, and A^g_{ab} as the *on-path* signals and truth table entry. Accordingly, all the other signals and entries in the truth table are *off-path*.

Lemma 2.8.1 *Given a two-input encrypted gate as described above in which the length of the wire signals is k, excluding the last tag bit, for every probabilistic polynomial-time algorithm A, every positive polynomial $p(\cdot)$, and sufficiently large k's, the probability that A outputs a correct off-path signal is strictly less than $\frac{1}{2^k} + \frac{1}{p(k)}$.*

Proof: Recall that a wire signal is a binary string uniformly selected by the constructor of the encrypted gate. Knowing its length k, one can guess any wire signal with probability at least $\frac{1}{2^k}$. This claim basically says random guessing is the best one can do to an off-path input or output signal. To prove it, first imagine the case where instead of defining the truth table of the gate according to Equation (2.4), the constructor simply filled in the off-path entries in the truth table with truly random strings. This way, holding an encrypted gate and a pair of input signals gives absolutely no information at all regarding the off-path input and output signals. Therefore, no algorithm can guess the off-line signals better than uniformly random guessing, no matter how powerful it is. Although in fact the truth table of an encrypted gate is not constructed like that, the pseudorandom generator G effectively hide the information of the off-path signals such that to a polynomial-time algorithm the off-path entries in the truth table are just like truly random

strings. Otherwise, we can distinguish a pseudorandom distribution ensemble from a corresponding truly random distribution ensemble in polynomial time.

More formally, assume for some positive polynomial $p(\cdot)$ and infinite many k's, there is a polynomial time algorithm A that outputs a correct off-path signal with probability $\geq \frac{1}{2^k} + \frac{1}{p(k)}$. We construct a distinguisher D that distinguishes pseudorandom binary strings of length $2k+2$ from truly random strings of the same length. The proof is analogous to the proof in Section 2.4.4. Define pseudorandom distribution $G(U_k)$ and truly random distribution U_{2k+2} as in Definition 2.4.4. We construct a polynomial-time distinguisher D for the ensembles of $G(U_k)$ and U_{2k+2} by repeated sampling. Consider two independent $(2k+2)$-bit strings X and Y, both of which come from either $G(U_k)$ or U_{2k+2}. Without loss of generality, D creates a gate g and selects input signal $s_{\alpha 0}$ for wire α and $s_{\beta 0}$ for β, as well as output signals $s_{\gamma 0}$ and $s_{\gamma 1}$. D defines randomly the semantics of these signals and constructs a truth table for g as the one defined by Equation (2.7). As argued before, if X and Y are $G(U_k)$-distributed, then g is an encrypted gate. If X and Y are U_{2k+2} distributed, then g is a *fake* gate which contains no information in its truth table about the off-path input and output signals. D feeds A with gate g and the two input signals $s_{\alpha 0}$, $s_{\beta 0}$. Then it checks the correctness of A's guesses of the off-path signals. In particular, D runs the pseudorandom generator G on A's guesses of the offline input signals and compare the results against the corresponding parts of X and Y. For A's guess of the off-path output signal, D simply compares against its selection of that signal. If any of these checkings passes, D outputs 1. Otherwise, D outputs 0. Obviously, if A runs in polynomial time, so does D.

If g is a fake gate (i.e., X and Y are U_{2k+2}-distributed), no algorithm can do better than uniformly randomly guessing any of the off-path signals, because there is simply no information about those signals. In fact, verifying A's guesses of the off-path input signals are bound to fail since X and Y are not pseudorandom. So D outputs 1 on a fake gate with probability at most $\frac{1}{2^k}$

for correctly guessing the output signal. (Notice the tag bit is deducible.) If g is a real encrypted gate, by our hypothesis, A can do better than random guessing, and so D outputs 1 on a real encrypted gate with probability $\geq 3(\frac{1}{2^k} + \frac{1}{p(k)})$. Hence, the gap of the probabilities that D outputs 1 in these two cases is at least $\frac{2}{2^k} + \frac{3}{p(k)}$. In other words, D distinguishes the ensembles of $G(U_k)$ and U_{2k+2}, which is in contradiction with Lemma 2.4.5. ■

Recall that besides encrypted gates with two inputs and a single output, secure encrypted circuits require a special kind of gate called the splitter to eliminate the correlations due to a single wire's driving multiple gates. A splitter is an one-input, two-output gate whose truth table is defined by Equation (2.6). The same unpredictability can be said about the off-path signals of a splitter gate.

Lemma 2.8.2 *Given a splitter gate as described in Section 2.4.3 in which the length of the wire signals is k, excluding the last tag bit, for every probabilistic polynomial-time algorithm A, every positive polynomial $p(\cdot)$, and sufficiently large k's, the probability that A outputs a correct off-path signal is strictly less than $\frac{1}{2^k} + \frac{1}{p(k)}$.*

Proof (Sketch): The proof is analogous to the one above. However, we only need a single sample string X. Assuming $s_{\alpha 0}$ is the on-path input signal, X_0 is used in the place of $G_0(s'_{\alpha 1})$ and X_1 is used in the place of $G_1(s'_{\alpha 1})$. The rest of the proof is the same as the proof of Lemma 2.8.1. ■

According to Lemma 2.4.2, an encrypted circuit generated by the enhanced BMR construction does not have any wire shared by multiple gates. Based on the two lemmas we just proved, a simple induction from the input gates to the output gates leads to the following property of encrypted circuits.

Theorem 2.8.3 *Given an encrypted circuit generated according to the enhanced BMR construction in which the length of every wire signal is k, excluding the last tag bit, for every probabilistic*

polynomial-time algorithm A, every positive polynomial $p(\cdot)$, and sufficiently large k's, the probability that A outputs a correct off-path signal for any given wire in the circuit is strictly less than $\frac{1}{2^k} + \frac{1}{p(k)}$.

2.8.2 Secure Mobile Agent Protocol against Static, Malicious Adversaries

In this section, we apply the UC compiler to Protocol 2.6.4, which is secure in the semi-honest adversarial model, with the exception that the originator only receives and verifies the messages intended for itself. For simplicity, we only give an outline of the resulting protocol in the real model. Detailed description of how the compiler is applied to a protocol can be found in Figure 17 of [24].

Protocol 2.8.4 *Secure Mobile Agent Computation (II)*

Assuming the originator and all the hosts share a common reference string, apply the universally composable compiler of [24] to Protocol 2.6.4, as depicted in Figure 17 of [24], with the following notes and exceptions.

1. *Replace the calls to each instance of ideal functionality $\mathcal{F}_{CP}^{1:M}$ with invocations of its real-model realization.*
2. *All message transmission is done through the broadcasting protocol of [37], which realizes an authenticated broadcast channel to all players in the protocol on top of authenticated point-to-point channels.*
3. *All participants including the originator O carry out the* random tape generation *phase. Then everyone starts the execution of Protocol 2.6.4 with additional steps required by the compiler. Notice that every message is broadcasted to all parties.*
4. *After sending out the agents (i.e., step 6 of Protocol 2.6.4), O only participates in the protocol when an agent returns home. Specifically, O drops out of the broadcasting of any messages*

other than the agent's coming back. All the hosts continue to behave according to Protocol 2.6.4 and the compiler.

5. *When O receives an agent back from the hosts, it checks the validity of the agent's final state. That is, if the strings that represent the agent state are valid output signals. If not, O ignores this message. Otherwise, it treats the agent according to Protocol 2.6.4.*

2.8.3 Proof of Security

This section proves the following theorem.

Theorem 2.8.5 *Assume that the Decisional Diffie-Hellman problem is intractable, that trapdoor permutations exist, that there exists a non-interactive threshold decryption scheme such as the Canetti-Goldwasser scheme which securely realizes ideal functionality $\mathcal{F}_{\mathrm{TD}}$, and that all parties share a common reference string. Then Protocol 2.8.4 is secure against static, malicious adversaries under the condition that the adversary corrupts hosts from up to t out of the n subsets of hosts due to the underlying t-secure threshold decryption scheme but not the originator, or any number of hosts and the originator.*

Proof (Sketch): Intuitively, this protocol emulates Protocol 2.6.4 despite interacting with a malicious adversary. Thus the security (or the simulatability) of this protocol by an ideal-model execution follows due to the security of Protocol 2.6.4.

We give a sketch of the proof that Protocol 2.8.4 emulates Protocol 2.6.4, since the bulk of the proof is identical to the proof of Proposition 9.6 of [24]. For a malicious adversary $\mathcal{A}$ interacting with Protocol 2.8.4, we construct a semi-honest adversary $\mathcal{A}'$ interacting with Protocol 2.6.4 such that the two executions are indistinguishable to the environment. Like before, $\mathcal{A}'$ runs $\mathcal{A}$ internally, receiving the messages $\mathcal{A}$ sends and simulating the messages it expects. The simulation is identical

to the one in [24]. Basically, $\mathcal{A}'$ plays the role of the honest parties in Protocol 2.8.4. Whenever externally $\mathcal{A}'$ sees a message from an honest party in Protocol 2.6.4 (since it has control over the communication channels), it simulates the corresponding messages $\mathcal{A}$ expects to see from the same honest party in Protocol 2.8.4. This is possible due to the security (i.e., simulatability) of the commit-and-prove functionality. Whenever $\mathcal{A}$ broadcasts a message intended for an honest party, $\mathcal{A}'$ checks the validity of this message just as every honest party would do. If the message is honest, externally $\mathcal{A}'$ sends out a corresponding message in Protocol 2.6.4 to the intended honest party. If cheating is detected, $\mathcal{A}'$ simply ignores the message.

This simulation assumes that every honest party follows the compiler. In Protocol 2.8.4, the originator O is an exception, which does not participate in the broadcasting of messages sent between the hosts. Since O does not respond to any of those messages in Protocol 2.6.4 and the compiler instructs a party to ignore any cheating messages, if O participated in the broadcasting of those messages, it would do nothing more than quietly listening to the broadcasting channel[21]. So its absence does not make any difference to the system. The only difference this causes is that internally O does not maintain a history of all the honest messages sent by all the players so far, which will be needed later by the compiler to check the messages intended for O, i.e., the agents returned. However, $\mathcal{A}'$ can still use the above simulation, particularly the message verification based on knowing all the previously sent messages, as $\mathcal{A}'$ sees all the messages anyway, which are either simulated by $\mathcal{A}'$ itself or are received from the internal $\mathcal{A}$. The question is "does this simulate what happens in the execution of Protocol 2.8.4?".

Observe that $\mathcal{A}'$'s simulation would catch all cheatings by $\mathcal{A}$, because a message is uniquely determined by the sender's private input, random input, and the messages it sees so far, which are all verified by the simulation through the commit-and-prove functionality. Thus, the proof turns

[21]The broadcast protocol [37] does require messages from every participants. But these messages are only for implementing the broadcasting and so can be treated as transparent to the upper layers.

to showing in Protocol 2.8.4 O can catch all cheatings, too. First, notice that up to the point right before an agent is returned to O, adversary $\mathcal{A}$ has to behave honestly due to the compiler. Otherwise, the cheating would have been caught and the protocol would not have moved on. Hence, the only cheating $\mathcal{A}$ can do is deviate from evaluating the last encrypted circuit of the agent with the input signals it obtained honestly. Since $\mathcal{A}$ cannot get help from any honest party (because of the compiler) and by itself $\mathcal{A}$ cannot discover any input signals (because of the security of COTD against semi-honest adversary), the input signals $\mathcal{A}$ obtained honestly are the only signals it possibly has. Therefore, to obtain any output signals (which will be the final agent state) other than the ones from the honest input amounts to predicting the off-path signals of the encrypted circuit. Theorem 2.8.3 asserts no polynomial-time bounded adversary can do that effectively better than probability $\frac{1}{2^k}$ for signals of length $k+1$. In other words, the probability that O misses catching a cheating agent state (and hence the simulation fails) is essentially $\frac{1}{2^k}$. Thus the simulation is indistinguishable to a polynomial-time environment. ■

2.8.4 Communication Complexity

Obviously, the UC compiler incurs large communication overhead. Specifically, implementing authenticated broadcasting in a point-to-point network requires message exchanges among all the parties. In our case, that is $O(\ell^2)$ messages for ℓ hosts. Since the underlying ZK proof for realizing $\mathcal{F}_{\mathrm{ZK}}^{1:\mathrm{M}}$ is non-interactive, both the commit and the prove operations of $\mathcal{F}_{\mathrm{CP}}$ involves only one broadcasting message. The random tape generation of the UC compiler requires $\ell \times (2 \times (\ell - 1))$ broadcasting messages. Likewise, the input commitments add another $(\ell + 1)$ broadcasting messages. Finally, all the messages in Protocol 2.6.4 are broadcasted, each with a non-interactive ZK proof attached. Altogether, there are $O(3\ell^4 + n_y k m \ell^3)$ messages in Protocol 2.8.4, and again the round complexity is not affected by the size of the encrypted circuit. Furthermore, this protocol

is more tightly coupled than the previous one in that every host has to know the locations of all other hosts.

CHAPTER 3

A PRACTICAL MOBILE AGENT PROTOCOL

Powerful as it is, the UC compiler achieves its security through extensive communications among the participants of the protocol. The purpose is to enable every party to maintain an honest, global view of the execution so that any cheating by the adversary will be immediately detected. Although Protocol 2.8.4 fits in the basic agent communication model, it incurs large communication overhead among all the hosts throughout the whole protocol execution. In addition, every player must have the knowledge about all the other players. This requirement of tightly coupled environment does not comply with the scenario of some mobile agent applications, such as e-commerce applications over the Internet. In the second part of this research, we turn our attention to obtaining a more efficient protocol.

3.1 Introduction

We would like the mobile agent protocol to work the following communication model. The only messages from the originator are the mobile agents, sent to the first groups of hosts. From then on, the originator is only allowed to receive the agents back from the hosts. Each host is “activated” when an agent arrives and finishes its job once the agent left. An active host may communicate freely with other “active” hosts, but there is no communication to an inactive host other than sending the agent to that host. The only possible message to an inactive host is the migration of an mobile agent. Under this model, a host is purely an execution environment for mobile agents. No preparation or aftermath processing is needed from the host. It also may have no idea about other hosts, as long as the multi-agent system provides some kind of directory service for agents

to locate each other. The communication from and to the originator is the minimum possible. We call this model the *restricted agent communication model*.

This model certainly prevents the use of the UC compiler. In fact, based on the current state of the art, it *seems* that we are not able to achieve equivalent level of security as Protocol 2.8.4 does, due to the limitation that a host cannot exchange messages with others whenever the agent is not on site. Simply put, an honest host is totally unaware of what has happened before an agent arrives. Although one can ask for a zero-knowledge proof of no wrong doing from the previous host, there is no way to verify the honesty of the witness on which the proof is based. Not surprisingly, this is exactly the core issue that the compiler addresses.

Zero-knowledge proofs are themselves another fundamental primitive which turn out to be unaffordable most of the time in practice. The current knowledge about constructing a zero-knowledge proof is based on a general but inefficient theorem. Theorem 4.4.11 of [35] asserts that given an NP language, a zero-knowledge proof can be constructed by reducing the language to an NP-complete language which has a direct ZK proof, such as the graph-3-coloring problem. The same philosophy lies under the construction of a non-interactive ZK proof [58]. Reducing an NP language to an NP-complete language is mostly of theoretic interest. In practice, it usually involves multiple reductions, each an inefficient process by itself, in order to reach the targeted NP complete language (A tree depiction of typical reduction paths appears in [27]). On the other hand, ZK proofs are indispensable in almost all provably secure protocols[1]. For example, suppose a real-model malicious adversary can modify the inputs of corrupted hosts. For an ideal-model adversary to simulate the case when a host's actual input is modified from the original one, it must find out what the actual input is and change the input in the ideal model accordingly. Currently this can only be done through requiring a zero-knowledge proof of knowledge from the host about

[1]In our protocol above, ZK proofs are enclosed in the UC compiler.

its input. The zero-knowledge property is necessary as otherwise the proof leaks the host's private input.

In the following, we study a protocol which runs in the restricted agent communication model and does not use the UC compiler and zero-knowledge proofs. We analyze the security of this protocol. In light of the trade-offs made for the sake of efficiency, we do not rigorously prove the security but only give reasonable arguments. In Chapter 4, we implement this protocol as well as the ACCK protocol, and compare their performance through experiments.

3.2 Security against Malicious Adversaries

It is easy to verify that Protocol 2.6.4 satisfies the restrictions on communication introduced for practicality. However, security against only semi-honest adversaries is not enough in practice. Thus, we enhance this protocol with defense against malicious attacks while preserving its other practical properties. Deprived of the use of a compiler and zero-knowledge proofs, our defense lies in the security of each building block of Protocol 2.6.4. As shown in the following, most *practical* attacks can be protected against without sacrificing the efficiency of the resulting protocol.

3.2.1 Secure Oblivious Threshold Decryption

Our construction of oblivious threshold decryption is a combination of non-interactive threshold decryption and oblivious transfer with little overlap. This enables us to consider each of the two separately against malicious attacks.

A threshold cryptosystem is a public key encryption scheme in the first place. In an isolated view, the security of a public key encryption scheme against a malicious adversary considers the ultimate attack in which the adversary has the assistance of an oracle to decrypt encryptions of its choice, except for the ciphertext that it is about to attack (i.e., finding out information about

this particular ciphertext). This is generally referred to as the Chosen Ciphertext Attack (CCA). Furthermore, in threshold decryption, the adversary sees the decryption shares of its chosen ciphertexts too. The desired security is that even with CCA attacks, the adversary cannot distinguish the target encryption from any valid encryption.

The threshold decryption scheme of Canetti and Goldwasser [22] is CCA-secure. A more efficient, non-interactive scheme was later devised by Shoup and Gennaro [60]. Under the *random oracle model* [14], where cryptographic hash functions are replaced with accesses to an oracle which returns truly random bits, this scheme is proved to be CCA-secure. With this additional assumption, the Shoup-Gennaro scheme has a few advantages over the Canetti-Goldwasser scheme. First, the SG scheme does not require any pre-shared secrets among the decryption servers other than the decryption key, while in the CG scheme, decrypting L ciphertexts requires L pre-computed secrets shared by the servers, which restricts the total number of decryptions by a pre-determined limit. Second, the SG scheme provides a handy way for the user to check the validity of a decryption share, an efficient defense against denial-of-service attacks by the adversary. Third, in the SG scheme, the minimum number of shares necessary for a successful decryption, denoted by m in our previous discussion, is optimal — one more than t the total number of corrupted servers. There are two variants of the Shoup-Gennaro scheme. TDH1 is based on the intractability of the Computational Diffie-Hellman problem, where the more efficient TDH2 is based on the Decisional Diffie-Hellman problem.

Once a malicious adversary controls some decryption servers, it can instruct these servers to deviate from the prescribed behavior. Security must therefore be extended to the case when corrupted servers do not follow the protocol[2]. This is generally referred to as the robustness of the threshold decryption scheme [22]. For non-interactive threshold decryption, deviations of corrupted

[2]In contrast, these attacks do not apply to traditional public key encryption scheme as no interaction is involved.

servers are of no help toward attacking a target encryption. The only possible damage is denial of service. Robustness can therefore be achieved by having a verification scheme for the user to check the validity of decryption shares and assuming enough honest servers in the system.

For easy reference, we include a description of the TDH2 variant of the Shoup-Gennaro scheme as follows.

Algorithm 3.2.1 *The Shoup-Gennaro Threshold Decryption Scheme [60]*

The TDH2 system works over an arbitrary group G of prime order q, with generator g. It requires m shares to decrypt a message, and is secure against up to $m-1$ malicious decryption servers. For simplicity, assume that messages and their labels are l-bit strings. We use three hash functions:

$$H_1 : G \to \{0,1\}^l, H_2 : \{0,1\}^l \times \{0,1\}^l \times G^4 \to Z_q, H_4 : G^3 \to Z_q$$

***Key Generation**: For an m out of n scheme, assume $q > n$. Choose Random points $f_0, \ldots, f_{m-1} \in Z_q$. Define a polynomial $F(X) = \sum_{j=0}^{m-1} f_j X^j \in Z_q[X]$. For $1 \leq i \leq n$, set $x_i = F(i) \in Z_q$ and $h_i = g^{x_i}$. Set $x = F(0)$ and $h = h_0 = g^x$. Choose another random generator $\bar{g} \in G$ and $\bar{g} \neq g$.*

The public key PK consists of a description of the group G, along with g, $\bar{g}$, and h. The public verification key VK for verifying decryption shares, consists of the public key PK and the tuple $\{h_1, \ldots, h_n\}$. For $1 \leq i \leq n$, the secret key SK_i consists of the public key PK, the index i, and the value x_i.

***Encryption**: To encrypt a message $z \in \{0,1\}^l$ with label $L \in \{0,1\}^l$ which can be any information associated with z, choose randomly r, $s \in Z_q$, and compute*

$$c = H_1(h^r) \oplus z, u = g^r, w = g^s, \bar{u} = \bar{g}^r, \bar{w} = \bar{g}^s, e = H_2(c, L, u, w, \bar{u}, \bar{w}), f = s + re.$$

The ciphertext is $(c, L, u, \bar{u}, e, f)$.

Label Extraction*: Given a ciphertext* $(c, L, u, \bar{u}, e, f)$*, anyone can extract its label by output* L[3].

Decryption*: Given a ciphertext* $(c, L, u, \bar{u}, e, f)$*, decryption server* i *checks if* $e = H_2(c, L, u, w, \bar{u}, \bar{w})$*, where* $w = g^f/u^e$ *and* $\bar{w} = \bar{g}^f/\bar{u}^e$*. If the condition does not hold, the ciphertext is invalid and the server outputs* $(i,$ *"?"*$)$*. Otherwise, the server chooses a random* $s_i \in Z_q$ *and computes*

$$u_i = u^{x_i}, \hat{u}_i = u^{s_i}, \hat{h}_i = g^{s_i}, e_i = H_4(u_i, \hat{u}_i, \hat{h}_i), f_i = s_i + x_i e_i.$$

and outputs (i, u_i, e_i, f_i).

Share Verification*: Given the verification key* VK*, a ciphertext* $(c, L, u, \bar{u}, e, f)$*, and a decryption share of server* i*. The verification begins with checking if* $e = H_2(c, L, u, w, \bar{u}, \bar{w})$*, where* $w = g^f/u^e$ *and* $\bar{w} = \bar{g}^f/\bar{u}^e$*. If it does not hold, an honest server should output share* $(i,$ *"?"*$)$*. If the condition holds, then a valid share is of the form* (i, u_i, e_i, f_i)*, and* $e_i = H_4(u_i, \hat{u}_i, \hat{h}_i)$*, where* $\hat{u}_i = u^{f_i}/u_i^{e_i}$*,* $\hat{h}_i = g^{f_i}/h_i^{e_i}$.

Share Combination*: With* m *valid shares, all of which are either of the form* $(i,$ *"?"*$)$ *or of the form* (i, u_i, e_i, f_i) *where* $i \in S$ *of cardinality* m*, the user outputs "?" in the first case, or otherwise*

$$z = H_1(\prod_{i \in S} u_i^{\lambda_{0i}^S}) \oplus c.$$

where $\lambda_{ij}^S \in Z_q$ *are the coefficients such that* $F(i) = \sum_{j \in S} \lambda_{ij}^S x_{i_j}$.

As to oblivious transfer, Protocol 2.5.2 is not secure when the receiver cheats by sending to the sender two random values both of which it can inverse. This in turn makes the string-wise oblivious transfer insecure. Without a compiler-like mechanism, the sender has to possess some

[3] As an application, a decryption server can check the label of a ciphertext before proceeding to decrypt the ciphertext.

other information against which it can check the validity of the receiver's message. Based on the Computational Diffie-Hellman problem, Bellare and Micali [13] gave a non-interactive OT protocol which has the same framework as Protocol 2.5.2.

Protocol 3.2.2 *The Bellare-Micali Non-interactive Oblivious Transfer Protocol — Basic Version [13]*

Assume prime number p and generator g of Z_p^*. Suppose element $C \in Z_p^*$ is public such that no one knows its discrete log. The sender A has two strings s_0 and s_1. The receiver B has input bit $i \in \{0, 1\}$.

1. *B picks at random $x \in \{0, \dots, p-2\}$. Then it computes $\beta_i = g^x$ and $\beta_{1-i} = C \cdot (\beta_i)^{-1}$. B sends (β_0, β_1) to A.*

2. *A checks if $\beta_0 \cdot \beta_1 = C$. A ignores the message if the condition does not hold. Otherwise, A proceeds to the next step.*

3. *A picks at random $y_0, y_1 \in \{0, \dots, p-2\}$ and computes $\alpha_0 = g^{y_0}$, $\alpha_1 = g^{y_1}$. A also computes $\gamma_0 = \beta_0^{y_0}$, $\gamma_1 = \beta_1^{y_1}$, $r_0 = s_0 \oplus \gamma_0$, $r_1 = s_1 \oplus \gamma_1$. A sends $(\alpha_0, \alpha_1, r_0, r_1)$ to B.*

4. *Upon receiving $(\alpha_0, \alpha_1, r_0, r_1)$, B computes $\gamma_i = \alpha_i^x$, and then recovers $s_i = \gamma_i \oplus r_i$.*

In practice, if the non-interactive property is not required, A can choose C by itself and sends it to B before the protocol starts. By verifying that the product of β_0 and β_1 is equal to C, the sender ensures the receiver does not know the discrete log of both numbers, which would imply knowing the discrete log of C. According to the Computational Diffie-Hellman assumption, without the knowledge of the discrete logs of both α_{1-i} and β_{1-i}, it is infeasible for B to compute γ_{1-i}, which is necessary to recover s_{1-i}. Thus, this protocol defeats any malicious attempt by the receiver. On the other hand, by β_0 and β_1 themselves, it is impossible for the sender to find out of which

one the receiver knows the discrete log, and there is no other message from the receiver. Hence, no malicious sender can detect or affect which value the receiver gets. Bellare and Micali pointed out that the security of each individual bit of s_{1-i} is not clear. A more secure scheme is presented in [13] that transfers one bit at a time with similar framework as Protocol 3.2.2. The obvious tradeoff is the performance. Through our experimental study, we found the more secure version incurs too much overhead to be practical in the mobile agent setting. Unless a specific attack on the bit security of the basic Bellare-Micali OT protocol is identified, it is practically acceptable to use the basic version, which is presented above.

Because the Shoup-Gennaro threshold decryption scheme (Algorithm 3.2.1) is non-interactive, combining the SG scheme and the Bellare-Micali OT protocol (Protocol 3.2.2) the same way like we did in Chapter 2 gives us another realization of oblivious threshold decryption. Practically speaking, with the use of strong cryptographic hash functions, this implementation of OTD is computationally secure against a malicious user who tries to illegally obtain decryptions, and perfectly secure against a malicious server who tries to discover the user's private input bit.

An implementation issue for both primitives is finding the appropriate cyclic group. Algorithm 3.2.1 requires a cyclic group of prime order q. This can be done by selecting a random prime p and finding a subgroup of Z_p^* with order q. Define the bit size of q as the *key size* of Algorithm 3.2.1. Protocol 3.2.2 explicitly assumes the cyclic group Z_p^*, and the bit size of p is the key size.

3.2.2 More Fortifications for OTD

Recall that our use of oblivious threshold decryption is for a host to obtain signals for its input bits. The decryption servers are agents residing on other hosts. For each bit, a host sends the encrypted signals of both 0 and 1 to enough agents, then through OTD it receives the decryption shares for

only one signal. For each host there should be only one OTD performed per input bit. Using the compiler certainly would detect a malicious host's attempt to decrypt the second encrypted signal of any of its input wires, but we want a more efficient way. Thus, we add a label to each encrypted signal identifying the wire with which the signal is associated. An agent maintains a record of wire ids for which it has served decryption requests. Upon receiving a new request, the agent first extracts the label (using the Label Extraction operation) of both encrypted signals, and only decrypts the signals if the wire that the label identifies has not been decrypted before.

The Shoup-Gennaro encryption scheme cryptographically bundles together a message and its label. Being able to break this tie and change the label without knowing the plaintext message or the private key means breaking the encryption scheme. Specifically, given a ciphertext $(c, L, u, \bar{u}, e, f)$, changing L with L' requires replacing $e = H_2(c, L, u, w, \bar{u}, \bar{w})$. As a result, $f = s + re$ also needs to be changed, where r and s are randomly selected when the message was encrypted. Knowing r immediately leads to the plaintext. If one can compute $f' = s + re'$, together with $f = s + re$ and the knowledge of e and e', one can uniquely determine s and r. This leads to breaking the encryption, which contradicts the proven security of the scheme. Alternatively, one may throw away u, $\bar{u}$, e, and f, and pick its own r and s. However, as $c = H_1(h^r) \oplus z$, coming up with a valid c that is consistent with the newly selected r requires the knowledge of z.

That said, label checking itself is not enough to prevent a malicious host from decrypting both signals of a wire. Obviously if enough agents are corrupted and collude with each other, they can decrypt any encryption. So the maximum number of corrupted agents, which is equal to the maximum number of host subsets that have one or more malicious hosts because each agent visits only one subset and no two subsets overlaps, must be less than the threshold for successful decryption. Still, a more subtle attack is possible even under this condition. The host can simply request two sets of agents to each conduct an OTD. If there is no overlap of honest agents between

the two sets, the host will not be caught. In light of this, we introduce *safety* s to be the upper bound of the number of host subsets the agent protocol can stand against that may contain one or more malicious hosts.

Definition 3.2.3 *Safety*

A mobile agent protocol is called ***s-safe*** *if it is secure against up to $s-1$ host subsets that may contain one or more malicious hosts.*

Theorem 3.2.4 *Let m be the threshold of the Shoup-Gennaro threshold decryption scheme and n be the total number of agents. The mobile agent protocol is s-safe, if $s \leq n$ and $m \geq \lceil \frac{n+s}{2} \rceil$.*

Proof: First, since $s \leq n$ and $m \geq \lceil \frac{n+s}{2} \rceil$, $m \geq s$ and so it is impossible for the adversary to control enough agents for threshold decryption. Next, we show that under the above conditions, for any two agent subsets S_1 and S_2 of size m, $S_1 \cap S_2$ contains at least one honest agent. S_1 and S_2 must each contain $m - |S_1 \cap S_2|$ unique agents, as well as $|S_1 \cap S_2|$ shared agents. So the total number of agents is at least $2(m - |S_1 \cap S_2|) + |S_1 \cap S_2| = 2m - |S_1 \cap S_2|$. Assume for the sake of contradiction that $S_1 \cap S_2$ contains no honest agent. Because there can be at most $s-1$ "bad host subsets", there can be at most $s-1$ bad agents. So $S_1 \cap S_2 \leq s-1$. Therefore, the total number of agents is at least

$$\begin{aligned} 2m - |S_1 \cap S_2| &\geq 2m - (s-1) \\ &\geq 2\lceil \tfrac{n+s}{2} \rceil - s + 1 \\ &\geq n + s - s + 1 \\ &= n + 1 \end{aligned}$$

which contradicts the fact we only have n agents. Hence, at least one agent in $|S_1 \cap S_2|$ must be honest. As a result, at least one common honest agent is needed for performing any two OTDs. Because an honest agent always conducts label checking before decrypting an encryption, and due

to the security of the SG scheme, it follows that no malicious host can successfully request two OTDs on encrypted signals of the same wire. ■

3.2.3 Protection against Other Attacks

Assuming secure OTD, the rest of the protocol is straightforward: The agents visit one host after another, and eventually return to the originator. Without help from the originator, there is little a malicious host can do to an agent due to the security of the encrypted circuit (Theorems 2.4.3, 2.8.3). It cannot analyze the circuit to find out the input from the agent (i.e., the agent state), nor can the host arbitrarily modify the output of the circuit, either the host output or the new agent state. As a last resort, the host can replace the whole agent with one created by itself. This gives the host the control over the computation that will be carried out in the subsequent hosts. Among other things, the malicious host can decode the signal output of the agent computation on the subsequent hosts. It can go further as to modify the circuit so that the output better reveals the inputs of the subsequent hosts. Serious as it might look, this attack turns out to be easy to defeat. The originator signs every encrypted circuit with unforgeable signature, and each host checks the signature before evaluating the circuit. Alternatively, the originator signs each encrypted signal, due to the 1-1 correspondence between an encrypted circuit and its signals. A host only sends out decryption requests with correctly signed encrypted signals, and uses share verification to check the honesty of a decryption share. Furthermore, this gives additional protection against chosen ciphertext attacks, as an honest agent only decrypts encrypted signals with the originator's signature. If the malicious host is more interested in causing a favorable final outcome from the agent, it may "hijack" the agent. That is, instead of forwarding the agent on, the host masquerades as subsequent hosts and evaluates the circuits on behalf of them with inputs to this host's favor. Like before, we add authentication in the OTD phase so that only the genuine host can decrypt its

input signals and hence evaluate its circuit.

The real problem occurs when the originator becomes malicious. Currently, the only effective way to defend against a malicious originator is asking for a zero-knowledge proof of the honesty of the encrypted circuits. It has to be zero-knowledge so that no signal and its semantics will be revealed. Currently, the inefficiency of zero-knowledge proofs prevents their use in practice. As a result, although our practical protocol is secure against a semi-honest originator, it offers no protection against a malicious originator.

3.3 Putting It Altogether

Combining Protocol 2.6.4 with all the enhancements discussed in the last section, we obtain a practical agent protocol that is ready for implementation. The tradeoff is that no protection is available against cheating from the originator, and there is no theoretic proof for the security against malicious hosts. Still, many practical concerns are addressed, and we begin with defining the security requirements for a real-world agent application.

3.3.1 Security Definitions

Unlike the first half of this work, in a practical setting, the desirable security may not necessarily be as "perfect" as its formally provable, simulation-based counterpart in theory. Instead of viewing the whole agent computation as secure function evaluation and trying to emulate an ideal process, we concentrate on the *privacy* and/or *integrity* of the three key objects: the agent code, the agent state, and the host input. Privacy is defined in terms of the information revealed about the object in question, and integrity is concerned about how the object can be modified in an unauthorized way.

As static code can be signed by the originator, and host input is handled locally by the execution

environment, integrity for these objects is easily obtained. For the integrity of the agent state, we would like to ensure that the state resulting from the computation on a host is the correct answer that would be obtained if all parties acted honestly. For example, a host arbitrarily modifying an agent's state is a violation of the agent's state integrity. As another example, a re-computation attack, where the host repeatedly computes the agent function with different inputs until a favorable outcome is obtained, is also a violation of agent state integrity.

The privacy requirements vary according to different applications. For most agent applications, the agent code is less privacy-critical than the agent state and the host input. It may be even desirable to have at least the purpose of the agent code public, so that the host knows what the agent is doing. As to how the agent does its job (the algorithm), the originator may want to keep it secret, which calls for some kind of code scrambling. We have already proved that encrypted circuits hide both the code and the data from its evaluator. However, from the host's point of view, it is certainly unsafe to execute unknown code. At the first glance, a zero-knowledge proof from the originator can help if efficiency is not concerned. But verifying that an algorithm implements a specific functionality is not even decidable. On the other hand, proving an encrypted circuit is correctly constructed from a given "plaintext" circuit is indeed in NP and henceforth has a ZK proof. So generally, if the host wants to verify that the scrambled agent code in the form of an encrypted circuit is safe, the originator has to reveal the corresponding "plaintext" circuit which gives away the algorithm of the agent code. The agent state and the host input are both input to the agent code. It should be noted that if an output somehow depends on the corresponding input, then it reveals information about the input. This is inevitable even in the ideal setting, so we cannot hope to have complete secrecy for the agent state and the host input with respect to a party that receives output from the agent code. However, we do want the party to deduce nothing more than the legal output implies, and for those not supposed to receive output, complete secrecy

is a realistic goal.

In summary, we define four variations of privacy, with the amount of privacy increasing as the list progresses.

Definition 3.3.1 *The amount of privacy afforded can be characterized in one of the following four ways.*

- **No Privacy***: All information about the object in question is revealed.*
- **Limited Privacy***: The amount of information revealed by some data about the object is limited only by the size of the data.*
- **Verifiable Privacy***: The amount and specifics of information revealed can be analyzed and verified. Specifically, if the object in question is the input to some function, this means that no information about the input can be learned other than what follows from the output.*
- **Complete Privacy***: No information is revealed.*

3.3.2 The Practical Protocol for Secure Mobile Agent Computation

We modify and enhance Protocol 2.6.4 as follows. First, we replace the two building blocks of concurrent oblivious threshold decryption with the more efficient algorithms, namely the Shoup-Gennaro threshold cryptosystem(Algorithm 3.2.1) and the Bellare-Micali oblivious transfer scheme (Protocol 3.2.2). Next, we add protections against host attacks. Each encrypted signal is bundled with a label uniquely identifying the wire of that signal. Given an OTD request, an agent checks its label and makes sure only one decryption per wire. We also enforce the concept of safety and select the appropriate threshold value according to Theorem 3.2.4. The originator signs the encrypted signals, and each host checks the signature before decrypting them. Finally, we remove from Protocol 2.6.4 the steps of host input notification and encrypting the agent state with the

next host's public key, both of whom are for security proof only. Again, this protocol assumes static number of agents, and static agent itinerary with no overlap between any two agents.

Protocol 3.3.2 *Secure Mobile Agent Computation (III)*[4]

Assume the originator and all the hosts share the following information. For threshold decryption, a cyclic group G of prime order q. For oblivious transfer, assume a prime number p, a generator g, and $C \in Z_p^*$ whose discrete log is unknown to everyone.

1. *O determines the hosts it wants to send agents to. Suppose there are ℓ hosts. O divides the hosts into n disjoint subsets, where n can be picked based on various criteria, such as the level of parallelism. Next, O invokes Key Generation of the Shoup-Gennaro scheme (Algorithm 3.2.1) to obtain a public key PK, a verification key VK, and n secret key shares SK_i for $i = 1, \ldots, n$ with threshold $m \geq \lceil \frac{n+s}{2} \rceil$, where s is the desired safety of the protocol.*

2. *For each host $H_{(i,j)}$, O creates an encrypted circuit $(\mathcal{C}^{(i,j)}, \mathcal{L}^{(i,j)}, \mathcal{K}^{(i,j)}, \mathcal{U}^{(i,j)})$ using the enhanced BMR construction, to compute function $f^{(i,j)}$. Then, O invokes Encryption of the SG scheme to encrypt $\mathcal{K}^{(i,j)}$, the signals for the host's input. For each signal $K_{k,b}^{(i,j)}$, where $k = 1, \ldots, n_y$ and $b \in \{0,1\}$, set its label to (i,j,k). O signs each encrypted signal with a publicly verifiable signature. Denote the signed encrypted signals as $\overline{\mathcal{K}}^{(i,j)} = ((\overline{K}_{1,0}^{(i,j)}, \overline{K}_{1,1}^{(i,j)}), \ldots, (\overline{K}_{n_y,0}^{(i,j)}, \overline{K}_{n_y,1}^{(i,j)}))$.*

3. *Next, O creates n mobile agents. For agent $\mathcal{MA}_i$, $i = 1, \ldots, n$, its code includes the encrypted circuits $\mathcal{C}^{(i,j)}$, the encrypted signals $\overline{\mathcal{K}}^{(i,j)}$, and $\mathcal{U}_z^{(i,j)} = ((U_{1,0}^{(i,j)}, U_{1,1}^{(i,j)}), \ldots, (U_{n_z,0}^{(i,j)}, U_{n_z,1}^{(i,j)}))$, the signals of the output wires corresponding to the local output to the host, for all the hosts in subset i. O also sets $L_k'^{(i,1)} = L_{k,x_{(i,0),k}}^{(i,1)}$ where $x_{(i,0),k}$ is the k-th bit of $x_{(i,0)}$, for $k = 1, \ldots, n_x$, as the initial state of $\mathcal{MA}_i$. Following our convention, denote this list of signals by $\mathcal{L}'^{(i,1)}$. Furthermore, each agent gets a secret key SK_i.*

[4]Definitions of notations appear in Section 2.6.2.

4. *Finally, O sends out $\mathcal{MA}_i$ with its initial state to $H_{(i,1)}$, the first host of subset i, for $i = 1, \ldots, n$.*

5. *Every host $H_{(i,j)}$, upon receiving agent $\mathcal{MA}_i$, verifies the signatures on the encrypted signals $\overline{\mathcal{K}}^{(i,j)}$ and aborts the execution once a verification fails.*

6. *If all verifications have passed, $H_{(i,j)}$ sends the encrypted signals $\overline{\mathcal{K}}^{(i,j)}$ to m agents including $\mathcal{MA}_i$, asking for decryption. In particular, this is done through n_y requests, where each request contains a pair of encrypted signals $\overline{K}_{k,0}^{(i,j)}$ and $\overline{K}_{k,1}^{(i,j)}$, for $k = 1, \ldots, n_y$. $H_{(i,j)}$ signs each request with its publicly verifiable signature.*

7. *Once receiving a decryption request, an agent invokes Label Extraction and checks the following conditions. First, the labels of the two encrypted signals must be the same. Second, the label has not appeared in this agent's decryption history. Third, the encrypted signals are correctly signed by the originator, and for a label (i, j, k) the request bears host $H_{(i,j)}$'s signature. If all conditions are true, the agent obtains a decryption share for each of the two encrypted signals by applying the* Decryption *operation of the SG scheme with its secret key share. Otherwise, the request is ignored.*

8. *When an agent finishes its decryption, it signals the requesting host $H_{(i,j)}$. Then the two execute the oblivious transfer protocol (Protocol 3.2.2) so that the host receives the decryption share for either $\overline{K}_{k,0}^{(i,j)}$ or $\overline{K}_{k,1}^{(i,j)}$, but not both.*

9. *After receiving a share, $H_{(i,j)}$ verifies the correctness of the share against the ciphertext with the Share Verification algorithm. An invalid share is discarded and the host tries with another agent. After collecting m valid shares, $H_{(i,j)}$ combines them into a valid decryption, which is the signal representing its input bit. After n_y successful OTDs, the host obtains the signals for its input. Denote this list of host input signals as $\mathcal{K}'^{(i,j)}$.*

10. *$H_{(i,j)}$ evaluates encrypted circuit $\mathcal{C}^{(i,j)}$ using the input signals $\mathcal{K}'^{(i,j)}$ and the current agent state $\mathcal{L}'^{(i,j)}$, recovers the value of its own private output using the corresponding $\mathcal{U}_z^{(i,j)}$ vector, and sends the agent (consisting of the encrypted circuits and signals for the following hosts, as well as the new agent state, in the form of signals) to the next host in the subset. If this is the last host in the subset, then the agent is returned back to the originator. If the transfer was to a different host, that host repeats steps 5 – 10.*

11. *After all n agents return back to O with their final states. O recovers the values they represent and uses function g to combine the results into the final result.*

3.3.3 Security Analysis

We argue the security — privacy and integrity — provided by Protocol 3.3.2. (The details of each claim are discussed in the previous sections.) Suppose that the originator is honest or semi-honest, and the total number of host subset that may contain at least one malicious host is less than s, the safety value. Furthermore, suppose s, n, and m satisfy the conditions listed in Theorem 3.2.4 (i.e., $s \leq n$ and $m \geq \lceil \frac{n+s}{2} \rceil$). The malicious hosts can collude arbitrarily, but all are limited to polynomial-time algorithms.

Agent code: The encrypted circuits perfectly hides the internal details of the circuits assuming secure OTD. As the originator is trusted, this *complete privacy* is acceptable. Complete privacy makes it impossible for most meaningful attempts to modify the encrypted circuits. Unforgeable signatures on the encrypted signals prevent replacing an entire circuit.

Agent state: Both the privacy and the integrity of the agent state depend on the security of the encrypted circuits, which in turn depends on the security of the oblivious threshold decryptions. As long as the OTD is secure, Theorem 2.4.3 asserts that the encrypted circuits reveal no more information than what the output contains (i.e., *verifiable privacy*). We lists common attacks on

OTD and their countermeasures.

- Corrupting enough agents to decrypt the encrypted signals. This is prevented by the fact that each agent visits a subset of hosts not overlapping with any other agent's itinerary. As a result, a malicious host can control at most one agent and the number of corrupted agents is equal to the number of host subsets containing malicious hosts, which is at most $s-1$. Because of our restriction on s, n, and m, $m \geq s$. So the malicious hosts are forced to go through OTD.

- Cheating in the requests for OTD.

  - Decrypting both signals for an input wire. With the threshold set to an appropriate value in regard to the safety, any successful decryption involves at least one honest agent. The honest agent checks the label on each encryption which is cryptographically bundled with the encryption, against its decryption history, and never decrypt twice for the same wire.

  - Decrypting an encryption other than the encrypted signals. This is a chosen ciphertext attack. Because of the CCA security of the threshold scheme, it does not help attacking any encrypted signals even if this chosen encryption is decrypted. Meanwhile, checking the originator's signature on a requested encryption makes this impossible.

- Cheating in the receiving with oblivious transfer. The security of the oblivious transfer protocol ensures the host gets the decryption share for only one encryption.

- Masquerading as another host. This is an attack on the integrity of the agent state. It is defeated by the authentication of the requesting host.

- Modifying the agent state arbitrarily. This requires guessing the signals other than those obtained from a valid evaluation of the encrypted circuit, or in other words, the off-path

output signals. The probability for a successful guess is only $\frac{1}{2^k}$ for a $(k+1)$-bit signal including the tag bit.

Thus, the only possible information leakage about the agent state is through the output that the host receives from the circuit. This gives us *verifiable privacy.* If there is no output to the host, then *complete privacy* is attained for the agent state. The integrity of the agent state is protected except for the negligible probability that the host successfully predicts the off-path signals.

Host data: Like the agent state, since the host data is also the input to the encrypted circuit whose output includes the updated agent state, only *verifiable privacy* can be and is achieved with regard to the (semi-honest) originator and any subsequent host who also receives output from the circuit. Notice that this is the only possible way that information may be revealed about a host's data. In particular, OTD does not reveal any information about the requesting host's data, due to the security of the OT protocol. As a corollary, no information of the host data is leaked to a subsequent host that does not receive output from the encrypted circuit. At the first glance, since the host data never leaves the host itself, integrity is automatically guaranteed. However, the issue is complicated by the use of OTD. Now the definition of host data integrity extends to getting the correct signals for the data. This is protected as share verification can detect invalid decryption shares for an encryption.

The following table summarizes the security properties of Protocol 3.3.2 when the originator is semi-honest but the hosts may be malicious and can collude with each other.

When the originator becomes malicious, the whole picture changes dramatically. The agent code and data are not our objects of protection any more, but there is little effective defense for the host data in the absence of a ZK proof enforcing the integrity of the encrypted circuits. The circuits may perform any arbitrary function and the host has no knowledge of the actual internal algorithm of the circuit. Therefore, only *limited privacy* is attainable for the host data, which is not

	Semi-honest Originator	Malicious Hosts
Agent Code	N/A	Complete Privacy, Provable Integrity
Agent Data	N/A	Verifiable Privacy (if hosts receive outputs) Complete Privacy (if hosts do not receive outputs) Provable Integrity
Host Data	Verifiable Privacy Provable Integrity	Verifiable Privacy (if other hosts receive outputs) Complete Privacy (if other hosts do not receive outputs) Provable Integrity

Table 3.1: Privacy and integrity achieved by Protocol 3.3.2.

very satisfactory. In conclusion, Protocol 3.3.2 is not geared toward protecting against a malicious originator.

CHAPTER 4

EXPERIMENTAL RESULTS

This chapter presents the implementation of Protocol 3.3.2 on the mobile agent platform JADE version 3.0b1. JADE (Java Agent Development Framework) [4] is a software platform for developing multi-agent applications conforming to FIPA standards for intelligent agents. The FIPA standards [1] specify, among other things, the ways for "interaction of heterogeneous agents and the services that they can represent". The specifications can be grouped into five main categories: Agent Communication, Agent Management, Agent Message Transport, Abstract Architecture, and Applications. A list of agent platforms that conforms to these specifications can be found at the FIPA web site.

Furthermore, we implemented the ACCK protocol (Protocol 5) on the same system. The purpose is to measure and compare the performance of Protocol 3.3.2 and the ACCK protocol. To the best of our knowledge, currently there is no other published implementation of the ACCK protocol. Our results are the first study of the actual performance of the SFE-based, secure agent computation protocols.

4.1 Implementation Overview

4.1.1 The JADE System

JADE is a Java-based development platform as well as execution environment for multi-agent systems. While not designed specifically for mobile agents, the JADE platform can be distributed across multiple hosts and does support agent mobility. An agent *container* provides the execution environment on a machine. There can be multiple containers on the same physical computer,

and each container can host multiple agents. The *main container* serves as the controller of the platform. A GUI interface is provided for the main container to manage (create, delete, etc.) all the containers and agents, locally as well as remotely in the system. JADE provides FIPA-compliant system services such as naming, yellow-page service, message transport and passing, agent communication, and so on.

From a programmer's point of view [2], JADE is implemented fully in Java, and it exports a group of Java packages for developing agent systems on the platform. We highlight the programming features of JADE which are essential to our design and implementation of the agent protocols.

- **Agents and Containers**. Each agent container is a Java virtual machine, and each agent is a Java thread. The system-provided *Agent* class is the base class for all agents. Specific agents are implemented by extending the Agent class. Agents are created by containers, and both can be created either programmatically or from the command line. An agent platform always starts with the creation of a main container, after which more containers can be created and joined with the main container. There can be multiple main containers and hence multiple platforms, too. However, as of JADE version 3.0b1, inter-platform agent mobility is not supported. Each agent is given an identifier when created. It is a concatenation of the user specified name and the container where the agent was created. Two system agents reside in the main container. The Agent Management System (AMS) agent provides white-page and life-cycle service and maintains a directory of agent identifiers and agent state. The Directory Facilitator (DF) agent provides yellow-page service in the platform.

- **Agent behaviors**. The activities of an agent is implemented through the class *Behaviour*. An agent can have multiple behaviors, and the main function of a behavior (named *action*) can be executed multiple times. There is a cooperative scheduler hidden from the programmer that schedules the execution of the behaviors in a round-robin fashion. Since there is no

preemption, every behavior has to explicitly yield its control by falling to the end of its *action* function. Once the *action* function returns, the scheduler queries the *done* function of the behavior. If the behavior is done, it is removed from the scheduler's queue of available behaviors. Otherwise, it will be rescheduled to run. A behavior can block itself. Once blocked, the behavior will no longer be scheduled for execution until being activated again, such as by the event of message arrival. The agent terminates when all its behaviors are done.

- **Agent communication**. Agent messages are defined by the *ACLMessage* class, which contains a set of attributes defined by FIPA. Message transmission is mostly transparent to the programmer. In most cases, what the sender needs to do is to specify the message type using one of the constants defined by FIPA, the receiver's agent ID, and the content of the message. As a corollary, one does not have to worry about the location of the receiver. The underlying system figures that out, even when the receiver agent may move from one container to another. We also found it very convenient that the content of an ACLMessage can be either an array of raw bytes or a Java object. This allows maximum flexibility about what data can be exchanged. Each agent maintains a message queue global to all its behaviors. Message filtering is provided for a behavior to receive specific messages. Receiving a message can be blocking and non-blocking. However, if called within a behavior, the blocking receive blocks the whole agent (i.e., the whole java thread), including all other behaviors. This could potentially degrade the performance of agents with parallel behaviors, including our application.

- **Agent migration**. An agent migrates from one container to another by calling the *doMove* function within one of its behaviors. This function stops the scheduler, wraps up the whole agent, which is a Java object, and sends the agent to the destination container. Once the

agent arrives at the new container, its scheduler is resumed, and all the ready behaviors are scheduled to run like before. Therefore, JADE supports only limited code mobility, in that the run-time state of the behaviors are not saved and transfered with the agent. It is not possible to stop at the middle of the behavior function and resume execution right from that point after migration.

4.1.2 Code Design

Both the ACCK protocol and our practical result (Protocol 3.3.2) are implemented on JADE version 3.0b1 with Java 2 Standard Edition version 1.4. Notice that the only active entity in JADE is the agent. Containers are just the run-time environment. Hence, all the players in both protocols are implemented as agents. The originator is a static agent always staying at the home container. Once started, the originator creates the mobile agents and sends them to the containers on remote hosts. After returning to the home container, the mobile agents send their final states to the originator, which then outputs the final result.

The tricky question is how the host is implemented. Throughout this work we give an active role to the host to better reflect its ability to control the agent. In fact, the agent is treated mostly as the carrier of code and data, except for the role in oblivious threshold decryption. On the other hand, the JADE container is just an environment. So there has to be a *host* agent that receive and evaluate the mobile agents. We simplified this by implementing all the host's activities in the mobile agent class. In other words, upon arriving at a new host, the mobile agent decrypts signals for the host's input through OTD, and then evaluates the encrypted circuit with the host's input and its current state. Finally, the agent calls *doMove* to migrate to the next host. Furthermore we include the host's input in the agents as another simplification. This is acceptable as the interaction for getting host input will vary across different applications.

For the other role of mobile agents as decryptors in OTD, we chose to separate it from the main activity described above in order to achieve maximum possible concurrency. Ideally, serving other hosts' decryption requests should be running in parallel with this agent's own computation. Recall that the scheduler for agent behaviors is non-preemptive, and waiting for a message in the blocking mode actually makes the whole agent stall. As a result, if an agent starts listening for OTD requests from other hosts, its own computation, which is implemented in another behavior, would also be blocked. Therefore, we implemented the decryptor behavior in another agent called *helper*. Each helper is bundled with a mobile agent, and migrates together with the agent. Because each agent is a separate Java thread, the helper and the agent do not interfere with each other when one is blocked for receiving messages, and preemptive scheduling applies between these two agents, provided by the underlying JVM. In summary, the mobile agent class is responsible for decrypting input signals for the current host (with the help from remote helpers) and evaluating the encrypted circuit, while the helper class is responsible for decrypting for other, remote mobile agents. When the mobile agent migrates, it tells the associated helper to migrate, too. Figure 4.1 depicts the conceptual model of our implementation of the secure agent computation protocol.

Most of the cryptographic tools used by our protocol do not have publicly available implementations. We implemented the encrypted circuit, the Shoup-Gennaro threshold decryption scheme, and the Bellare-Micali oblivious transfer protocol, using Java's secure pseudorandom number generator and the *BigInteger* class as well as secure hash functions provided by the Cryptix JCE package [3]. We pre-computed the underlying cyclic groups and the value C required by the threshold decryption and the oblivious transfer protocol[1]. We created a general interface to transform a plaintext circuit consisting of any gates with up to 3 inputs, into an encrypted circuit using the enhanced BMR construction. We used SHA1 as the secure hash functions in the SG scheme, with enhancement

[1]In the real world, the cyclic groups and C should be generated by an honest, general-purpose third party, and the generation can be totally independent from the protocol.

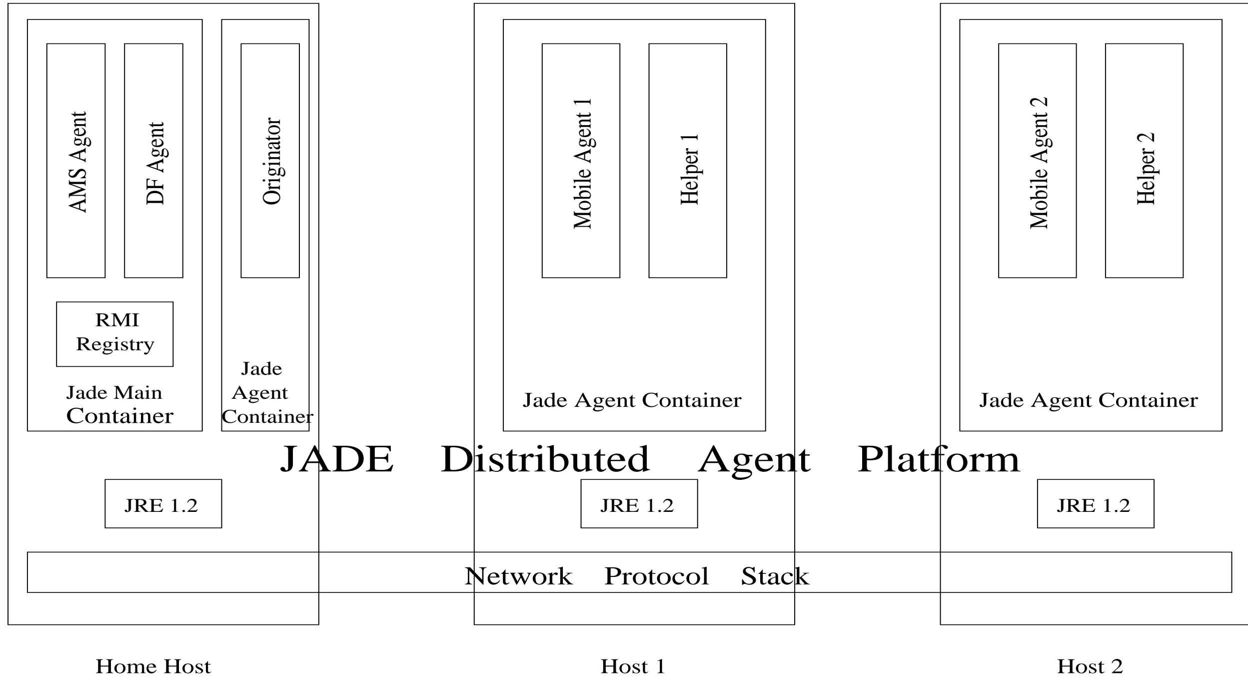

Figure 4.1: Implementation of the secure mobile agent computation protocol

suggested by [14]. For the digital signature, we chose RSA with SHA1, provided by Cryptix.

We are not aware of a general-purpose tool to generate boolean circuits for arbitrary functions. This could be an interesting future project and has applications of its own. For this performance study, we hand-picked the function of finding the larger of two integers as the agent function, and manually built a circuit for this function. Such a circuit is the basis functionality of a bidding agent and has appeared before in the theoretical part of this work. Our circuit design is exactly as depicted in Figure 2.1. This MAX circuit has a uniform structure even as the number of input bits varies, so generation of circuits for different input sizes can be automated. The originator sets the initial states of all the agents to be an integer of its choice. At each host, the mobile agent gets another number from the host, and through the circuit computes the larger number. This number is outputed to the host and set to be the new agent state. In the end, the originator compare the final states of all the agents and obtains the maximum all the integers.

For the ACCK protocol, the trusted party is implemented as another agent. For simplicity,

the trusted party resides at the home container just like the originator. There is only one mobile agent in the system, which visits all hosts and brings back the maximum number. We used RSA as the public-key encryption of the trusted party. Due to the fact that each signal is a short binary string, we chose RSA with OAEP using SHA1 as the hash function. We left out the use of the symmetric encryption, but "chained" together neighboring circuits so that the state output signals of the previous circuit can be directly fed into the next circuit as input. The same technique is used in our multi-agent protocol. In fact, both implementations shared the same group of Java classes for encrypted circuits. Table 4.1 lists our choices for the cryptographic tools required by the two protocols.

Encrypted circuits	The enhanced BMR construction, our implementation
Threshold decryption	Algorithm 3.2.1, our implementation
Oblivious transfer	Protocol 3.2.2, our implementation
Digital signature	RSA with SHA1, provided by Cryptix JCE
Public-key encryption	RSA with OAEP using SHA1, provided by Cryptix JCE
Pseudorandom number generator	Based on SHA1, provided by Sun JCE

Table 4.1: Cryptographic tools used by our implementation.

4.2 Test Configurations

Our test environment is a local network with 7 Pentium 4 2GHz machines, all running Red Hat Linux 8.0. The JADE system used is version 3.0b1, and the underlying Java virtual machine is Sun's Java 2 SDK version 1.4.0_02. We used a snapshot version of the Cryptix JCE package dated 2003/02/17.

We measured the time taken by the various stages of the two protocols. There are two types of timer we could use. One measures the actual elapsed time, called the *real time*, while the other reports only the CPU time spent on the tested program, known as the *user time*. In distributed computing, usually a significant amount of time is spent on communication, which is reflected in

the real time. On the other hand, user time provides insight on the computational overhead of the protocols. Also keep in mind that the real time is bound to vary as the system load changes. (Our testing environment is a relatively quiet setting.) In order to measure the real time, all machines had their clocks synchronized. We used Java function *System.currentTimeMillis* to record the real time and the native C function *getrusage* to measure the user time.

Of the 7 machines, one is the home host and the other 6 are the remote hosts. As depicted in Figure 4.1, both the main container and the home container reside on the home host. There is one container on each of the remote host. For Protocol 3.3.2, we fixed the number of agents to 4. Two mobile agents each visits two hosts, while the other two only visit one host each. With 4 agents, according to Theorem 3.2.4 the threshold for decryption can only be either 3 or 4, with the corresponding safety being 2 and 3 respectively. In other words, if at most 2 host subsets have malicious hosts, the 4-agent protocol is secure.

We also fixed the length of the signals to be 256-bit excluding the tag bit. Assuming the rest of the protocol is secure, there is no more efficient way of breaking an encrypted circuit than brute-forcefully searching for the random signals. Thus, 256 bits is a fairly strong setting. The other primitives (threshold decryption, oblivious transfer, public-key encryption, and digital signature) are based on either Diffie-Hellman or RSA. With the advance of cryptanalysis, the acceptable key size for both problems is continuously being raised higher. Currently, 1024-bit keys are recommended as minimally sufficient for Diffie-Hellman and RSA if long-term security is important [63]. In our experiments, the key size for of all these primitives was set to 1024 bits. For testing purpose, we also tried 512-bit keys for the multi-agent protocol. Lastly, we had two variants of the MAX circuits, one with 16-bit inputs and the other with 32-bit inputs. The following table summarizes the parameters we have experimented.

Encrypted circuit signal size	256 bits
Public key size	1024 bits, 512 bits
Threshold value	3 (for safety = 2), 4 (for safety = 3)
Circuit	16-bit MAX, 32-bit MAX
Number of agents	4 in Protocol 3.3.2, 1 in the ACCK protocol
Number of hosts	6

Table 4.2: Test parameters.

4.3 Performance Results

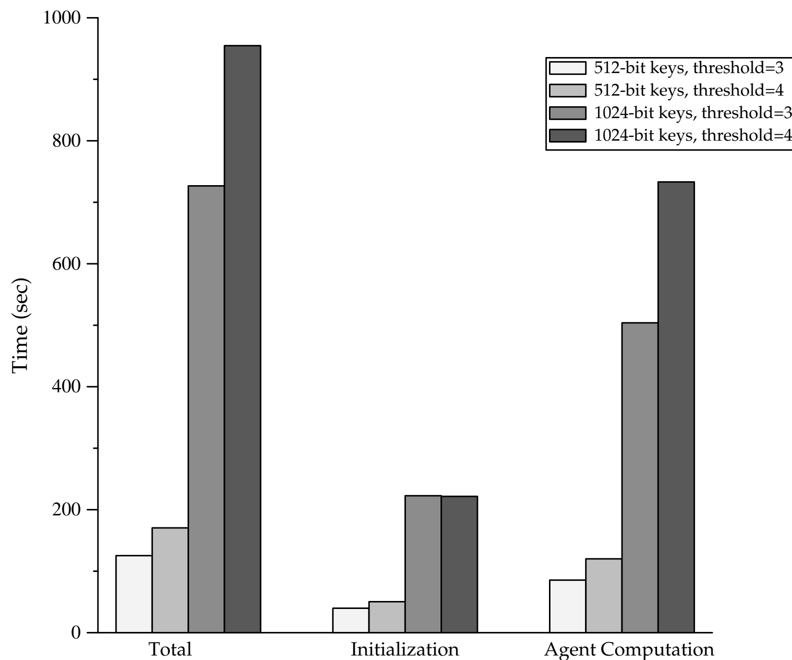

Figure 4.2: Overall running time of Protocol 3.3.2 with 32-bit MAX circuit

Figure 4.2 shows the overall performance of Protocol 3.3.2 with various configurations. The time measured is the elapsed real time. Again, the size of the signals is fixed to 256 bits, and there are 6 hosts and 4 agents in all configurations. The total time is the sum of the initialization time and the agent computation time. Initialization spans from the starting of the protocol up to the point right after the agents are sent out. During this phase, only the originator is active, while all the hosts are quietly waiting for an agent to come. Specifically, the originator locates and partitions the hosts. It also generates the encrypted circuits, the keys for the threshold decryption scheme,

and encrypts the host input signals. Then 4 mobile agents and 4 helper agents as discussed above are created and dispatched out to the hosts. The agent computation time covers from the end of initialization to the end of the protocol when the originator obtains the final result after all the mobile agents and helpers return home.

Security improves with larger threshold values or key sizes, but this also decreases the performance. With a moderate key size like 512 bits, increasing the threshold seems not to be expensive. When it comes to a longer key length like 1024 bits, adjusting the threshold may cause larger impact on performance, so real-world applications should carefully choose the smallest threshold value that makes sense. Likewise, if long-term security is not essential, a smaller key size should be worthwhile. Initialization time accounts for from about 1/3 to 1/4 of the total time among different configurations. Thus, the originator does not need a high-performance communication line, but does require a fairly large amount of computational power.

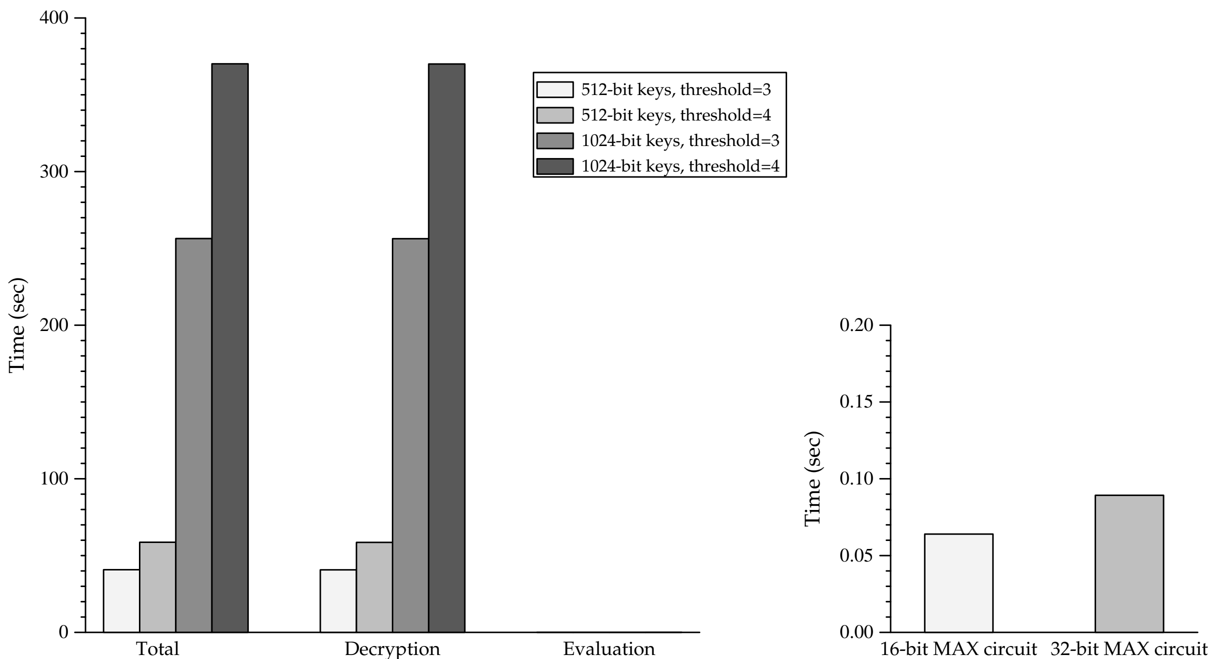

Figure 4.3: Average time an agent spends on each host in Protocol 3.3.2 with 32-bit MAX circuit, and the circuit evaluation time

The left hand side of Figure 4.3 shows the average time an agent spent on each host in our tests.

This is further broken down to the time for decrypting signals for the host inputs, and the time for evaluating the encrypted MAX circuit. Surprisingly, evaluating the MAX circuit is so fast (the right hand side of Figure 4.3) that it is totally negligible compared to the time for decrypting the signals. This finding actually brings out two good points about our protocol. For one thing, it takes less than 0.1 second to evaluate a 32-bit MAX encrypted circuit, which has 488 gates including the splitter gates. Because the evaluation of any gate in an encrypted circuit with the same number of input wires is identical regardless of the gate functionality, the evaluation time of the whole circuit is linear in the number of gates in the circuit. Thus, we can foresee that evaluating an encrypted circuit of moderate size should take acceptable amount of time in practice. For another, although in our tests, the signal decryption time is huge compared to circuit evaluation, it is the circuit evaluation that varies according to the agent function. The communication complexity of both our theoretical and practical results is not affected by the size of the encrypted circuit. Even better, for each host, the communication and computation overhead of signal decryption remains the same as long as both the bit size of the input and the threshold value do not change, no matter how large the circuit is or how many hosts are in the system. In other words, a 100-gate circuit and a 10000-gate both with 32-bit inputs would take exactly the same amount of OTDs in the same threshold setting, and hence the same amount of time to decrypt their input signals. This is a distinct feature resulting from the use of the encrypted circuit[2].

There are essentially two factors that affect the performance of signal decryption. Obviously, the larger the threshold value, the longer for decrypting a signal. Another factor in the performance is the number of concurrent OTDs in the system. For example, consider the case of 4 agents and the threshold being 4. This means decrypting a signal for any host requires the work of all 4 agents, one on the host itself and three remotely. In addition, it is likely that 4 hosts are doing signal

[2]There are other general-purpose SFE protocols in which the number of communication rounds is proportional to the size of the function evaluated.

decryption simultaneously. As a result, ideally a total of 12 instances of remote decryption and oblivious transfer are conducted concurrently in the system per host input bit. However, in our actual implementation, restricted by the non-preemptive scheduler inside the JADE agent, each helper (recall that the decrypting service of an agent is separated from the agent and implemented as a helper agent) cannot serve the requests from 3 remote hosts at the same time, but has to do it in a sequential manner[3]. Such amount of work and the delay due to the lack of true parallelism caused the relatively large overhead for signal decryption at every host. Despite this, the overall agent computation time is saved by the use of multi-agents, which does bring true parallelism to the whole computation.

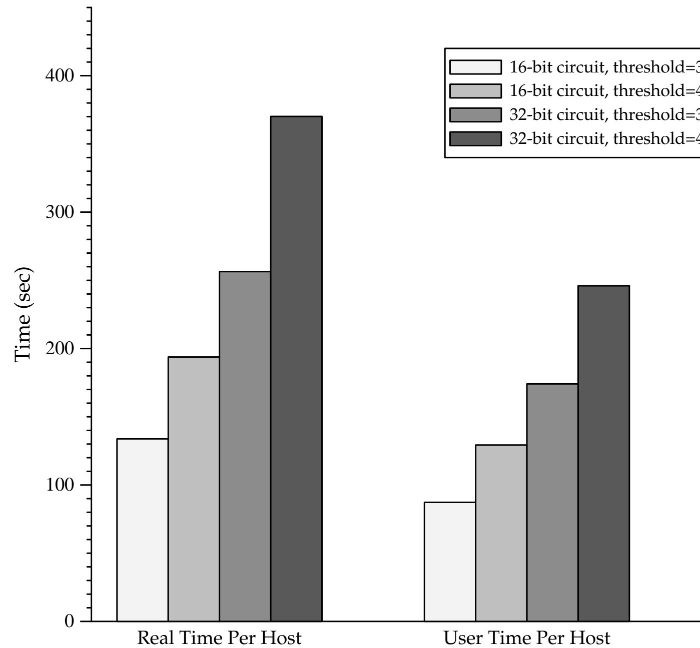

Figure 4.4: Average computation overhead per host

Figure 4.4 compares, on average for each host, the CPU time consumed by an agent and its associated helper versus the total real time spent by the agent. The key size is fixed to 1024 bits. Recall that the "mobile agent" computes the agent function, while the "helper agent" serves as a

[3]It is possible of course to build in explicit cooperative multitasking into the helper agent, but the effort of this may be largely offset by the complexity induced. We believe it is more of a platform issue.

decryption server for other, remote agents. The shown user time is the sum of the time from both the mobile agent and the helper on the same host. Approximately 2/3 of the real time is spent on computation (for this agent and for helping other agents), while communication takes the remaining 1/3 of the time. To some extent, this reflects the importance of limiting interactions among the agents, particularly when the underlying platform does not provide full support for parallelism.

Finally, we measured the performance of the ACCK protocol with the same agent function. As mentioned before, in this case there is a single agent visiting 6 hosts. The trusted party is implemented as another agent sharing the same container as the originator. We used RSA with 1024-bit keys. The following table summarizes our findings. Unless specified otherwise, all time shown are real time. Without doubt, the ACCK protocol is simpler and more efficient than Protocol 3.3.2, but at the cost of much stronger trust assumptions and thus a weaker level of security. Still, the test results show that the ACCK protocol is quite efficient. This is mainly due to its use of a trusted party as the dedicated decryption server and a fast RSA implementation (although the security of the implementation might be questionable in the theoretical sense). This suggests a potential way to improve our multi-agent protocol is looking for faster implementations of the cryptographic tools used by the protocol.

	Protocol Total	Agent Computation	Time Per Host	Trusted Party User Time
16-bit MAX circuit	11.187 sec	5.639 sec	0.655 sec	2.163 sec
32-bit MAX circuit	17.921 sec	8.938 sec	1.149 sec	4.316 sec

Table 4.3: Performance of the ACCK protocol.

As a mobile agent carries the encrypted circuits for all the hosts it is going to visit, the size of the mobile agent depends on the number of hosts in its itinerary. In addition, there is overhead due to the underlying agent platform. Table 4.4 reports, under our test configurations, the initial sizes of the 4 mobile agents in the multi-agent protocol and the single agent in the ACCK protocol. This

is the size of each agent when it is sent out by the originator. After visiting a host, the agent (in both protocols) discards the encrypted circuit for that host before migrating to the next one. Thus the initial size is the maximum size of the agent during its lifetime. The choice of the threshold does not affect the agent size, but because the size of the encryption in the Shoup-Gennaro scheme (Algorithm 3.2.1) varies due to the random values used, there is slight difference between the sizes of the agents with the same number of hosts to visit. Finally, all the helper agents have the same size.

	Agent 1 (visiting 2 hosts)	Agent 2 (2 hosts)	Agent 3 (1 host)	Agent 4 (1 host)	Helper
Protocol 3.3.2 (threshold = 3)	319688	319688	165170	165173	9942
Protocol 3.3.2 (threshold = 4)	319692	319688	165177	165177	9942
ACCK protocol (Single agent visiting 6 hosts)	638191	N/A	N/A	N/A	N/A

Table 4.4: The initial sizes of the mobile agents in number of bytes, with 32-bit MAX circuit, 256-bit signal size, and 1024-bit keys.

CHAPTER 5

CONCLUSION

Mobile agents is a relatively new paradigm of distributed computing. The unique combination of autonomous agents and code mobility brings promising benefits to applications such as e-commerce. This same feature also poses serious security issues, especially for the mobile agent. The hosting environment's full control over the agent makes effective protection of the agent's privacy and integrity difficult to achieve.

This research builds upon a new direction of protecting mobile agents through cryptographic protocols. Prior work has obtained reasonable results but with strong trust assumptions. Our results remove trust requirements for any particular entity, and instead base security on enough participants being honest, a minimum and easy condition in the real world. We extend the prior work by defining a model of secure mobile agent computation, which embodies the latest development in the theory of cryptography such as the concept of universal composability, and addresses security concerns for all entities in the mobile agent paradigm. Based on this model, we present a provably secure protocol utilizing multi-agents against static, semi-honest or malicious adversaries. To our knowledge, this is the first protocol of its kind. Our work on the practical front results in an efficient protocol that is ready to be applied to real world applications. Although not as complete a solution as its theoretical counterpart, this protocol still offers solid security measures against most attacks by malicious hosts on mobile agents with the same minimum trust assumption. The experimental study demonstrates the practicality of this protocol as well as the application of cryptographic protocols to mobile agents in general. Considering the strong security offered by this direction, our results have the potential for a significant impact on the mobile agent field.

This work also contains intensive study of some fundamental cryptographic primitives, one of which is Yao's encrypted circuit. By correcting a security flaw in a former construction method, we provide arguably the first provably secure implementation of the encrypted circuit. Our study on oblivious transfer and threshold decryption in theory and in real systems brings new insights to these fundamental tools, such as applying universal composability to threshold decryption, combining the two primitives into oblivious threshold decryption, and obtaining experimental results that reflect the performance of these primitives.

5.1 Open Problems

Quite a few interesting problems are left for future research. Protocol 2.6.4 and Protocol 2.8.4 are proven to be secure against static adversaries. Naturally the next step is moving to adaptive adversaries. An adaptive adversary may corrupt parties at any time of its choice, including after the protocol is over. Corrupting parties during the protocol execution gives the adversary significant information, particularly when a party is not supposed to erase its internal state pertaining to the protocol in question. For instance, consider the simulation of encrypted circuits by fake circuits. As we know, if the output of the circuit is unknown, the fake circuit can be any randomly generated circuit. But once the output is known, an evaluation path must be selected and the truth table entries along this path have to be consistent in order to produce the known output. Suppose a real adversary $\mathcal{A}$ attacking the agent protocol does not corrupt any host until after the originator sends out all mobile agents. To simulate this case, an ideal adversary $\mathcal{S}$ has to construct fake circuits in the beginning when agents are sent out. At that moment, no host is corrupted, and so $\mathcal{S}$ does not have access to any host in the ideal model. Therefore, $\mathcal{S}$ does not know any output, and the best $\mathcal{S}$ can do is to create random fake circuits with no evaluation path. But when $\mathcal{A}$ later corrupts a host, it sees the host's output, and to $\mathcal{A}$ a randomly generated fake circuit with no evaluation

path leading to the same output is no longer indistinguishable from the actual encrypted circuit. Generally, to defend against adaptive adversaries, there needs to be a way for the simulator to generate something that is consistent with information discovered later when a party is corrupted. One solution is the so-called “non-binding” encryption [21]. It scrambles data just like traditional encryption, but the same ciphertext can be opened to different plaintexts as needed. As this example shows, more investigation is certainly needed to make the current protocols secure against adaptive adversaries.

Our theoretical result against malicious adversaries is not suitable for implementation because of the use of the UC compiler. Currently we do not have an alternative for the compiler. Avoiding the compiler would bring huge improvement in the round complexity and computational overhead, and may lead to a provably secure and practical solution to mobile agent security. As mentioned before, we restrict the itinerary of each agent not to overlap with any other agent’s itinerary. This could undermine the fault-tolerance of the system. Consequently, an interesting topic is improving the fault-tolerance while still maintaining the security. Another restriction we currently have is the pre-determination of which hosts to visit. Dynamic agent itinerary is a feature we would like to see in the future. The ideal scenario is that the agents can jointly generate an encrypted circuit just in time for any agent to visit its next host. This would also drastically reduce each agent’s size.

The performance results of our practical variant of the secure agent protocol is promising but with significant room for improvement. More efficient threshold decryption and oblivious transfer is desirable. It is also a good idea to study other agent platforms which have better concurrency support. In addition, we would like to see how security against a malicious originator can be added without incurring large overhead. Ultimately, provable security is desirable for the practical protocol, even with a weaker security model such as the random oracle model [14].

For many real-world applications, the whole agent code may be too complex to be implemented

in such a primitive form as boolean circuits. However, as our performance study finds out, the privacy-critical part such as the bid comparison of a shopping agent is indeed suitable for being realized by encrypted circuits. Thus a long-term research topic down the road would be the interface between the encrypted circuit and the rest of the agent code and its impact on security.

BIBLIOGRAPHY

[1] *Foundation for Intelligent Physical Agents. http://www.fipa.org.*

[2] *JADE Programmer's Guide. Available in the JADE download.*

[3] *The Cryptix JCE. http://www.cryptix.org.*

[4] *The Java Agent Development Framework. http://sharon.cselt.it/projects/jade.*

[5] Aglets Web site: http://www.trl.ibm.co.jp/aglets.

[6] J. Algesheimer, C. Cachin, J. Camenisch, and G. Karjoth, *Cryptographic security for mobile code*, in Proc. of the IEEE Symposium on Security and Privacy, May 2001, pp. 2–11.

[7] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, *On the (im)possibility of obfuscating programs*, in Advances in Cryptology — Crypto 2001, J. Kilian, ed., vol. 2139 of Lecture Notes in Computer Science, Springer, 2001, pp. 1–18.

[8] J. Baumann, F. Hohl, K. Rothermel, M. Schwehm, and M. Straßer, *Mole 3.0: A middleware for Java-based mobile software agents*, in Proc. of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), 1998, pp. 355–370.

[9] D. Beaver, *Foundations of secure interactive computing*, in Advances in Cryptology — Crypto'91, J. Feigenbaum, ed., vol. 576 of Lecture Notes in Computer Science, Springer, 1991, pp. 377–391.

[10] ——, *Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority*, Journal of Cryptology, 4 (1991), pp. 75–122.

[11] D. Beaver, S. Micali, and P. Rogaway, *The round complexity of secure protocols*, in Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC), 1990, pp. 503–513.

[12] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, *Relations among notions of security for public-key encryption schemes*, in Advances in Cryptology — Crypto'98, H. Krawczyk, ed., vol. 1462 of Lecture Notes in Computer Science, Springer, 1998, pp. 26–45.

[13] M. Bellare and S. Micali, *Non-interactive oblivious transfer and applications*, in Advances in Cryptology — Crypto'89, G. Brassard, ed., vol. 435 of Lecture Notes in Computer Science, Springer, 1989, pp. 547–557.

[14] M. Bellare and P. Rogaway, *Random oracles are practical: A paradigm for designing efficient protocols*, in Proc. of the 1st ACM Conference on Computer and Communication Security, 1993, pp. 62–73.

[15] M. Ben-Or, S. Goldwasser, and A. Wigderson, *Completeness theorems for non-cryptographic fault-tolerant distributed computation*, in Proc. of the 20th Annual ACM Symposium on Theory of Computing (STOC), 1988, pp. 1–10.

[16] C. Cachin, J. Camenisch, J. Kilian, and J. Müller, *One-round secure computation and secure autonomous mobile agents*, in Proc. of the 27th International Colloquium on Automata, Languages and Programming (ICALP), U. Montanari, J. P. Rolim, and E. Welzl, eds., vol. 1853 of Lecture Notes in Computer Science, Springer, 2000, pp. 512–523.

[17] R. Canetti. Private communication.

[18] ——, *Studies in Secure Multi-Party Computation and Applications*, PhD thesis, Weizmann Institute, Israel, 1995.

[19] ——, *Security and composition of multi-party cryptographic protocols*, Journal of Cryptology, 13 (2000), pp. 143–202.

[20] ——, *Universally composable security: A new paradigm for cryptographic protocols.* ECCC TR 01-16. Also available at Cryptology ePrint Archive http://eprint.icar.org. An extended abstract appeared in the 42nd IEEE Symposium on Foundations of Computer Science (FOCS), pp. 136-145, 2001.

[21] R. Canetti, U. Feige, O. Goldreich, and M. Naor, *Adaptive secure computation*, in Proc. of the 28th Annual ACM Symposium on Theory of Computing (STOC), 1996, pp. 639–648. Fuller version in MIT-LCS-TR#682, 1996.

[22] R. Canetti and S. Goldwasser, *An efficient threshold public key cryptosystem secure against adaptive chosen ciphertext attack*, in Advances in Cryptology — EuroCrypt'99, J. Stern, ed., vol. 1592 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 90–106.

[23] R. Canetti, H. Krawczyk, and J. B. Nielsen, *Relaxing chosen-ciphertext security*, in Advances in Cryptology — Crypto 2003, D. Boneh, ed., vol. 2729 of Lecture Notes in Computer Science, Springer, 2003, pp. 565–582. Full version available online at http://eprint.icar.org, 2003.

[24] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai, *Universally composable two-party and multi-party secure computation.* Available at Cryptology ePrint Archive http://eprint.acar.org. An extended abstract appeared in the 34th Annual ACM Symposium on Theory of Computing (STOC), pp. 494-503, 2002.

[25] D. Chaum, C. Crépeau, and I. Damgård, *Multiparty unconditionally secure protocols*, in Proc. of the 20th Annual ACM Symposium on Theory of Computing (STOC), 1988, pp. 11–19.

[26] D. Chess, B. Grosof, C. Harrison, D. Levine, and C. Paris, *Itinerant agents for mobile computing*, IEEE Personal Communications, 2 (1995), pp. 34–49.

[27] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, The MIT Press, 1990.

[28] R. Cramer and V. Shoup, *A practical public-key cryptosystem provably secure against adaptive chosen ciphertext attack*, in Advances in Cryptology — Crypto'98, H. Krawczyk, ed., vol. 1462 of Lecture Notes in Computer Science, Springer, 1998, pp. 13–25.

[29] Y. Desmedt and Y. Frankel, *Threshold cryptosystems*, in Advances in Cryptology — Crypto'89, G. Brassard, ed., vol. 435 of Lecture Notes in Computer Science, Springer-Verlag, 1989, pp. 307–315.

[30] D. Dolev, C. Dwork, and M. Naor, *Non-malleable cryptography*, SIAM Journal on Computing, 30 (2000), pp. 391–437. Preliminary version in 23rd ACM Symposium on Theory of Computing (STOC), 1991.

[31] J. Domingo-Ferrer and J. Herrera-Joancomartì, *A privacy homomorphism allowing field operations on encrypted data*, in I Jornades de Matemàtica Discreta i Algorísmica, 1999.

[32] R. Endsuleit and T. Mie, *Secure multi-agent computations*, in Proc. of the 2003 International Conference on Security and Management (SAM'03), 2003.

[33] W. Farmer, J. Guttman, and V. Swarup, *Security for mobile agents: Authentication and state appraisal*, in Proc. of the 4th European Symposium on Research in Computer Security (ESORICS'96), 1996, pp. 118–130.

[34] O. Goldreich, *Draft of a chapter on general protocols (second version)*. Extracts from a working draft for Volume 2 of Foundation of Cryptography, January 26, 2003. Available at http://www.wisdom.weizmann.ac.il/ oded/foc-vol2.html.

[35] ——, *Foundations of Cryptography, Volume 1 Basic Tools*, Cambridge University Press, 2001.

[36] O. Goldreich, S. Micali, and A. Wigderson, *How to play any mental game or a completeness theorem for protocols with honest majority*, in Proc. of the 19th Annual ACM Symposium on Theory of Computing (STOC), 1987, pp. 218–229.

[37] S. Goldwasser and Y. Lindell, *Secure computation without agreement*, in Distributed Computing: 16th International Conference (DISC), D. Malkhi, ed., vol. 2508 of Lecture Notes in Computer Science, Springer, 2002, pp. 17–32.

[38] R. S. Gray, *Agent Tcl: A flexible and secure mobile-agent system*, in Proc. of the 4th Annual Tcl/Tk Workshop (TCL'96), 1996, pp. 9–23.

[39] F. Hohl, *Time limited blackbox security: Protecting mobile agents from malicious hosts*, in Mobile Agent and Security, G. Vigna, ed., vol. 1419 of Lecture Notes in Computer Science, Springer, 1998, pp. 92–113.

[40] W. Jansen and T. Karygiannis, *Mobile agent security*, Tech. Rep. Special Publication 800-19, NIST, 1999.

[41] G. Karjoth, N. Asokan, and C. Gülcü, *Protecting the computation results of free-roaming agents*, in 2nd International Workshop on Mobile Agents, 1998.

[42] G. Karjoth, D. B. Lange, and M. Oshima, *A security model for Aglets*, IEEE Internet Computing, (August 1997), pp. 68–77.

[43] N. Karnik, *Security in Mobile Agent Systems*, PhD thesis, Department of Computer Science, University of Minnesota, October 1998.

[44] D. Kotz, R. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko, *AGENT TCL: Targeting the needs of mobile computers*, IEEE Internet Computing, 1(4) (July/August 1997), pp. 58–67.

[45] S. Loureiro and R. Molva, *Function hiding based on error correcting codes*, in Proc. of Cryptec'99 – International Workshop on Cryptographic Techniques and Electronic Commerce, 1999, pp. 92–98.

[46] S. Micali and P. Rogaway, *Secure computation*, in Advances in Cryptology — Crypto'91, J. Feigenbaum, ed., vol. 576 of Lecture Notes in Computer Science, Springer, 1991, pp. 392–404.

[47] M. Naor, B. Pinkas, and R. Sumner, *Privacy preserving auctions and mechanism design*, in Proc. of the 1st ACM Conference on Electronic Commerce, 1999, pp. 129–139.

[48] M. Naor and M. Yung, *Public key cryptosystems provably secure against chosen ciphertext attacks*, in Proc. of the 22nd Annual ACM Symposium on Theory of Computing (STOC), 1990, pp. 427–437.

[49] G. Necula and P. Lee, *Safe kernel extensions without run-time checking*, in Proc. of the 2nd Symposium on Operating System Design and Implementation (OSDI'96), 1996, pp. 229–243.

[50] J. Ousterhout, *Scripting: Higher-level programming for the 21st century*, IEEE Computer, (March 1998), pp. 23–30.

[51] C. Rackoff and D. Simon, *Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack*, in Advances in Cryptology — Crypto'91, J. Feigenbaum, ed., vol. 576 of Lecture Notes in Computer Science, Springer, 1991, pp. 433–444.

[52] R. Richardson and J. Kilian, *On the concurrent composition of zero-knowledge proofs*, in Advances in Cryptology — EuroCrypt'99, J. Stern, ed., vol. 1592 of Lecture Notes in Computer Science, Springer, 1999, pp. 415–431.

[53] R. Rivest, L. Adleman, and M. Dertouzos, *On data banks and privacy homomorphisms*, in Foundations of Secure Computation, R. DeMillo, D. Dobkin, A. Jones, and R. Lipton, eds., Academic Press, 1978, pp. 169–177.

[54] P. Rogaway, *The Round Complexity of Secure Protocols*, PhD thesis, Laboratory for Computer Science, MIT, April 1991.

[55] V. Roth, *Secure recording of itineraries through cooperating agents*, in Proc. of the ECOOP Workshop on Distributed Object Security and 4th Workshop on Mobile Object Systems: Secure Internet Mobile Computations, 1998, pp. 147–154.

[56] T. Sander and C. F. Tschudin, *Protecting mobile agents against malicious hosts*, in Mobile Agents and Security, G. Vigna, ed., vol. 1419 of Lecture Notes in Computer Science, Springer, 1998, pp. 379–386.

[57] T. Sander, A. Young, and M. Yung, *Non-interactive cryptocomputing for NC^1*, in Proc. of the 40th IEEE Symposium on Foundations of Computer Science (FOCS), 1999, pp. 554–566.

[58] A. D. Santis, G. D. Crescenzo, R. Ostrovsky, G. Persiano, and A. Sahai, *Robust non-interactive zero knowledge*, in Advances in Cryptology — Crypto 2001, J. Kilian, ed., vol. 2139 of Lecture Notes in Computer Science, Springer, 2001, pp. 566–598.

[59] F. B. Schneider, *Towards fault-tolerant and secure agentry*, in Proc. of the 11th International Workshop on Distributed Algorithms, 1997.

[60] V. Shoup and R. Gennaro, *Securing threshold cryptosystems against chosen ciphertext attack*, Journal of Cryptology, 15 (2002), pp. 75–96.

[61] J. Tardo and L. Valente, *Mobile agent security and telescript*, in Proc. of IEEE COMPCON'96, 1996, pp. 58–63.

[62] S. R. Tate and K. Xu, *On garbled circuits and constant round secure function evaluation*, tech. rep., Department of Computer Science, University of North Texas, 2003.

[63] J. Viega and M. Messier, *Secure Programming Cookbook for C and C++*, O'Reilly, 2003.

[64] G. Vigna, *Protecting mobile agents through tracing*, in Proc. of the 3rd ECOOP Workshop on Mobile Object Systems, 1997.

[65] R. Wahbe, S. Lucco, and T. Anderson, *Efficient software-based fault isolation*, in Proc. of the 14th ACM Symposium on Operating Systems Principles, ACM SIGOPS Operating Systems Review, December 1993, pp. 203–216.

[66] J. E. White, *Telescript technology: Mobile agents*, in Software Agents, J. Bradshaw, ed., AAAI/MIT Press, 1996.

[67] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, and B. Peet, *Concordia: An infrastructure for collaborating mobile agents*, in Proc. of the First International Workshop on Mobile Agents, vol. 1219 of Lecture Notes in Computer Science, Springer, 1997, pp. 86–97.

[68] A. C. Yao, *How to generate and exchange secrets*, in Proc. of the 27th IEEE Symposium on Foundations of Computer Science (FOCS), 1986, pp. 162–167.

[69] B. S. Yee, *A sanctuary for mobile agents*, in Secure Internet Programming, J. Vitek and C. Jensen, eds., vol. 1603 of Lecture Notes in Computer Science, Springer, 1999, pp. 261–273.