New Applications of Public Ledgers

by

Gabriel Kaptchuk

A dissertation submitted to The Johns Hopkins University
in conformity with the requirements for the degree of
Doctor of Philosophy

Baltimore, Maryland
March 27, 2020

Abstract

The last decade and a half has seen the rise of a new class of systems loosely categorized as public ledgers. Public ledgers guarantee that all posted information is permanently available to the entire public. Common realizations of public ledgers include public blockchains and centralized logs. In this work we investigate novel applications of public ledgers. We begin by describing *enclave ledger interaction*, a computational method that allows the execution of trusted execution environments or cryptographically obfuscated programs to be conditioned on the contents of the ledger. We then show how this *conditional execution* paradigm can be used to achieve fairness in dishonest majority secure multiparty computation, which is impossible in the plain model. Finally, we show how conditional execution can be used to build systems that facilitate law enforcement access to ciphertext while ensuring robust transparency and accountability mechanisms.

Thesis Committee

Primary Readers

Avi Rubin (Primary Advisor)
Professor
Department of Computer Science
Johns Hopkins Whiting School of Engineering

Matthew Green (Advisor)
Associate Professor
Department of Computer Science
Johns Hopkins Whiting School of Engineering

Abhishek Jain
Assistant Professor
Department of Computer Science
Johns Hopkins Whiting School of Engineering

Acknowledgments

I would like to thank the many people who made completing this journey possible. First, many thanks to my advisors, Avi Rubin and Matthew Green, for the uncountable hours of support and teaching. I would also like to thanks Abhishek Jain for all of his mentorship. Additionally, I would like to thank the NSF for funding me on grant #1329686 throughout my PhD.

I owe a big debt of gratitude to the other security and cryptography PhD students I have had the pleasure of working with at JHU, including Ian Miers, Christina Garman, Paul Martin, Michael Rushanan, Alishah Chator, Dave Russell, Arka Rai Choudhuri, Aarushi Goel, Zhengzhong Jin, Tushar Jois, Max Zinkus, Gabby Beck, and Gijs Van Laer. Thank you all for helping me learn and providing such a supportive environment.

I would like to thank the organizations and people that hosted me for internships during my PhD. Thank you Michael Steiner and Mic Bowman at Intel Labs for mentoring me over Summer 2017. Thank you to Senator Ron Wyden for making it possible for me to work in his personal office in the Senate over Summer 2018. Also, thank you so much to his staff, Chris Soghoian in particular, and the other fellows, Shaanan Cohney in particular, for making working in the Senate such a rewarding experience. Thanks to

Tech Congress for funding me while I worked in the Senate.

Thank you to my non-academic communities that kept me sane while working on my PhD, including Rat City Ultimate, WETOM Ultimate, FCDC Ultimate, the Bnai Israel Community, and my roommates Nick Igo, Joseph Yu, and Ben Strober. Thanks to my parents, Maribeth and Ted Kaptchuk for their tireless and unending support. Finally, thank you so so much to my partner Rose!

Table of Contents

Chapter 1

Introduction

The last decade and a half has seen the rise of a new class of systems loosely categorized as *public ledgers*. These systems act as the bulletin boards of the internet age, where posted information can be reliably retrieved by the public. In some cases, these public ledgers are run by single, trusted operators, and in other cases, the systems are comprised of many mutually distrusting parties. No matter the underlying system structures, public ledgers provide the public access to some current *state*. Over time, the ledger protocol participants are able to update this state while ensuring that all participants have consistent view of the most recent state. Many realizations of public ledgers make it difficult for individual parties to tamper with the ledger's state.

Public ledgers provide the following guarantees:

- Information posted on the ledger is accessible to the entire public
- Information posted on the ledger is permanently accessible

These two guarantees ensure that each member of the public has access to the same information. Protocol designers use public ledgers to realize

other, higher-level applications. For example, cryptocurrency systems like Bitcoin (Nakamoto, 2008) leverage *decentralized* public ledgers, realized by blockchains, to ensure that everyone agrees on the current balances of all currency holders. Ethereum (*The Ethereum Project* 2018) extended existing cryptocurreny techniques to construct a massive distributed virtual machine capable of arbitrary, verifiable public computation. The Certificate Transparency log leverages *centralized* public ledgers, realized by a server with a private signing key, to ensure that browsers can verify that TLS certificates are issued properly.

While there is a tremendous amount of research focusing on developing new techniques to realize public ledgers, particularly the *decentralized* variety (see (Bano et al., 2017; Raikwar, Gligoroski, and Kralevska, 2019), and the citations contained therein), relatively little academic focus has been given to *applications* of these systems beyond cryptocurrencies. Public ledgers are more than just an academic concept — public ledgers exist and operate in practice. Public ledgers are powerful cryptographic building-blocks, so exploring new applications has the potential for practical impact. In this work, we present new application powered by public ledgers. These applications make minimal assumptions about the underlying ledger mechanism, so any realization of a public ledger can be used as a building-block.

1.1 Blockchains and Decentralized Consensus

The most notable systems to provide the public ledger guarantees are blockchains. Blockchains were first introduced along with the cryptocurrency Bitcoin in a

2008 white paper titled *Bitcoin: A Peer-to-Peer Electronic Cash System*, written under the pseudonym Satoshi Nakamoto (Nakamoto, 2008). Since then, the techniques described in the work have become commonplace in the computer science world and beyond. In 2017 and 2018, a dramatic rise in the value of the Bitcoin currency accompanied widespread public attention and excitement from non-technical onlookers. Although the craze quickly settled, blockchains now underpin thousands of distributed, digital currencies, nearly a hundred of which have market capitalizations above $50 million. While this swift technological adoption is widely viewed as a revolution in decentralized finance, blockchains are still struggling to make an impact in other industries.[1] In this work, we focus on exploring new applications of blockchains, when considered as a cryptographic building block.

Blockchains are a method of achieving *consensus* among a large number of *distributed*, mutually distrusting parties. This consensus can be on any arbitrary piece of information. In the case of cryptocurrencies, a given blockchain comes to consensus about the current balances of each party in the network. This consensus process must be robust; if a single party can convince the entire network that their balance should be arbitrarily increased, users of a cryptocurrency could forge currency. As such, each party participating in the protocol must be viewed with skepticism and the overall network must enforce rules that allow the high-level application to function correctly.

It is important to note that blockchains are not the only way to achieve

[1] There is significant attention on using blockchains to manage and track supply chains. While this is not strictly a financial application, from a technical perspective it is virtually identical; instead of tracking the flow of currency, the blockchain must just track the flow of tokens representing goods.

distributed consensus. This exact problem has a lengthy history in computer science, generally referred to as *The Byzantine Generals Problem* (Pease, Shostak, and Lamport, 1980; Lamport, Shostak, and Pease, 2019). The study first considered *faulty* machines that might inject errors into a replicated system. These replicated systems might be sensors guiding an aircraft or giving monitoring information to a controller at a power station. Allowing faulty systems to exert undo influence in these applications would have devastating effects. While widely studied by computer scientists, solutions to the Byzantine Generals Problem rarely scaled efficiently when the number of parties grew very large.

Because blockchains were originally suggested as a consensus mechanism to support a global currency, scaling to thousands or millions of protocol participants in the consensus mechanism is crucial. To overcome the limitations of prior work, blockchains run a non-interactive lottery procedure called "mining," in which a random party (or possibly set of parties) is selected to update the state of network. Any party that accepts this update (indicating that the update is in accordance with the network rules) begins mining state updates to append to the accepted update. Each of these updates is encoded into a "block" of information, and the iterated updates are linked together to create a chain of blocks (thus the name). Under the assumption that a certain percentage of the network resources are honest, the network is robust.

The exact mechanism for this mining differs depending on the blockchain. Bitcoin (Nakamoto, 2008) and Ethereum (*The Ethereum Project* 2018) are two *proof-of-work* blockchains that make the assumption that the operators of a majority of the network's computational power are honest. Mining in these

networks takes the form of solving randomized cryptographic puzzles, with a solver deemed the lottery winner. If honest operators solve a majority of these puzzles, the blockchain is, over time, robust to malicious activity. Algorand (Gilad et al., 2017; Chen et al., 2018) and Orberous (Kiayias et al., 2017) are two *proof-of-stake* blockchains that make the assumption that honest parties hold a majority of the currency in the network. The lottery for these networks has currency in the network as lottery tickets, with a random subset of these tickets being randomly selected as winners. More complex consensus mechanisms have also been studied, including *proof-of-space* (Dziembowski et al., 2015; Park et al., 2015) and *proof-of-space-time* (Moran and Orlov, 2019).

1.1.1 Blockchains as Public Ledgers

By their very nature, blockchains are a realization of public ledgers. Because blockchains consider everyone a potential protocol participant, all posted information is made public. It is straight forward to see why all information posted on the blockchain becomes public. To achieve consensus, all protocol participants in the network *must* know (and agree upon) all information that is part of the consensus state.

As mentioned before, blockchains consist of a series of state updates encoded into blocks. Each update in appended to the previous block. As such, the current state of the blockchain is defined by looking at the *entire* history of the protocol. To provide the required consistency, this means that all the information on the blockchain is *immutable*, *i.e.* it cannot be changed. Indeed, popular proof-of-stake blockchains like Bitcoin leverage the immutability of

old information explicitly, requiring that information must be sufficiently "old" before it is considered part of the consensus state.

1.2 Other Realizations of Ledgers

While blockchains are the most widespread realization of public ledgers, there are other realizations found in practice. These alternatives use different assumptions to achieve security, but still provide the ledger guarantees. We briefly describe two prominent realizations of the ledger functionality: centralized public ledgers and private ledgers.

Centralized Public Ledgers. Centralized ledgers provide the same guarantees as public ledgers realized from blockchains but make stronger assumptions about party or parties running the ledger. Specifically, a centralized public ledger only function correctly if the party or parties running the ledger are honest. While this is clearly a dramatically stronger assumption, it appears to hold true in practice for specific applications. For instance, Google has recently started to run a Certificate Transparency log (*Certificate Transparency* 2018), which is a functioning centralized public ledger. This log tracks the list of TLS certificates that are issued by certificate authorities. Providing this information publicly makes abusing a compromised certificate authority key more difficult. The security of the ledger requires that Google's secret signing key remains private. Additionally, the guarantee that information posted to the ledger becomes public relies on Google's infrastructure publishing updates regularly and in a publicly available way.

When leveraging centralized public ledgers in our applications, we generally make the simplifying assumption that the purpose of the ledger is unrelated to our application. For example, it is reasonable to assume that the operator of the Certificate Transparency log cares primarily about certificates. If our application is leveraging ledgers to give stateless execution environments protections from rollback attacks (see Chapter 3), information posted to the log for this application does not meaningfully concern certificates. Instead, we are piggybacking on an service operating independently. A malicious ledger operator is unlikely to interfere with the execution of our protocol.

Private Ledgers. An important case of public ledgers is the private ledger. Private ledgers operate in the same way as a public ledger, but require authentication before a party can participate or access data. These protocols may have fewer mutually distrusting protocol participants, but still use similar consensus mechanisms to ensure the robustness of the network. For example, JP Morgan recently developed a private version of Ethereum called Quorum (*Quorum*). The Quorum blockchain updates the Ethereum protocol by making it possible to compute on data that remains secret to observers. Additionally, Quorum deployments restrict the parties that can update the state of the system and can participate in the underlying consensus mechanism. This is a key difference from public blockchains. To leverage a private ledger in our applications, the application users must have access to this ledger. In some cases, this limitation may render a particular ledger inappropriate. Throughout this work, we focus on public ledgers and note that if all parties using the application have access to a private ledger, this private ledger may be

substituted.

1.3 Applications of Ledgers

In this work, we present new applications of public ledgers. These applications all take advantage of the core public ledger guarantees, *i.e.*

- Public Access
- Immutability

Put another way, once information is published on the ledger, it is permanently available to everyone. We investigate conditioning various other events on the publication of information onto the ledger. This is conceptually straight forward if actual protocol participants can run the conditional events — the participant simply monitors the ledger and executes subprotocols as appropriate. However, we are interested in building *cryptography* that is innately responsive to the publication of information on a ledger. A cryptographic primitive, *i.e.* encryption, is not a protocol participant, and thus cannot directly reach out and monitor events on the network. To overcome this limitation, we require one additional property of our public ledgers:

- Offline Verifiability

To realize offline verifiability, we assume that public ledgers create an *authenticator* or *proof of publication* that an efficient algorithm can use to decide if some information has been posted to the ledger. We use the terms *authenticator*

or *proof of publication* interchangeably throughout this work. This algorithm must be able to run locally — without network access. Importantly, this algorithm must have an unforgeability property in order to be meaningful. If a player can locally compute a proof of publication that convinces the algorithm, then the algorithm does not serve its purpose. A simple solution with cryptographic security would be to use a cryptographic signature scheme; any information posted to the ledger is signed under some well-known key. This signature can then act as the authenticator, and can be verified locally. Centralized public ledgers, like Certificate Transparency, employ this exact solution. However, the use of a single secret key makes public ledgers of this kind prone to abuse. The ledger administrator must ensure the key is protected, and if the administrator is malicious, they may even give out signatures improperly. Proof-of-work blockchains generally use the solution of a sequence of randomized puzzles as an authenticator. This provides only *economic security*, rather than cryptographic security — meaning that forging a proof of publication is economically infeasible for any party to do alone. Verifying that a piece of information is in a block followed by many more blocks, each with a valid puzzle solution, is sufficient to verify that the information is public. In Bitcoin, a sequence of six blocks with valid puzzle solutions is considered securely part of the public state.

Given this authenticator, it is now possible to build cryptography that is responsive to the ledger. Upon the publication of some information, a protocol participant can gather that information along with the appropriate authenticator. This information can be processed by the other cryptographic components

using the verification algorithm. Once convinced that the information must indeed be public (*i.e.* posted), the conditional event can happen (for instance, decryption of a ciphertext).

With the properties of our public ledger in hand, we now proceed to describe the three applications contained in this work.

1.3.1 Enclave Ledger Interaction

We begin our study of public ledgers by exploring the interaction between trusted execution environments, such as trusted hardware and cryptographic obfuscation, and public ledgers. We name this paradigm *enclave ledger interaction* (ELI), as enclave is a common name for these trusted computing systems. These two computation paradigms have highly complementary properties. Public ledgers are highly distributed and are immutable, while enclaves are highly localized and innately vulnerable to *reset attacks*. In these reset attacks, an attacker tricks the enclave into executing the same logic multiple times on different inputs, which can be devastating depending on the application. For instance, a common use of enclaves in mobile devices is to check a user's password and make sure that the user cannot guess too many times; this allows users to use short passwords without compromising security. However, if the enclave can be reset, this guess limit can be circumvented. These reset attacks are a consequence of not being able to keep state. We show that combining public ledgers and enclaves can prevent this class of attack.

Enclaves and ledgers can also complement each other in another way. Enclaves can keep information secret, whereas all information stored on a

ledger is publicly visible. When combined, we are able to realize systems that select the best qualities of each component: private computation that is contingent on public, immutable events. This combination can be used to realize blockchains that support arbitrary transactions that are evaluated on information that is secret. As noted earlier, Quorum (*Quorum*) leverages this paradigm to allow secret, robust computation in a private network. [2] This computation can allow computation over multiple players' private inputs, simulating the same primitive as secure multiparty computation, albeit from different (possibly more powerful) building blocks.

ELI is a protocol between an enclave (either a piece of trusted hardware or a cryptographically obfuscated program), a host (the machine that runs the enclave), and a public ledger. The enclave is programmed to run some step-by-step functionality, in which each step takes in some tuple (`old_state`, `input`) and outputs a tuple (`new_state`, `output`). Because the enclave contains no non-volatile storage, state must be offloaded to the host after each step of computation, although the state itself may be encrypted, and therefore opaque to the host. ELI binds enclave execution tightly to a chain of state updates on the ledger. These chains are linear sequences of updates that are bound using cryptographic hash functions. Chains are common in blockchain based public ledgers, and can be simulated on all ledgers (see Section 2.1 for a description of ledgers with chains). The enclave only executes the next step of the computation if it can verify that the host has appropriately posted a commitment to their inputs on the ledger. This commitment is also used as

[2] While the properties of Quorum are very similar to our work, it became public after initial publication of the ideas in this work.

a pseudorandom generator seed, *derandomizing* the computation; repeated execution on the same input always yields the same result. The linearity of the computation (not allowing multiple execution paths to be run from the same initial starting state) depends on the linearity of the chain present on the ledger.

ELI realizes a specific form of *conditional, secret execution*. That is, the enclave is able to execute secret computation depending on (possibly encrypted) public state updates. This paradigm can be used to realize a massive number of applications. In Chapter 3, we discuss how this can be used to realize private smart contracts, guaranteed logging before accessing a sensitive file, security with short passwords, and autonomous ransomware. In fact, the other applications we discuss in Chapters 4 and 5 can actually be seen as specific notions of conditional, secret execution as well.

1.3.2 Overcoming the Impossibility of Fairness

A desirable property of interactive protocols is *fairness*, the guarantee that either all players learn the output of the protocol or none do. However, there is a well-known result that achieving fairness is impossible in many circumstances, specifically when a majority of the protocol participants can act dishonestly (Cleve, 1986). This impossibility is particularly relevant to secure multiparty computation protocols (MPC) (Yao, 1986; Goldreich, Micali, and Wigderson, 1987; Chaum, Crépeau, and Damgård, 1988; Ben-Or, Goldwasser, and Wigderson, 1988). MPC protocols allow mutually distrusting parties to compute arbitrary functions without revealing their private inputs. Without

fairness, these protocols lose some of their utility. Consider a group of parties computing an auction over secret bids. If an adversarial party learns the output first, they can abort the computation, claiming a protocol failure. This party can then adaptively change their input when the protocol is re-executed, allowing them to win the auction. Put simply, this abort condition is what makes fair MPC impossible; if we let the adversary always be the party to send the last message of the protocol, then the adversary can simply abort instead, and still learn the computation's output (notice that the adversary does not need to receive any more messages in the protocol, so it already knows the output).

We show that it is possible to overcome this impossibility when players have access to a public ledger. Intuitively, the protocol participants can leverage the conditional execution property discussed before, where the program that is executed conditionally is just decryption. At a high level, this is accomplished by modifying the MPC protocol to compute the encryption of the output, such that decryption is conditioned on each player posting some information publicly. Importantly, because the information itself must be posted publicly, this part of the protocol is not vulnerable to early abort attacks. The adversary can either post its information, thereby letting all players access this information and decrypt the output, or abort, making it impossible to decrypt the output for any player — including itself.

Realizing the details of this protocol is slightly more complex. In Chapter 4, we give two constructions of this protocol. The first is of primarily theoretical interest. The MPC protocol is modified to compute the witness encryption

(see Section 2.2.6) of the function output, where the language is formalized with respect to the public ledger interface. Recall that the execution that is conditioned on the public ledger in the intuitive solution above is ciphertext decryption. As such witness encryption is sufficient. The second construction, intended to be practically interesting, relies on trusted hardware to provide fairness (but importantly, not privacy). Instead of computing a witness encryption, this protocol uses an authenticated symmetric key encryption scheme, where the key is hidden inside of trusted hardware. Only upon seeing that all players are ready to receive the decrypted output (by examining the public ledger for tokens signaling readiness), the trusted hardware outputs the proper decryption key to all players.

1.3.3 Abuse Resistance Law Enforcement Access System

In the previous applications, the "public" in public ledgers is relatively limited. For instance, it is important that the MPC protocol participants can access the ledger to achieve fairness, but the general public need not read any of the information posted there. We now turn our attention to an application that leverages the public aspects of public ledgers more expansively.

Proliferation of End-to-End Encryption. As concern for user privacy has grown, modern communication systems almost universally use some degree of encryption to protect the contents of messages. A growing percentage of these systems have also started to deploy *end-to-end encryption* (E2E), in which only the sender and the receiver of a message are able to recover the plaintext. This means that the service provider cannot read the message, and learns

nothing beyond the metadata associated with the message. E2E systems include some of the most widely used messaging applications, including Apple iMessage (*iMessage*) and WhatsApp (WhatsApp, 2017), as well as the privacy-focused messaging applications like Signal (*Signal Secure Messaging System*) and Telegram (*Telegram*). Platforms like Apple FaceTime (*FaceTime*) also use E2E to keep real-time video and audio data private. The spread of E2E is widely considered a victory for privacy advocates as it ensures that companies cannot spy on their users, and sensitive information cannot be compromised in the event of a breach.

A consequence of the proliferation of E2E systems is that service providers are unable to provide message plaintext in response to a court order. Members of law enforcement often refer to this as the "going dark" problem, as law enforcement is no longer able to access the material to which they had become accustom (Federal Bureau of Investigation, 2016). In 2016, the closely related issue of device encryption became the subject of public discourse in the United States after the FBI was unable to access the iPhone of the San Bernardino shooter (*The Apple-F.B.I. Case* 2016). Notable members of law enforcement, including FBI Director Christopher Wray, testified before Congress on the detrimental effect of widespread encryption adoption, although it has since become clear that elements of their testimony were overstated (Wray, 2017). Starting in 2018, the Justice Department, under Attorney General William Barr, resurrected government attacks on encryption systems, explicitly addressing the spread of E2E messaging. In an open letter, Barr, along with officials from the UK, criticized the technology industry for enabling the spread of child

sexual abuse material by rolling out E2E (Patel et al., 2019). In December 2019, another round of congressional hearings featured senators on both sides of the aisle chastising technology companies for not finding a "workable" solution and warning about the possibility of future legislation curtailing the legal uses of cryptography (Senate Committee on the Judiciary, 2019).

Achieving Accountability using Ledgers. All existing technical proposals for encryption systems that provide law enforcement with access to plaintext fail to provide meaningful abuse resistance properties (*e.g.* (Denning, 1994; Savage, 2018; Bellovin et al., 2018; Tait, 2016; Levy and Robinson, 2018; Bellare and Rivest, 1999)). Generally, these proposals rely on *key escrow*, in which the government holds long-term keys that allow for legal access. These solutions are incredibly vulnerable to abuse; if government key material is compromised, by a foreign government for example, it would be impossible to detect its misuse. Alternatively, if members of law enforcement decided to surreptitiously decrypt messages, it would be impossible for civil liberty groups to notice. In fact, there is little transparency in these proposed systems at all. Noting these problems, recent policy working groups (Encryption Working Group, 2019; National Academies of Sciences, Engineering, and Medicine, 2018) indicate that key escrow solutions are insufficient, with the possible exception of device encryption. Any truly workable solution *must* provide both meaningful transparency and accountability.

In this work, we leverage public ledgers to design encryption systems that allow for law enforcement access only after law enforcement has published the proper accountability and transparency information. This information

can provide civil liberty groups and the general public with confidence that the law is being properly followed, and zero knowledge proofs can be used to hide any information that is considered sensitive, *e.g.* the target of an active investigation. We provide formal definitions for an abuse resistant law enforcement access system, and explore the requirements for realizing it. Specifically, we show that building an abuse resistant system is relatively straightforward, providing law enforcement posts the accountability information before a message is encrypted. Intuitively, law enforcement can generate and publicly propagate key information such that messages that are the target of surveillance can be decrypted, and all others will be information theoretically destroyed. Additionally, law enforcement can use non-interactive zero knowledge proofs to show that this new key material has been generated in accordance with valid warrants.

However, achieving abuse resistant law enforcement access for ciphertext that was encrypted *before* accountability information is posted publicly requires extractable witness encryption (see Section 2.2.6.1) for specific languages. Our construction is similar to the conditional decryption mechanism leveraged in ELI. Unlike the case where key material is generated before encryption, in this case decryption should only be possible if a specific event happens after, *i.e.* law enforcement posts the proper accountability information. This is a form of conditioning decryption on public events. Without extractable witness encryption (or a non-cryptographic assumption like trusted hardware), it is impossible to realize this functionality.

1.4 Organization of this work.

Chapter 2 gives notation and definitions required throughout other chapters. This includes the interface and requirements for the ledger used. Chapter 3 contains a description of the enclave ledger iteration paradigm and discusses some direct applications. Chapter 4 shows how public ledgers can be used to overcome prior impossibility results for secure multiparty computation. Lastly, Chapter 5 investigates the way public ledgers can be used to achieve abuse resistant law enforcement access systems.

Introduction References

Nakamoto, S. (2008). "Bitcoin: A peer-to-peer electronic cash system, 2008". In: URL: http://www.bitcoin.org/bitcoin.pdf.

The Ethereum Project (2018). https://www.ethereum.org/.

Bano, Shehar, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis (2017). *Consensus in the Age of Blockchains*. arXiv: 1711.03936 [cs.CR].

Raikwar, Mayank, Danilo Gligoroski, and Katina Kralevska (2019). *SoK of Used Cryptography in Blockchain*. Cryptology ePrint Archive, Report 2019/735.

Pease, Marshall, Robert Shostak, and Leslie Lamport (1980). "Reaching agreement in the presence of faults". In: *Journal of the ACM (JACM)* 27.2, pp. 228–234.

Lamport, Leslie, Robert Shostak, and Marshall Pease (2019). "The Byzantine generals problem". In: *Concurrency: the Works of Leslie Lamport*, pp. 203–226.

Gilad, Yossi, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich (2017). *Algorand: Scaling Byzantine Agreements for Cryptocurrencies*. Cryptology ePrint Archive, Report 2017/454.

Chen, Jing, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos (2018). *ALGORAND AGREEMENT: Super Fast and Partition Resilient Byzantine Agreement*. Cryptology ePrint Archive, Report 2018/377.

Kiayias, Aggelos, Alexander Russell, Bernardo David, and Roman Oliynykov (2017). "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *CRYPTO 2017, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. LNCS. Springer, Heidelberg, pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12.

Dziembowski, Stefan, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak (2015). "Proofs of Space". In: *CRYPTO 2015, Part II*. Ed. by Rosario Gennaro and Matthew J. B. Robshaw. Vol. 9216. LNCS. Springer, Heidelberg, pp. 585–605. DOI: 10.1007/978-3-662-48000-7_29.

Park, Sunoo, Krzysztof Pietrzak, Albert Kwon, Joël Alwen, Georg Fuchsbauer, and Peter Gaži (2015). *Spacemint: A Cryptocurrency Based on Proofs of Space*. Cryptology ePrint Archive, Report 2015/528.

Moran, Tal and Ilan Orlov (2019). "Simple Proofs of Space-Time and Rational Proofs of Storage". In: *CRYPTO 2019, Part I*. Ed. by Alexandra Boldyreva and Daniele Micciancio. Vol. 11692. LNCS. Springer, Heidelberg, pp. 381–409. DOI: 10.1007/978-3-030-26948-7_14.

Certificate Transparency (2018). Available at https://www.certificate-transparency.org.

Quorum. *Quorum*. URL: https://www.goquorum.com/.

Cleve, Richard (1986). "Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract)". In: *STOC*, pp. 364–369.

Yao, Andrew Chi-Chih (1986). "How to Generate and Exchange Secrets (Extended Abstract)". In: *27th FOCS*. IEEE Computer Society Press, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

Goldreich, Oded, Silvio Micali, and Avi Wigderson (1987). "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, pp. 218–229. DOI: 10.1145/28395.28420.

Chaum, David, Claude Crépeau, and Ivan Damgård (1988). "Multiparty Unconditionally Secure Protocols (Abstract) (Informal Contribution)". In: *CRYPTO'87*. Ed. by Carl Pomerance. Vol. 293. LNCS. Springer, Heidelberg, p. 462. DOI: 10.1007/3-540-48184-2_43.

Ben-Or, Michael, Shafi Goldwasser, and Avi Wigderson (1988). "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *20th ACM STOC*. ACM Press, pp. 1–10. DOI: 10.1145/62212.62213.

Apple. *iMessage*. Available at https://support.apple.com/explore/messages. URL: https://support.apple.com/explore/messages.

WhatsApp (2017). *WhatsApp Encryption Overview*. Available at https://scontent.whatsapp.net/v/t61/68135620_760356657751682_6212997528851833559_n.pdf/WhatsApp-Security-Whitepaper.pdf?_nc_ohc=KLWdfyGXHtkAX_XDL-n&_nc_ht=scontent.whatsapp.net&oh=8436499c2708501873e5bae88ce96c6e&oe=5E450BE5.

Signal. *Signal Secure Messaging System*. URL: https://signal.org/.

Telegram. URL: https://telegram.org/.

Apple. *FaceTime*. Available at https://apps.apple.com/us/app/facetime/id1110145091. URL: https://apps.apple.com/us/app/facetime/id1110145091.

Federal Bureau of Investigation (2016). *Going Dark*. Available at https://www.fbi.gov/services/operational-technology/going-dark.

The Apple-F.B.I. Case (2016). The New York Times. Available at https://www.nytimes.com/news-event/apple-fbi-case.

Wray, Director Christopher (2017). *'Keeping America Secure in the New Age of Terror:' Statement Before the House Homeland Security Committee:* FBI website. Available at https://www.fbi.gov/news/testimony/keeping-america-secure-in-the-new-age-of-terror.

Patel, Priti, William Barr, Kevin McAleenan, and Peter Dutton (2019). *Open Letter: Facebook's 'Privacy First' Proposals*. Justice Department website. Available at https://www.justice.gov/opa/press-release/file/1207081/download.

Senate Committee on the Judiciary (2019). *'Encryption and Lawful Access: Evaluating Benefits and Risks to Public Safety and Privacy:' Hearing before the Senate Committee on the Judiciary*. Senate Committee on the Judiciary website. Available at https://www.judiciary.senate.gov/meetings/encryption-and-lawful-access-evaluating-benefits-and-risks-to-public-safety-and-privacy.

Denning, Dorothy E. (1994). "The US key escrow encryption technology". In: *Computer Communications* 17.7, pp. 453–457. DOI: 10.1016/0140-3664(94)90099-X. URL: https://doi.org/10.1016/0140-3664(94)90099-X.

Savage, Stefan (2018). "Lawful Device Access without Mass Surveillance Risk: A Technical Design Discussion". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, pp. 1761–1774. ISBN: 9781450356930. DOI: 10.1145/3243734.3243758. URL: https://doi.org/10.1145/3243734.3243758.

Bellovin, Steven M., Matt Blaze, Dan Boneh, Susan Landau, and Ronald R. Rivest (2018). *Analysis of the CLEAR Protocol per the National Academies' Framework*. Tech. rep. CUCS-003-18. Columbia University. URL: \url{https://mice.cs.columbia.edu/getTechreport.php?techreportID=1637}.

Tait, Matt (2016). "An Approach to James Comey's Technical Challenge". In: *Lawfare*. URL: \url{https://www.lawfareblog.com/approach-james-comeys-technical-challenge}.

Levy, Ian and Crispin Robinson (2018). "Principles for a More Informed Exceptional Access Debate". In: *Lawfare*. URL: \url{https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate}.

Bellare, Mihir and Ronald L. Rivest (1999). "Translucent Cryptography - An Alternative to Key Escrow, and Its Implementation via Fractional Oblivious Transfer". In: *Journal of Cryptology* 12.2, pp. 117–139. DOI: 10.1007/PL00003819.

Encryption Working Group (2019). *Moving the Encryption Policy Conversation Forward*. Tech. rep. Carnegie Endowment for International Peace. URL: \url{https://carnegieendowment.org/files/EWG__Encryption_Policy.pdf}.

National Academies of Sciences, Engineering, and Medicine (2018). *Decrypting the Encryption Debate: A Framework for Decision Makers*. Washington, DC: The National Academies Press. URL: \url{https://www.nap.edu/catalog/25010/decrypting-the-encryption-debate-a-framework-for-decision-makers}.

Chapter 2

Definitions and Modeling

Notation. Let λ be an adjustable security parameter and $\mathsf{negl}(1^\lambda)$ be a negligible function in λ. Throughout our constructions we will use $\|$ to denote bitwise concatenation, $\overset{c}{\approx}$ to denote computational indistinguishably, and $\overset{s}{\approx}$ to denote statistical indistinguishably. Let $\leftarrow$ or $\rightarrow$ denote assignment, as appropriate. To denote that an algorithm or protocol is parameterized by a value or other algorithm, we will use superscript.

2.1 Modeling the Ledger

We begin our definitions chapter by formalizing the interface for the public ledger. The ledger can be thought of as an bulletin board that allows parties to publish arbitrary information, encoded as bit-strings. Recall from the introduction that we will require a primitive that provides the following guarantees:

- Public access to all published information
- Immutability of all published information

- Offline verifiability of publication

In general, we will accomplish this third property using a publicly-verifiable *authentication tag* or *proof of publication*, which will establish that the string was indeed published.

In Chapter 3, we require an additionally property of our public ledgers. Specifically, we require that that information is posted as part of a specific *chain* of posted values. This property occurs naturally in blockchains and is easy to realize in other concrete instantiations of public ledgers. In Chapters 4 and 5, we do not require this property. As such, we begin by giving a definition of a public ledger that has this property, followed by a description of the relaxed public ledger.

2.1.1 Public Ledger with Chaining

Our ideal ledger posts an arbitrary string S as part of specific a *chain* of posted values. As an abstraction, we will require each post to identify a *chain identifier* CID. While the host may generate many such identifiers (and thus create an arbitrary number of distinct chains), our abstraction assumes that other parties (*e.g.*, other host machines) will not be allowed to post under the same identifier. The exact nature of the identifier CID depends on the specific Ledger instantiation, which we discuss in detail in Section 2.1.2.1.

The advantage of our interface is that similar checks and chaining are natural properties to achieve when using blockchain-based consensus systems to instantiate the ledger, since many consensus systems perform the necessary checks as part of their consensus logic. Indeed, we can significantly reduce

the cost of deploying our system by using existing ledger systems, including Bitcoin and Ethereum, as they provide these capabilities already, as we discuss in Section 2.1.2.1.

We now define our ledger abstraction, parameterized by a verification function Verify, which has the following interface.[1] Let $\mathsf{H_L}$ be a collision-resistant hash function:

- $\mathcal{L}^{\mathsf{Verify}}.\mathsf{Post}(\mathsf{Data}, \mathsf{CID}) \rightarrow (\mathsf{post}, \pi_{\mathsf{publish}})$.

  When a party wishes to post a string Data onto the chain identified by CID, the ledger constructs a data structure $\mathsf{post} = (\mathsf{PrevHash}, \mathsf{CID}, \mathsf{Data})$ by performing the following steps:

  1. It finds the most recent $\mathsf{post}_{\mathsf{prev}}$ on the Ledger that is associated with CID (if one exists).
  2. If $\mathsf{post}_{\mathsf{prev}}$ was found, it sets $\mathsf{post.PrevHash} \leftarrow \mathsf{post}_{\mathsf{prev}}.\mathsf{Hash}$. Otherwise it sets $\mathsf{post.PrevHash} \leftarrow (\textbf{Root:}\ \mathsf{CID})$, where this labeling uniquely identifies it as first post associated with CID.
  3. It sets $\mathsf{post.Data} \leftarrow \mathsf{Data}$.
  4. It sets $\mathsf{post.Hash} \leftarrow \mathsf{H_L}(\mathsf{post.Data} \| \mathsf{post.PrevHash})$.
  5. It records $(\mathsf{post}, \mathsf{CID})$ on the public ledger.

  The ledger computes an authentication tag π_{publish} over the entire structure post and returns $(\mathsf{post}, \pi_{\mathsf{publish}})$ such that $\mathsf{Verify}(\mathsf{post}, \pi_{\mathsf{publish}}) = 1$.

[1] We omit the ledger *setup* algorithm for this description, although many practical instantiations will include some form of setup or key generation.

- $\mathcal{L}^{\mathsf{Verify}}.\mathsf{GetChain}(\mathsf{CID}) \rightarrow (\mathsf{post}_i, \pi_{\mathsf{publish},i})_{i \in *}$.

  When a party wishes to read from the ledger, they specify a chain. This call then returns all posts associated with that chain, along with their authenticators.

We require that it is difficult to construct a pair $(S, \pi_{\mathsf{publish}})$ such that $\mathsf{Verify}(S, \pi_{\mathsf{publish}}) = 1$ except as the result of a call to $\mathcal{L}^{\mathsf{Verify}}.\mathsf{Post}(\cdot, \cdot)$. Intuitively we refer to this definition as SUF-AUTH. This definition is analogous to the SUF-CMA definition used for signatures. Indeed, looking forward, these signatures will be one of the techniques used to construct the authentication tag.

Remark. We note that the functionality of the above ledger can be simulated from a more simple ledger that has no conception of chaining. For instance, a local agent can retrieve the *full* ledger contents L. Each string in the chain can be tagged, with either an identifier or the public key of a digital signature scheme (if only a single player should be able to post to each chain). Then, using the full ledger contents, the agent can compute Hash_{i-1} and Hash_i locally, performing any hashing required.

Security and finality of the Ledger. Informally, we require that it is difficult to construct a new pair $(S, \pi_{\mathsf{publish}})$ such that $\mathsf{Verify}(S, \pi_{\mathsf{publish}}) = 1$ except as the result of a call to $\mathcal{L}^{\mathsf{Verify}}.\mathsf{Post}(\cdot, \cdot)$, even after the adversary has received many authenticator values on chosen strings. We refer to this definition as SUF-AUTH, and it is analogous to the SUF-CMA definition used for signatures. We note that in some of our proofs we will assume an oracle that produces authentication tags, optionally without actually posting strings to a real ledger. For example,

in a ledger based on signatures, our proofs might assume the existence of a signing oracle that produces signatures on chosen messages. When using "proof of work" ledgers, the authenticators have economic security instead of cryptographic security; we discuss this further in Section 2.1.2.1.

2.1.2 Relaxed Public Ledger

We now consider a more relaxed version of the public ledger that does not have the chaining primitive. We will be using this simplified primitive in Chapters 4 and 5. The ledger ideal functionality is given below (and a UC version of it is given in Figure 2.1). This functionality allows users to post arbitrary information to the ledger; this data is associated with a particular index on the ledger, with which any user can retrieve the original data as well as a proof of publication. As before, our functionality encodes a notion we refer to as *ledger unforgeability*, which requires that there exists an algorithm to verify a authenticator that a message has been posted to the ledger, and that adversaries cannot forge this authenticator.

As noted above, it is possible to realize chaining on these relaxed public ledger using tagging and local simulation. We now define the ledger abstraction, parameterized by a verification function Verify:

- $\mathcal{L}^{\mathsf{Verify}}.\mathsf{GetCounter}() \to t$.

  This algorithm allows any party to retrieve the current ledger counter $t \in \mathbb{N}$.

- $\mathcal{L}^{\mathsf{Verify}}.\mathsf{Post}(x) \to (t, x, \pi_{\mathsf{publish}})$.

Any party can post some bit-string x to the ledger. The ledger increments the t by 1, computes the proof of publication π_{publish} on $(t||x)$ such that $\mathsf{Verify}((t\|x), \pi_{\mathsf{publish}}) = 1$. The ledger then adds the entry $(t, x, \pi_{\mathsf{publish}})$ to the set of entries T and respond with $(t, x, \pi_{\mathsf{publish}})$

- $\mathcal{L}^{\mathsf{Verify}}.\mathsf{GetVal}(t) \rightarrow \{(t, x, \pi_{\mathsf{publish}}), \perp\}$.

  To retrieve any entry in the set of entries on the ledger, a party calls GetVal on the index t. If $\mathcal{L}^{\mathsf{Verify}}.\mathsf{GetCounter}() \geq t$, the ledger returns the appropriate entry in the set T. Otherwise, the ledger returns $\perp$.

As before, we require that it is difficult to construct a new pair $(s, \pi_{\mathsf{publish}})$ such that $\mathsf{Verify}(s, \pi_{\mathsf{publish}}) = 1$ except as the result of a call to $\mathcal{L}^{\mathsf{Verify}}.\mathsf{Post}(\cdot)$.

In Chapter 5, we will be using the universally-composable (UC) framework from (Canetti, 2001). We give a UC-style definition for this ledger functionality in Figure 2.1. This definition is also parameterized by a verify function $\mathsf{Verify}(\cdot, \cdot)$.

Functionality $\mathcal{L}^{\mathsf{Verify}}$

GetCounter: Upon receiving (GetCounter) from any party, return t.

Post: Upon receiving (Post, x), the trusted party increments t by 1, computes the proof of publication π_{publish} on $(t||x)$ such that $\mathsf{Verify}((t\|x), \pi_{\mathsf{publish}}) = 1$. Add the entry $(t, x, \pi_{\mathsf{publish}})$ to the entry table T. Respond with $(t, x, \pi_{\mathsf{publish}})$

GetVal: Upon receiving (GetVal, t), check if there is an entry $(t, x, \pi_{\mathsf{publish}})$ in the entry table T. If not, return $\perp$. Otherwise, return $(t, x, \pi_{\mathsf{publish}})$.

Figure 2.1: Ideal relaxed public ledger functionality for a proof-of-publication ledger.

2.1.2.1 Realizing the Ledger

There are many different systems that may be used to instantiate the ledger. In principle, any stateful centralized server capable of producing SUF-CMA signatures can be used for this purpose. We describe four potential realizations in detail: unstructured, centralized public ledgers such as Certificate Transparency (*Certificate Transparency* 2018), proof of work blockchains like Bitcoin, smart contract systems like Ethereum, and private blockchains.

Certificate Transparency A number of browsers have begun to mandate *Certificate Transparency* (CT) proofs for TLS certificates (*Certificate Transparency* 2018). In these systems, every CA-issued certificate is included in a public log, which is published and maintained by a central authority such as Google. Every certificate in the log is included as a leaf in a Merkle tree, and the signed root and associated membership proofs are distributed by the log maintainer.

Provided that the log maintainer is trustworthy, this system forms a public bedger with strong cryptographic security. The inclusion of a certificate can be verified by any party who has the maintainer's public key, while the tree location can be viewed as a unique identifier of the posted certificate. Because many certificate authorities support CT, the ability to programmatically submit certificate signing requests, using services like LetsEncrypt, allows us to use CT as a log for any arbitrary data that can be incorporated into an X.509 certificate. In our presentation we implicitly assume that the Enclave can verify CT inclusion proofs from a specific log *i.e.,* that it has been provisioned with a copy of the log maintainer's public verification key.

A limitation of the CT realization is that, to implement our Ledger functionality of Section 2.1, we require a way to ensure that the PrevHash field of each record truly does identify the previous entry in the log. Unfortunately, the current instantiation of CT does not guarantee this; instead, the enclave must read *the entire certificate log* to verify that no interceding entries exist. This makes CT less bandwidth-efficient than the other realizations.

Bitcoin and Proof-of-work Blockchains Public blockchains, embodied most prominently by Bitcoin, are designed to facilitate distributed consensus as to the contents of a ledger. In these systems, new blocks of transactions are added to the ledger each time a participant solves a costly *proof of work* (PoW), which typically involves solving a hash puzzle over the block contents. These PoW solutions are publicly verifiable, and can be used as a form of "economic" authentication tag over the block contents: that is, while these tags can be forged, the financial cost of doing so is extremely high. Moreover, because blocks are computed in sequence, a sub-chain of n blocks (which we refer to as a "fragment") will include n chained proofs-of-work, resulting in a linear increase in the cost of forging the first block in the fragment.

The remaining properties of our ledger are provided as follows: in Bitcoin, transactions are already uniquely identified by their hash, and each transaction (by consensus rules) must identify a previous transaction hashes as an *input*. Similarly, due to double spending protections in the consensus rules, there cannot be two transactions that share a previous input. Finally, we can encode arbitrary data into the transaction using the `OP_RETURN` script (*Bitcoin Wiki: Script* 2018).

Analyzing the cost of forging blockchain fragments. Proof-of-work blockchains do not provide a cryptographic guarantee of unforgeability. To provide some understanding of the cost of forging in these systems, we can examine the economics of real proof-of-work blockchains. We propose argue that the cost of forging an authenticator can can be determined based on from block reward offered by a proof-of-work cryptocurrency, assuming that the market is liquid and reasonably efficient.

In currencies such as Bitcoin, the reward for producing a valid proof-of-work block is denominated in the blockchain currency, which has a floating value with respect to currencies such as the dollar. Critically, because each instance of a PoW puzzle in the real blockchain is based on the preceding block, an adversarial miner must choose at mining time if they want to mine on the blockchain or attempt to forge a block for use in the ELI scheme; their work cannot do double duty. Thus we can calculate the opportunity cost of forgoing normal mining in order to attack an ELI system: the real cost of forging a block is at least the value of a block reward. Similarly, the cost of forging a blockchain fragments of length n is at least n times the block reward. At present, the cost of forging a fragment of length 7 would be 87.5 BTC.

Remark. This simple analysis ignores that a single blockchain fragment may be used by multiple instances of a given enclave. This admits the possibility that an attacker with significant capital might amortize this cost by spreading it across many instances. Indeed, if amortized over a sufficient number of forged ledger posts, this fixed cost could be reduced. For scenarios where we expect sufficient instances for this attack to be practical, it is necessary to rate

limit the number of ledger posts included in a given block that the enclave will accept results from.

Ethereum and Smart Contract Systems A very natural realization of our ledger system is a smart contract systems such as Ethereum (Nakamoto, 2008; *The Ethereum Project* 2018). Smart contract systems enable distributed public computation on the blockchain. Typically, a program is posted to some specific address on the blockchain. When a user submits a transaction to the associated address, the code is executed and appropriate state is updated. As noted previously, our system allows for smart contracts with private data, which is impossible on current implementations of smart contract system.

Private Blockchains Many recent systems such as Hyperledger (Hyperledger, 2017) implement private smart contracts by constructing a shared blockchain among a set of dedicated nodes. In some instantiations, the parties forgo the use of proof-of-work in favor of using digital signatures and trusted hardware to identify the party who writes the next block (Intel Corporation, 2018). Private blockchains represent a compromise between centralized systems such as CT and proof-of-work blockchains. They are able to use digital signatures to produce ledger authentication tags so the security is not economic in nature. Moreover, the ledger can be constructed to provide efficient rules for ledger state updates, which enables an efficient realization of our model.

2.2 Cryptographic Primitives

2.2.1 Pseudorandom Functions

We now give a definition for pseudorandom functions (Goldreich, Goldwasser, and Micali, 1984).

Definition 1 (Pseudorandom Function) *A pseudorandom function family is a family of functions indexed by an λ-bit key $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\ell$, for $\ell = \mathrm{poly}(\lambda)$. We may also write this as $\mathsf{PRF}_K(m) \to r$, where $K \in \{0,1\}^\lambda, m \in \{0,1\}^*$, and $r \in \{0,1\}^\ell$. A pseudorandom function satisfies the property*

$$\Pr\left[b \leftarrow \mathcal{A}^{O_b(\cdot)}(1^\lambda) \,\middle|\, b \xleftarrow{\$} \{0,1\}\right] < \mathsf{negl}(1^\lambda)$$

Where $O_b(\cdot)$ is an oracle. If $b = 1$, upon receiving some query m, if there exists an entry (m, r) in the responses table, $O_b(\cdot)$ returns r. Otherwise, $O_b(\cdot)$ samples uniform random bit strings $r \in \{0,1\}^\ell$, adds (m, r) to the entry table, and returns r. If $b = 0$, $O_b(\cdot)$ initially samples $K \xleftarrow{\$} \{0,1\}^\lambda$, and responds to queries m with $\mathsf{PRF}_K(m)$.

2.2.2 Authenticated Encryption With Associated Data

We require a symmetric authenticated encryption scheme consisting of the algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$. This notion is most commonly captured in practice as Authenticated Encryption with Associated Data (Rogaway, 2002). We will not require the authenticated data, and so we just define a simplified authenticated encryption here (Goldwasser and Micali, 1984; Bellare et al., 1997; Bellare and Rogaway, 2000; Rogaway et al., 2001).

Definition 2 (Authenticated Encryption) *A symmetric authenticated encryption encryption scheme* Π_{AE} *is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *defined as follows:*

- $\mathsf{KeyGen}(1^\lambda)$ *generates a key* $k \in \{0,1\}^\kappa$
- $\mathsf{Enc}(k, m)$ *takes in a key* $k \in \{0,1\}^\kappa$ *and a message* $m \in \{0,1\}^*$ *and outputs a ciphertext* c*. This algorithm is either randomized, or can take in explicit randomness* r*.*
- $\mathsf{Dec}(k, c)$ *takes in a key* $k \in \{0,1\}^\kappa$ *and a ciphertext* $c \in \{0,1\}^*$ *and outputs either a message* m *or the error symbol* $\perp$

We require that these algorithms satisfy the following property

$$\Pr\left[\mathsf{Dec}(k, c^*) \neq \perp \,\middle|\, k \leftarrow \mathsf{KeyGen}(1^\lambda), c^* \leftarrow \mathcal{A}^{\mathsf{Enc}(k,\cdot),\mathsf{Dec}(k,\cdot)}(1^\lambda)\right] < \mathsf{negl}(1^\lambda)$$

Where $\mathsf{Enc}(k, \cdot)$ *and* $\mathsf{Dec}(k, \cdot)$ *are oracles and* c^* *is a ciphertext not output by the encryption oracle.*

For simplicity we further define the specialized algorithm $\mathsf{Enc}^{\mathsf{pad}}$ as one that *pads* the plaintext to (a maximum state size) n bits prior to encrypting it, and define $\mathsf{Dec}^{\mathsf{unpad}}$ as removing this padding.

2.2.3 CCA Secure Public Key Encryption

We now define chosen-ciphertext secure public key encryption

Definition 3 (CCA Public Key Encryption) *A public key encryption scheme* Π_{PKE} *is a tuple of algorithms* $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *defined as follows*

- $\mathsf{KeyGen}(1^\lambda)$ *generates a public, private keypair* (pk, sk).
- $\mathsf{Enc}(pk, m)$ *takes in a public key pk and a message m and outputs a ciphertext c.*
- $\mathsf{Dec}(sk, c)$ *takes in a secret key sk and a ciphertext c and outputs either a message m or the error symbol* $\perp$.

We say that a Π_{PKE} *satisfies chosen-ciphertext security if*

$$\Pr\left[\begin{array}{l|c} & (pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda), \\ b \leftarrow \mathcal{A}_2^{\mathsf{Dec}_2(sk,\cdot)}(1^\lambda, pk, c^*, aux) & m_0, m_1, aux \leftarrow \mathcal{A}_1^{\mathsf{Dec}_1(sk,\cdot)}, \\ & b \xleftarrow{\$} \{0,1\}, c^* \leftarrow \mathsf{Enc}(pk, m_b) \end{array}\right] < \mathsf{negl}(1^\lambda)$$

where $\mathsf{Dec}_2(sk, \cdot)$ *is a decryption oracle, with the additional requirment that on input* c^**, it responds* $\perp$.

2.2.4 Lossy Encryption

Lossy encryption (Bellare, Hofheinz, and Yilek, 2009) is an encryption application of lossy trapdoor functions, which were introduced by Peikert and Waters (Peikert and Waters, 2008). Intuitively, lossy encryption is a public key encryption scheme that has an algorithm to generate *lossy keys* that are computationally indistinguishable from normal keys. When encrypting with

these lossy keys, the resulting ciphertext contains no information about the plaintext.

Definition 4 (Lossy Encryption) A lossy public-key encryption scheme Π_{Lossy} is a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{KeyGen}_{\mathsf{loss}}, \mathsf{Enc}, \mathsf{Dec})$ defined as

- $\mathsf{KeyGen}(1^\lambda)$ generates an injective keypair (pk, sk)
- $\mathsf{KeyGen}_{\mathsf{loss}}(1^\lambda)$ generates a lossy keypair $(pk, \cdot)$
- $\mathsf{Enc}(pk, m)$ takes in a public key pk and a plaintext message m and outputs a ciphertext c
- $\mathsf{Dec}(sk, c)$ takes in a secret key sk and a ciphertext c and either outputs $\perp$ or the message m

We require that the above algorithms satisfies the following properties:

- **Correctness on real keys**: For all messages m, it should hold that

$$\Pr\left[m = \mathsf{Dec}(sk, \mathsf{Enc}(pk, m)) \,\middle|\, (pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)\right] = 1$$

- **Lossiness on lossy keys**: for all $(pk, \cdot) \leftarrow \mathsf{KeyGen}_{\mathsf{loss}}(1^\lambda)$ and m_0, m_1 such that $|m_0| = |m_1|$,

$$\mathsf{Enc}(pk, id, m_0) \overset{s}{\approx} \mathsf{Enc}(pk, id, m_1)$$

- **Indistinguishability of keys**: Finally, over all random coins, it should hold that

$$\mathsf{KeyGen}(1^\lambda) \overset{c}{\approx} \mathsf{KeyGen}_{\mathsf{loss}}(1^\lambda)$$

In our next definition, we will require the instantiation of lossy encryption presented in (Bellare, Hofheinz, and Yilek, 2009). For completeness, we include it here. Let $\mathbb{G}$ be a cyclic group of order p (where p is prime) in which the DDH assumption holds. let g,h denote generators of $\mathbb{G}$.

- $\mathsf{KeyGen}(1^\lambda)$: $g \xleftarrow{\$} \mathbb{G}, x, r \leftarrow \mathbb{Z}_p$. $pk \leftarrow (g, g^r, g^x, g^r x)$ and $sk \leftarrow x$. Output (pk, sk).
- $\mathsf{KeyGen}_{\mathsf{loss}}(1^\lambda)$: $g \xleftarrow{\$} \mathbb{G}, x, y, r \leftarrow \mathbb{Z}_p$ such that $x \neq y$. $pk \leftarrow (g, g^r, g^x, g^r y)$ and $sk \leftarrow \perp$. Output (pk, sk).
- $\mathsf{Enc}(pk, m)$: $(g, h, \hat{g}, \hat{h}) \leftarrow pk$, $r_1, r_2 \leftarrow \mathbb{Z}_p$. $c \leftarrow (g^{r_1} h^{r_2}, (\hat{g})^{r_1} (\hat{h})^{r_2} \cdot m)$. Output c.
- $\mathsf{Dec}(sk, c)$: $(c_0, c_1) \leftarrow c$, $m \leftarrow \frac{c_1}{c_0^{sk}}$

For proof that this scheme is indeed a lossy encryption scheme, please see (Bellare, Hofheinz, and Yilek, 2009).

2.2.4.1 Lossy-Tag Encryption

In Chapter 5 we require a specific generalization of lossy encryption we call lossy-tag encryption (LTE). Intuitively, this is an encryption scheme with a single public key in which encryption takes as input a "tag" in addition to the public key and plaintext. Encrypting under a tag from a specific subset will produce in an injective ciphertext, while the remaining tags will produce a lossy ciphertext. This notion is closely related to numerous previous works, including lossy encryption (Bellare, Hofheinz, and Yilek, 2009), lossy trapdoor

functions (Peikert and Waters, 2008), identity-based lossy trapdoor functions (Bellare et al., 2012) all-but-one functions (Peikert and Waters, 2008), and all-but-n functions (Hemenway et al., 2011). We define lossy-tag encryption formally as follows:

Definition 5 (Lossy Tag Encryption) A lossy-tag encryption (LTE) scheme with respect to a tag space $\mathbb{T}$ consists of a tuple of algorithms $(\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ defined as follows:

- $\mathsf{KeyGen}(1^\lambda, \mathcal{T})$ takes in a set of tags $\mathcal{T} \subset \mathbb{T}$ of polynomial size in λ and outputs a public key *params* and a secret key *msk*.
- $\mathsf{Enc}(params, id, m)$ encrypts the message m under the public key *params* and $id \in \mathbb{T}$ to produce ciphertext c.
- $\mathsf{Dec}(msk, id, c)$ takes in the secret key *msk*, $id \in \mathbb{T}$ and a ciphertext c and either outputs a message m or $\perp$.

We require that the above algorithms satisfy the follow properties

- **Correctness on injective tags**:

$$\Pr\left[\begin{array}{c} m = \mathsf{Dec}(msk, id, \mathsf{Enc}(params, id, m)) \end{array} \middle| \begin{array}{c} id \in \mathcal{T} \\ \\ (params, msk) \leftarrow \mathsf{KeyGen}(1^\lambda, \mathcal{T}) \end{array}\right] = 1$$

- **Lossiness on lossy tags**: for all messages m_0, m_1 and all sets $\mathcal{T}$, if $id \notin \mathcal{T}$ and $(params, msk) \in \mathsf{KeyGen}(1^\lambda, \mathcal{T})$, then

$$\mathsf{Enc}(params, id, m_0) \overset{s}{\approx} \mathsf{Enc}(params, id, m_1)$$

– **Indistinguishability of tag sets**: for all sets $\mathcal{T}_0 \neq \mathcal{T}_1$ such that $|\mathcal{T}_0| = |\mathcal{T}_1|$,

$$\mathsf{KeyGen}(1^\lambda, \mathcal{T}_0) \overset{c}{\approx} \mathsf{KeyGen}(1^\lambda, \mathcal{T}_1)$$

A stronger version of this definition could remove the requirement that $|\mathcal{T}_0| = |\mathcal{T}_1|$ for indistinguishability of tag sets. For our constructions, we do not concern ourselves with this leakage.

Realizing lossy-tag encryption: When the size of $\mathbb{T}$ is polynomial in the security parameter, it is trivial to realize lossy-tag encryption from standard lossy encryption (Bellare, Hofheinz, and Yilek, 2009) simply by generating one lossy keypair to represent each "tag". However, even for small sets $\mathbb{T}$ this may produce unreasonably large public keys. In this work, we present a direct instantiation of a lossy-tag encryption scheme based on DDH, such that the public parameters *params* that are linear in $|\mathcal{T}|$.

Let $\mathbb{G}$ be a cyclic group of order p. Define $pk = (g, h, \tilde{g}, \tilde{h}) \in \mathbb{G}^4$, and $\mathsf{DoubleEncrypt}(pk, m; r_1, r_2)$ to output $(g^{r_1}h^{r_2}, \tilde{g}^{r_1}\tilde{h}^{r_2} \cdot m)$. As noted in (Bellare, Hofheinz, and Yilek, 2009), if pk is a DDH tuple then this encryption is injective, but if pk is a random tuple then the encryption is statistically lossy. We now present a construction Π_{LTE} for lossy-tag encryption as follows:

- $\mathsf{KeyGen}(params, \mathcal{T}) \rightarrow (params, msk)$. Sample $params = p, \mathcal{G}, g, h, \hat{g}, \hat{h}$ where $\mathcal{G}$ has order p and $g, h, \hat{g}, \hat{h}$ are generators of $\mathcal{G}$. Sample a random polynomial $A(x)$ in $\mathbb{Z}_p$ of degree $k = |\mathcal{T}|$ such that for each $s \in \mathcal{T}, A(x) = 0$. Then compute $B(x) = d_2 A(x)$ for some constant $d_2 \neq \frac{d_0}{d_1}$. Let α_i be the i^{th} coefficient of $A(x)$ and β_i be the i^{th} coefficient of

$B(x)$. Use rejection sampling to sample random $\eta_0 \ldots \eta_k$ such then when η_i is interpreted as the i^{th} coefficient of a polynomial $E(x)$, $E(id) \neq 0$ for all $id \in \mathcal{T}$. Compute $params = ((g^{\eta_k}\hat{g}^{\alpha_k}, \ldots, g^{\eta_0}\hat{g}^{\alpha_0}), (h^{\eta_k}\hat{h}^{\beta_k}, \ldots, h^{\eta_0}\hat{h}^{\beta_0}))$. Compute $msk = (\eta_0, \eta_1, \ldots, \eta_k)$ and output $params, msk$.

- $\mathsf{Enc}(params, params, id, m) \rightarrow c$.
  - Parse
  $$((g^{\eta_k}\hat{g}^{\alpha_k}, \ldots, g^{\eta_0}\hat{g}^{\alpha_0}), (h^{\eta_k}\hat{h}^{\beta_k}, \ldots, h^{\eta_0}\hat{h}^{\beta_0})) \leftarrow params$$
  - Compute the user public key
  $$pk = (g, h, \prod_{i=0}^{k}(g^{\eta_i}\hat{g}^{\alpha_i})^{id^i}, \prod_{i=0}^{k}(h^{\eta_i}\hat{h}^{\beta_i})^{id^i})$$
  - Sample $r_1, r_2 \leftarrow \mathbb{Z}_p$
  - Compute and return $\mathsf{DoubleEncrypt}(pk, m; r_1, r_2)$
- $\mathsf{Dec}(params, msk, id, c) \rightarrow m$ Parse $(\eta_0, \eta_1, \ldots, \eta_k) \leftarrow msk$ Parse $(c_1, c_2) \leftarrow c$ Compute $y = \sum_{i=0}^{k} \eta_0 (id)^k$ Compute and return $m = \frac{c_2}{c_1^y}$.

We now prove that Π_{LTE} above realizes the the functionality of lossy-tag encryption. To do so, we recall the construction of a lossy encryption scheme in Section 4.1 of (Bellare, Hofheinz, and Yilek, 2009). As mentioned above, they observe that ElGamal double encryption is injective when the public key has the structure (g, h, g^x, h^x), but is lossy when the public key has the structure (g, h, g^x, h^y), for $x \neq y$. For completeness, we have included this algorithm above in Section 2.2.4

Correctness on injective tags. For injective tags, $A(x) = B(x) = 0$. Recall that the public key used during encryption is computed as $(g, h, \prod_{i=0}^{k}(g^{\eta_i}\hat{g}^{\alpha_i})^{id^i}$, $\prod_{i=0}^{k}(h^{\eta_i}\hat{h}^{\beta_i})^{id^i})$. Written another way, this is $(g, h, g^{E(id)}\hat{g}^{A(id)}, h^{E(id)}\hat{h}^{B(id)})$. Because $A(x) = B(x) = 0$, the public key is $(g, h, g^{E(id)}, h^{E(id)})$, where $E(id)$ is non-zero. Note that this form is the same form as an injective key from (Bellare, Hofheinz, and Yilek, 2009), so the resulting ciphertext is injective with corresponding private key $E(id)$.

Lossiness on lossy tags. For lossy tags, $A(x) \neq B(x) \neq 0$. This can be observed because $B(x) = kA(x)$, and there are at most $|\mathcal{T}|$ zeros of a degree $|\mathcal{T}|$ polynomial, and all all these zeros were set to be the injective tags. The public key used for encryption is, as before, $(g, h, g^{E(id)}\hat{g}^{A(id)}, h^{E(id)}\hat{h}^{B(id)})$. Without loss of generality, the public key can then be written as $(g, h, g^{E(id)+d_0A(id)}, h^{E(id)+d_1B(id)})$. Note that because $B(\cdot)$ was sampled such that $d_2A(x) = B(x)$, and $\frac{d_0}{d_1} \neq d_2$, then $E(id) + d_0A(id) \neq E(id) + d_1B(id)$. Thus this public key is structured exactly like the lossy key from (Bellare, Hofheinz, and Yilek, 2009), so the resulting ciphertext is lossy.

Indistinguishability of tag sets. Due to the key indistinguishability of (Bellare, Hofheinz, and Yilek, 2009), it is clear that a lossy key and an injective key, when computed during encryption, are statistically indistinguishable. All that remains to argue is that the public parameters leak no information about the tag set besides its size (note that the size of *params* trivially leaks $|\mathcal{T}|$). Notice that it is sufficient to show that this property holds when two sets differ in only a single tag, as a straightforward hybrid argument in which a single tag is swapped in each hybrid can generalize the result. Next, notice that

each element in *params* is formed like a Pendersen commitment (Pedersen, 1992) to α_i or β_i. Thus, it is clear to see that if there exists an adversary that can distinguish between sets, it can be used to break the hiding property of Pendersen commitments.

2.2.5 Secure Multiparty Computation

Secure multiparty computation (MPC) (Yao, 1986; Goldreich, Micali, and Wigderson, 1987; Chaum, Crépeau, and Damgård, 1988; Ben-Or, Goldwasser, and Wigderson, 1988) allows a set of n player compute a function over their joint inputs without sharing any information about those inputs. A MPC protocol is a protocol executed by n number of parties $P_1, \cdots, P_n$ for a n-party functionality F. We allow for parties to exchange messages simultaneously. In every round, every party is allowed to broadcast messages to all parties. We require that at the end of the protocol, all the parties receive the output $F(x_1, \ldots, x_n)$, where x_i is the i^{th} party's input.[2] There are many properties that a MPC protocol can have. We begin by giving the classic simulation-based formalization of MPC with security with abort.

Ideal World. We start by describing the ideal world experiment where n parties $P_1, \cdots, P_n$ interact with an ideal functionality for computing a function F. An adversary may corrupt any subset $\mathcal{P}^{\mathcal{A}} \subset \mathcal{P}$ of the parties. We denote the honest parties by $\mathcal{H}$.

Inputs: Each party P_i obtains an initial input x_i. The adversary $\mathcal{S}$ is given

[2]One can also consider asymmetric functionalities where every party receives a different output. Since there are generic transformations from the symmetric case to the asymmetric case, we only consider symmetric functionalities for simplicity of exposition.

auxiliary input z. $\mathcal{S}$ selects a subset of the parties $\mathcal{P}^{\mathcal{A}} \subset \mathcal{P}$ to corrupt, and is given the inputs x_k of each party $P_k \in \mathcal{P}^{\mathcal{A}}$.

Sending inputs to trusted party: Each honest party P_i sends its input x_i to the trusted party. For each corrupted party $P_i \in \mathcal{P}^{\mathcal{A}}$, the adversary may select any value x_i^* and send it to the ideal functionality.

Trusted party computes output: Let $x_1^*, \ldots, x_n^*$ be the inputs that were sent to the trusted party. If any of the received inputs were $\perp$, then the trusted party sends $\perp$ to all the parties.

Adaptive Abort: The trusted party sends $F(x_1^*, \ldots, x_n^*)$ to the adversary. If the adversary responds with $\perp$, the trusted party sends $\perp$ to the honest parties. Otherwise, the trusted party sends $F(x_1^*, \ldots, x_n^*)$ to the honest parties.

Outputs: Honest parties output the function output they obtained from the ideal functionality. Malicious parties may output an arbitrary PPT function of the adversary's view.

The overall output of the ideal-world experiment consists of the outputs of all parties. For any ideal-world adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$, input vector $\vec{x}$, and security parameter λ, we denote the output of the corresponding ideal-world experiment by $\mathsf{IDEAL}_{\mathcal{S},F}(1^\lambda, \vec{x}, z)$.

Real World. The real world execution begins by an adversary $\mathcal{A}$ selecting any arbitrary subset of parties $\mathcal{P}^{\mathcal{A}} \subset \mathcal{P}$ to corrupt.[3] The parties then engage in an

[3] In our work, we concern ourselves only with static corruption models.

execution of a real n-party protocol Π. Throughout the execution of Π, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties, and may follow an arbitrary polynomial-time strategy. In contrast, the honest parties follow the instructions of Π.

At the conclusion of all the update phases, each honest party P_i outputs whatever output it received from the computation. Malicious parties may output an arbitrary PPT function of the view of $\mathcal{A}$.

For any adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$, input vector $\vec{x}$, and security parameter λ, we denote the output of the MPC protocol Π by $\mathsf{REAL}_{\mathcal{A},\Pi}(1^\lambda, \vec{x}, z)$

Definition 6 *A protocol Π is a secure* n*-party protocol computing F if for every PPT adversary $\mathcal{A}$ in the real world, there exists a PPT adversary $\mathcal{S}$ corrupting the same parties in the ideal world such that for every initial input vector $\vec{x}$, every auxiliary input z, it holds that*

$$\mathsf{IDEAL}_{\mathcal{S},F}\left(1^\lambda, \vec{x}, z\right) \approx_c \mathsf{REAL}_{\mathcal{A},\Pi}\left(1^\lambda, \vec{x}, z\right).$$

2.2.5.1 Fair Multiparty Computation

A secure fair multi-party computation protocol is a slight modification of the security with abort definition given above. The real-world experiment stays the same as above. The ideal world experiment is modified below:

Inputs: Each party P_i obtains an initial input x_i. The adversary $\mathcal{S}$ is given auxiliary input z. $\mathcal{S}$ selects a subset of the parties $\mathcal{P}^{\mathcal{A}} \subset \mathcal{P}$ to corrupt, and is given the inputs x_k of each party $P_k \in \mathcal{P}^{\mathcal{A}}$.

Sending inputs to trusted party: Each honest party P_i sends its input x_i to the trusted party. For each corrupted party $P_i \in \mathcal{P}^{\mathcal{A}}$, the adversary may select any value x_i^* and send it to the ideal functionality.

Trusted party computes output: Let $x_1^*, \ldots, x_n^*$ be the inputs that were sent to the trusted party. If any of the received inputs were $\perp$, then the trusted party sends $\perp$ to all the parties. Otherwise, the trusted party sends $F(x_1^*, \ldots, x_n^*)$ to all parties

Outputs: Honest parties output the function output they obtained from the ideal functionality. Malicious parties may output an arbitrary PPT function of the adversary's view.

The overall output of the ideal-world experiment consists of the outputs of all parties. For any ideal-world adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$, input vector $\vec{x}$, and security parameter λ, we denote the output of the corresponding ideal-world experiment by $\mathsf{IDEAL}^{\mathsf{fair}}_{\mathcal{S},F}(1^\lambda, \vec{x}, z)$.

MPC with Complete Fairness. We say that a protocol Π is a secure protocol if any adversary, who corrupts a subset of parties and runs the protocol with honest parties, gains *no information* about the inputs of the honest parties beyond the protocol output.

Definition 7 *A protocol* Π *is a secure* n*-party protocol computing* F *with complete fairness if for every PPT adversary* $\mathcal{A}$ *in the real world, there exists a PPT adversary* $\mathcal{S}$ *corrupting the same parties in the ideal world such that for every initial input*

vector $\vec{x}$, every auxiliary input z, it holds that

$$\mathsf{IDEAL}^{\mathsf{fair}}_{\mathcal{S},F}\left(1^\lambda, \vec{x}, z\right) \approx_c \mathsf{REAL}_{\mathcal{A},\Pi}\left(1^\lambda, \vec{x}, z\right).$$

2.2.6 Witness Encryption

In this section, we define witness encryption (Gentry, Lewko, and Waters, 2014).

Definition 8 (Witness Encryption) *A witness encryption $\Pi_{WE} = (\mathsf{Enc}, \mathsf{Dec})$ for a NP language $\mathcal{L}$ consists of the following algorithms:*

- **Encryption,** $\mathsf{Enc}(1^\lambda, x, m)$*: On input instance x, message m and it outputs a ciphertext* CT*.*
- **Decryption,** $\mathsf{Dec}(\mathsf{CT}, w)$*: On input ciphertext* CT *and witness w, it outputs m'.*

We require that the following properties hold:

- **Correctness***: For every $x \in \mathcal{L}$, let w be such that $(x, w) \in R$, for every $m \in \{0,1\}$,*

$$\mathsf{Pr}[m \leftarrow \mathsf{Dec}(\mathsf{Enc}(x, m), w)] = 1$$

- **Message Indistinguishability***: For every PPT adversary $\mathcal{A}$, there is a negligible function ε, such that for every $x \notin \mathcal{L}$ the following holds:*

$$\left|\mathsf{Pr}[1 \leftarrow \mathcal{A}(1^\lambda, \mathsf{Enc}(x, m_0))] - \mathsf{Pr}[1 \leftarrow \mathcal{A}(1^\lambda, \mathsf{Enc}(x, m_1))]\right| \leq \varepsilon.$$

2.2.6.1 Extractable Witness Encryption

Some of our constructions require extractable witness encryption (Boyle, Chung, and Pass, 2014), a variant of witness encryption in which the existence of a distinguisher can be used to construct an extractor for the necessary witness (Gentry, Lewko, and Waters, 2014). In the constructions in Chapter 5, all statements we use are in the corresponding language, but finding a witness will be difficult for an adversary.

Definition 9 (Extractable Witness Encryption) *An extractable witness encryption* $\Pi_{WE} = (\mathsf{Enc}, \mathsf{Dec})$ *for a NP language* $\mathcal{L}$ *associated with relation* R *consists of the following algorithms:*

- $\mathsf{Enc}(1^\lambda, x, m)$*: On input instance* x *and message* $m \in \{0,1\}$*, it outputs a ciphertext* c*.*
- $\mathsf{Dec}(c, w)$*: On input ciphertext* c *and witness* w*, it outputs* m'*.*

We require that the above primitive satisfy the following properties:

- **Correctness***: For every* $(x, w) \in R$*, for every* $m \in \{0,1\}$*,*

$$\Pr[m = \mathsf{Dec}(\mathsf{Enc}(1^\lambda, x, m), w)] = 1$$

- **Extractable Security***: For any PPT adversary* $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$*, if:*

$$\Pr\left[\mathcal{A}_1(1^\lambda, c, z) = b \;\middle|\; \begin{array}{c} b \leftarrow \{0,1\} \\ (m_0, m_1, z) \leftarrow \mathcal{A}_0(1^\lambda, x) \\ c \leftarrow \mathsf{Enc}(1^\lambda, x, m_b) \end{array}\right] \geq \frac{1}{2} + \mathsf{negl}(1^\lambda)$$

then there exists a PPT extractor Ext *such that for all auxiliary inputs* aux*:*

$$\Pr\left[w \leftarrow \mathsf{Ext}_{\mathcal{A}}(1^{\lambda}, x, aux) \; s.t. \; : (x,w) \in R\right] \geq \mathsf{negl}(1^{\lambda})$$

We now define the notion of polynomial witness languages.

Definition 10 (Witness Languages) *Consider an NP language* $\mathcal{L}$ *and let* R *be its associated relation. We say that* $\mathcal{L}$ *is a polynomial witness language if there exists a fixed polynomial* p *such that for every* $x \in \mathcal{L}$ *it holds that there exists a size* $p(|x|)$ *set of witnesses* w *such that* $w \in \{0,1\}^{\mathrm{poly}(|x|)}$ *and* $(x,w) \in R$*.*

The following theorem was shown in (Boyle, Chung, and Pass, 2014).

Theorem 1 *Suppose* $\mathcal{L}$ *is a polynomial witness language. Then, witness encryption for* $\mathcal{L}$ *implies extractable witness encryption for* $\mathcal{L}$*.*

2.2.7 Simulation Extractable Non-Interactive Zero Knowledge

In our protocols we require non-interactive zero knowledge proofs of knowledge that are simulation extractable (Sahai, 1999; De Santis et al., 2001).

Definition 11 (Simulation Extractable Non-Interactive Zero Knowledge) A non-interactive zero-knowledge proof of knowledge scheme Π_{NIZK} for a relation $\mathcal{R}$ is a set of algorithms $(\mathsf{ZKSetup}, \mathsf{ZKProve}, \mathsf{ZKVerify}, \mathsf{ZKSimulate})$ defined as follows:

- $\mathsf{ZKSetup}(1^{\lambda})$ returns the common reference string and simulation trapdoor (crs, τ).

– $\mathsf{ZKProve}(crs, x, \omega)$ takes in the common reference string crs, a statement x and a witness ω and outputs a proof π.

– $\mathsf{ZKVerify}(crs, x, \pi)$ takes in the common reference string crs, a statement x and a proof π, and outputs either 1 or 0.

– $\mathsf{ZKSimulate}(crs, \tau, x)$ takes in the common reference string crs, a statement x and the simulation trapdoor τ and outputs a proof π.

We say that a non-interactive zero-knowledge argument of knowledge is simulation extractable if it satisfies the following properties:

– **Completeness:** If a prover has a valid witness, then they can always convince the verifier. More formally, for all relations $\mathcal{R}$ and all x, ω, if $\mathcal{R}(x, \omega) = 1$, then

$$\Pr\left[\mathsf{ZKVerify}(crs, x, \pi) = 1 \;\middle|\; \begin{array}{l}(crs, \tau) \leftarrow \mathsf{ZKSetup}(1^\lambda), \\ \\ \pi \leftarrow \mathsf{ZKProve}(crs, x, \omega)\end{array}\right] = 1$$

– **Perfect Zero knowledge:** A scheme has zero-knowledge if a proof leaks no information beyond that truth of the statement x. We formalize this by saying that an adversary with oracle access to a prover cannot tell if that prover runs the honest algorithm ZKProve or uses the trapdoor and ZKSimulate.

$$\Pr\left[\mathcal{A}^{\mathsf{ZKProve}(crs,\cdot,\cdot)}(crs) = 1 | (crs, \cdot) \leftarrow \mathsf{ZKSetup}(1^\lambda)\right] \overset{s}{\approx}$$

$$\Pr\left[\mathcal{A}^{\mathsf{ZKSimulate}(crs,\tau,\cdot)}(crs) = 1 | (crs, \tau) \leftarrow \mathsf{ZKSetup}(1^\lambda)\right]$$

– **Simulation Extractability:** There exists an extractor Extract such that

$$\Pr\left[\mathcal{R}(x,\omega)=1 \;\middle|\; \begin{array}{c}(crs,\tau)\leftarrow \mathsf{ZKSetup}(1^\lambda),\\ (x,\pi)\leftarrow \mathcal{A}^{\mathsf{ZKSimulate}(crs,\tau,\cdot)}(crs),\\ \omega\leftarrow \mathsf{Extract}(crs,\tau,x,\pi)\end{array}\right]\geq 1-\mathsf{negl}(1^\lambda)$$

It has been shown that realizing this primitive for languages outside BBP requires the common reference string model (Oren, 1987; Goldreich and Oren, 1994; Goldreich and Krawczyk, 1996). We present the common reference string ideal functionality in Figure 2.2, which can be found in Figure 2.2 (Canetti and Fischlin, 2001).

Functionality $\mathcal{F}_{CRS}^{\mathcal{D}}$

$\mathcal{F}_{CRS}^{\mathcal{D}}$ proceeds as follows, when parameterized by a distribution $\mathcal{D}$:

Common Reference String: When activated for the first time on input $(\mathsf{CRS}, \mathsf{sid})$, choose a value $d \xleftarrow{\$} \mathcal{D}$ and send back to the activating party. In each other activation return the value d to the activating party.

Figure 2.2: Ideal functionality for generating a Common Reference String, from (Canetti and Fischlin, 2001).

2.2.8 Authenticated Communication

We use a variant of Canetti's ideal functionality for authenticated communication, $\mathcal{F}_{AUTH}$, to abstract the notion of message authentication (Canetti, 2001). This is presented in Figure 2.3. Since we restrict our analysis to static corruption, we simplify this functionality to remove the adaptive corruption

interface.[4]

Functionality $\mathcal{F}_{AUTH}$

Sending: Upon receiving and input $(\mathsf{Send}, \mathsf{sid}, P_j, m)$ from P_i, if it is *not* the first instance of $(\mathsf{Send}, \cdot, \cdot, \cdot)$, the ideal functionality does nothing. Otherwise, it sends $(\mathsf{Sent}, \mathsf{sid}, P_i, P_j, m)$ to the adversary. If the adversary responds with (OK), then the ideal functionality sends $(\mathsf{Sent}, \mathsf{sid}, P_i, P_j, m)$ and halts. If the adversary does not approve, the ideal functionality drops the message and halts.

Figure 2.3: The message authentication ideal functionality $\mathcal{F}_{AUTH}$ supporting static corruption, adapted from (Canetti, 2001).

2.2.9 Commitment Schemes

We now define a commitment scheme.

Definition 12 (Commitment Scheme) *A commitment scheme* Π_{com} *is a tuple of algorithms* $(\mathsf{Setup}, \mathsf{Commit}, \mathsf{Open})$*, defined as follows:*

- $\mathsf{Setup}(1^\lambda)$ *outputs some parameters for the scheme,* $\mathsf{PP}_{\mathsf{com}}$.
- $\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, m; r)$ *takes in a message m and some randomness r and outputs a commitment c*
- $\mathsf{Open}(\mathsf{PP}_{\mathsf{com}}, c; r)$ *takes in a commitment c and some randomness r and outputs either a message m or the error symbol* $\perp$

[4]Note that this ideal functionality only handles a single message transfer, but to achieve multiple messages, we rely on universal composition and use multiple instances of the functionality.

We say that a commitment scheme is binding if

$$\Pr\left[\, m' \neq m \;\middle|\; \begin{array}{c} \mathsf{PP}_{\mathsf{com}} \leftarrow \mathsf{Setup}(1^\lambda), \\ m, r, aux \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{PP}_{\mathsf{com}}), \\ c \leftarrow \mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, m; r), \\ r' \leftarrow \mathcal{A}_2(1^\lambda, \mathsf{PP}_{\mathsf{com}}, c, m, r, aux), \\ m' \leftarrow \mathsf{Open}(\mathsf{PP}_{\mathsf{com}}, c; r') \end{array} \right] \leq \mathsf{negl}(1^\lambda).$$

We say that a commitment scheme is binding if

$$\Pr\left[b \leftarrow \mathcal{A}_2(1^\lambda, pk, c^*, aux) \;\middle|\; \begin{array}{c} \mathsf{PP}_{\mathsf{com}} \leftarrow \mathsf{Setup}(1^\lambda), \\ m_0, m_1, aux \leftarrow \mathcal{A}_1(1^\lambda, \mathsf{PP}_{\mathsf{com}}), \\ b \xleftarrow{\$} \{0,1\}, r \leftarrow \{0,1\}^\lambda, \\ c^* \leftarrow \mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, m_b; r) \end{array} \right] < \mathsf{negl}(1^\lambda).$$

Finally, we say that a commitment scheme is secure if and only if it is both hiding and binding.

2.2.10 Cryptographic Hash Functions

We will now define collision-resistant hash functions.

Definition 13 (Collision-resistant Hash Function) *A hash function* $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\ell$ *compresses arbitrary length input into some fixed length digest. For a hash*

function H *to be collision-resistant, we require that*

$$\Pr\left[\mathsf{H}(m_0) = \mathsf{H}(m_1) \,\middle|\, m_0, m_1 \leftarrow \mathcal{A}(1^\lambda)\right] \leq \mathsf{negl}(1^\lambda).$$

While not strictly required by collision-resistance, we require additionally that H *has pre-image resistance,* i.e.

$$\Pr\left[\mathsf{H}(m) = h \,\middle|\, \forall h \in \{0,1\}^\ell, m \leftarrow \mathcal{A}(1^\lambda, h)\right] \leq \mathsf{negl}(1^\lambda).$$

2.2.11 Garbled Circuits

Definition 14 (Garbling Scheme) *A garbling scheme for circuits is a tuple of PPT algorithms* $\mathsf{GC} := (\mathsf{Gen}, \mathsf{Garble}, \mathsf{Evaluate})$ *such that*

- $\mathsf{Gen}(1^\lambda, \mathsf{input}; r)$*:* Garble *takes the security parameter* λ *and length of input for the circuit as input and outputs a set of input labels* $\{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}}$*. When we explicitly need to specify the randomness in* Gen *we will include it as* r *as here.*

- $\mathsf{Garble}(C, \{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}})$*:* Garble *takes as input a circuit* $C : \{0,1\}^{\mathsf{input}} \rightarrow \{0,1\}^{\mathsf{output}}$ *and a set of input labels* $\{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}}$ *and outputs the garbled circuit* $\widetilde{C}$*.*

- $\mathsf{Evaluate}(\widetilde{C}, \mathsf{label}^x)$*:* Evaluate *takes as input the garbled circuit* $\widetilde{C}$*, input labels* label^x *corresponding to the input* $x \in \{0,1\}^{\mathsf{input}}$ *and outputs* $y \in \{0,1\}^{\mathsf{output}}$*.*

This garbling scheme satisfies the following properties:

1. ***Correctness:*** *For any circuit* C *and input* $x \in \{0,1\}^{\mathsf{input}}$*,*

$$\Pr[C(x) = \mathsf{Evaluate}(\widetilde{C}, \mathsf{label}^x)] = 1$$

where

$$(\{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{input})$$

and

$$\widetilde{C} \leftarrow \mathsf{Garble}(C, \{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}})$$

.

2. ***Selective Security:*** *There exists a PPT simulator* $\mathsf{Sim}_{\mathsf{GC}}$ *such that, for any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $\mu(.)$ *such that,*

$$|\Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{Sim}_{\mathsf{GC}}}(1^\lambda, 0) = 1] - \Pr[\mathsf{Exp}_{\mathcal{A},\mathsf{Sim}_{\mathsf{GC}}}(1^\lambda, 1) = 1]| \leq \mu(\lambda)$$

where the experiment $\mathsf{Exp}_{\mathcal{A},\mathsf{Sim}_{\mathsf{GC}}}(1^\lambda, b)$ *is defined as follows:*

   (a) *The adversary* $\mathcal{A}$ *specifies the circuit* C *and an input* $x \in \{0,1\}^{\mathsf{input}}$ *and gets* $\widetilde{C}$ *and* label^x*, which are computed as follows:*

   - ***If*** $b = 0$*:*
     - $(\{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}}) \leftarrow \mathsf{Gen}(1^\lambda, \mathsf{input})$
     - $\widetilde{C} \leftarrow \mathsf{Garble}(C, \{\mathsf{label}^{w,b}\}_{w\in\mathsf{input},b\in\{0,1\}})$
   - ***If*** $b = 1$*:*
     - $(\widetilde{C}, \mathsf{label}^x) \leftarrow \mathsf{Sim}_{\mathsf{GC}}(1^\lambda, C, x)$

   (b) *The adversary outputs a bit* b'*, which is the output of the experiment.*

2.2.12 Programmable Global Random Oracle Model

In Chapter 5, we will prove the security of one construction in the random oracle model. More specifically, we will leverage the global random oracle

model, introduced in (Camenisch et al., 2018). The ideal functionality $\mathcal{G}_{\mathsf{pRO}}$ is illustrated in Figure 2.4.

Functionality $\mathcal{G}_{\mathsf{pRO}}$.

Initiate with an empty list $\mathsf{List}_{\mathcal{H}}$

Query: Upon receiving message $(\mathsf{HashQuery}, m)$ from party P, the ideal functionality proceeds as follows. Find h such that $(m, h) \in \mathsf{List}_{\mathcal{H}}$. If no such h exists, let $h \xleftarrow{\$} \{0,1\}^{\ell}$ and store (m, h) in $\mathsf{List}_{\mathcal{H}}$. Return $(\mathsf{HashConfirm}, h)$ to P.

Program: Upon receiving message $(\mathsf{ProgramRO}, m, h)$ from adversary $\mathcal{A}$, the ideal functionality proceeds as follows. If $\exists h' \in \{0,1\}^{\ell}$ such that $(m, h') \in \mathsf{List}_{\mathcal{H}}$ and $h \neq h'$, then abort. Otherwise, add (m, h) to $\mathsf{List}_{\mathcal{H}}$ and output $(\mathsf{ProgramConfirm})$ to $\mathcal{A}$

Figure 2.4: Ideal functionality for the global programmable random oracle, from (Camenisch et al., 2018).

2.3 Trusted Hardware

Trusted hardware is a non-cryptographic primative that has proliferated in recent years. Generally, trusted hardware are realized as co-processors, hardened against attack, that can keep secrets hidden even in the presence of a malicious host machine an operator. The first such co-processors were Trusted Platform Modules (TPM) (*Information technology âĂŤ Trusted platform module library* 2015), capable of performing a limiting number of cryptographic operations like random number generation, attestation, and public key encryption. More recent generations of trusted hardware include trusted execution environments (TEE), including Intel SGX (*Intel Software Guard Extensions (Intel*

SGX) 2018) and ARM Trustzone (*ARM Trustzone*). Unlike TPMs, TEEs allow for arbitrary private computation — all without leaking any information about the computation to the host system or operating system. The execution environment is often referred to as an *enclave*.

Intel SGX is an extension of the x86 instruction set. When the processor wants to enter this privileged execution mode, it executes a special instruction that loads external encrypted state and decrypts it with a device-specific hard-coded key. The program code is then loaded into the enclave. All computation is then executed on the processor and cannot be interrupted or observed. The processor then executes the exit instruction, paging out and encrypting memory.

Attacks on Intel SGX. Since the introduction and adoption of TEEs, a new class of attacks has started to call into question the efficacy of these hardware-backed protections. The most notable of these attacks, named Foreshadow (Van Bulck et al., 2018), leverages the speculative execution nature of modern processors (similar to the Meltdown (Lipp et al., 2018) and Spectre (Kocher et al., 2019) attacks) to extract long-term keys used by SGX to encrypt memory. There have also been many side-channel attacks used to attack SGX, including (Weichbrodt et al., 2016; Moghimi, Irazoqui, and Eisenbarth, 2017; Wang et al., 2017; Lee et al., 2017; Biondo et al., 2018; Dall et al., 2018). These side channels allow the host to recover information about the control flow of the computation or the actual value of variables used in the computation, which ought to be impossible. While there has been some work to harden TEEs against these attacks, *e.g.* (Shih et al., 2017; Seo et al., 2017; Chandra et al.,

2017; Chen et al., 2018), it is not clear that truly secure TEEs will ever exist. As such, assuming the existence of secure TEEs can be controversial. In this work, we make somewhat liberal use of TEEs as a primitive because they exist in practice, and continue to improve as new attacks are discovered. We note that if cryptographic obfuscation ever exists, our results naturally extend to the obfuscation setting.

2.3.1 TEE Theoretical Modeling

In Chapter 4 we give a formal proof using the theoretical modeling of trusted execution environments provided by Pass *et. al* in (Pass, Shi, and Tramèr, 2017). This work focuses specifically on the guarantees provided by Intel SGX. We provide this functionality in Figure 2.5. Notation from (Pass, Shi, and Tramèr, 2017) used is described below:

- $\mathcal{P}$ is the identifier of party.
- reg refers to the registry of machines with the trusted hardware.
- prog is the program.
- inp, outp refers to the input and output.
- mem is the program's memory tape.
- Π_{Sign} is a signature scheme.

$\mathcal{G}_{\mathsf{att}}[\Pi_{\mathsf{Sign}}, \mathsf{reg}]$

//initialization

- Upon initialization, set $(params, msk) \leftarrow \Pi_{\mathsf{Sign}}.\mathsf{Gen}(\lambda)$, $T = \phi$

//public query interface

- Upon receiving `getpk()` from some $\mathcal{P}$: send *params* to $\mathcal{P}$

Enclave operations

//local interface — install an enclave

- Upon receiving `install`(idx, prog) from some $\mathcal{P} : \in \mathsf{reg}$
  - if $\mathcal{P}$ is honest, assert *idx=sid*
  - generate nonce $eid \in \{0,1\}^\lambda$, store $T[eid, \mathcal{P}] := (idx, \mathsf{prog}, \mathbf{0})$

//local interface — resume an enclave

- Upon receiving `resume`(eid, inp) from some $\mathcal{P} : \in \mathsf{reg}$
  - let $(idx, \mathsf{prog}, \mathsf{mem}) := T[eid, \mathcal{P}]$, abort if not found.
  - let $(\mathsf{outp}, \mathsf{mem}) := \mathsf{prog}(\mathsf{inp}, \mathsf{mem})$, update $T[eid, \mathcal{P}] := (idx, \mathsf{prog}, \mathsf{mem})$
  - let $\sigma \leftarrow \Pi_{\mathsf{Sign}}.\mathsf{Sign}_{msk}(idx, eid, \mathsf{prog}, \mathsf{outp})$ and send (outp, σ) to $\mathcal{P}$.

Figure 2.5: A global functionality modeling an SGX-like secure processor. The enclave program prog may be probabilistic and this is important for privacy-preserving applications. Enclave program outputs are included in an anonymous attestation σ. For honest parties, the functionality verifies that installed enclaves are parameterized by the session id *sid* of the current protocol instance.

Model and Preliminaries References

Canetti, Ran (2001). "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

Certificate Transparency (2018). Available at https://www.certificate-transparency.org.

Bitcoin Wiki: Script (2018). Available at https://en.bitcoin.it/wiki/Script.

Nakamoto, S. (2008). "Bitcoin: A peer-to-peer electronic cash system, 2008". In: URL: http://www.bitcoin.org/bitcoin.pdf.

The Ethereum Project (2018). https://www.ethereum.org/.

Hyperledger (2017). *Hyperledger Architecture, Volume 1*. Available at https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

Intel Corporation (2018). *Hyperledger Sawtooth*. Available at http://hyperledger.org/projects/sawtooth.

Goldreich, Oded, Shafi Goldwasser, and Silvio Micali (1984). "How to Construct Random Functions (Extended Abstract)". In: *25th FOCS*. IEEE Computer Society Press, pp. 464–479. DOI: 10.1109/SFCS.1984.715949.

Rogaway, Phillip (2002). "Authenticated Encryption with Associated Data". In: *CCS '02*. ACM Press.

Goldwasser, Shafi and Silvio Micali (1984). "Probabilistic Encryption". In: *Journal of Computer and System Sciences* 28.2, pp. 270–299.

Bellare, Mihir, Anand Desai, Eric Jokipii, and Phillip Rogaway (1997). "A Concrete Security Treatment of Symmetric Encryption". In: *38th FOCS*. IEEE Computer Society Press, pp. 394–403. DOI: 10.1109/SFCS.1997.646128.

Bellare, Mihir and Phillip Rogaway (2000). "Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient

Cryptography". In: *ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. LNCS. Springer, Heidelberg, pp. 317–330. DOI: 10.1007/3-540-44448-3_24.

Rogaway, Phillip, Mihir Bellare, John Black, and Ted Krovetz (2001). "OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption". In: *ACM CCS 2001*. Ed. by Michael K. Reiter and Pierangela Samarati. ACM Press, pp. 196–205. DOI: 10.1145/501983.502011.

Bellare, Mihir, Dennis Hofheinz, and Scott Yilek (2009). "Possibility and Impossibility Results for Encryption and Commitment Secure under Selective Opening". In: *EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. LNCS. Springer, Heidelberg, pp. 1–35. DOI: 10.1007/978-3-642-01001-9_1.

Peikert, Chris and Brent Waters (2008). "Lossy trapdoor functions and their applications". In: *40th ACM STOC*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, pp. 187–196. DOI: 10.1145/1374376.1374406.

Bellare, Mihir, Eike Kiltz, Chris Peikert, and Brent Waters (2012). "Identity-Based (Lossy) Trapdoor Functions and Applications". In: *EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. LNCS. Springer, Heidelberg, pp. 228–245. DOI: 10.1007/978-3-642-29011-4_15.

Hemenway, Brett, Benoît Libert, Rafail Ostrovsky, and Damien Vergnaud (2011). "Lossy Encryption: Constructions from General Assumptions and Efficient Selective Opening Chosen Ciphertext Security". In: *ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. LNCS. Springer, Heidelberg, pp. 70–88. DOI: 10.1007/978-3-642-25385-0_4.

Pedersen, Torben P. (1992). "Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing". In: *CRYPTO'91*. Ed. by Joan Feigenbaum. Vol. 576. LNCS. Springer, Heidelberg, pp. 129–140. DOI: 10.1007/3-540-46766-1_9.

Yao, Andrew Chi-Chih (1986). "How to Generate and Exchange Secrets (Extended Abstract)". In: *27th FOCS*. IEEE Computer Society Press, pp. 162–167. DOI: 10.1109/SFCS.1986.25.

Goldreich, Oded, Silvio Micali, and Avi Wigderson (1987). "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, pp. 218–229. DOI: 10.1145/28395.28420.

Chaum, David, Claude Crépeau, and Ivan Damgård (1988). "Multiparty Unconditionally Secure Protocols (Abstract) (Informal Contribution)". In: *CRYPTO'87*. Ed. by Carl Pomerance. Vol. 293. LNCS. Springer, Heidelberg, p. 462. DOI: 10.1007/3-540-48184-2_43.

Ben-Or, Michael, Shafi Goldwasser, and Avi Wigderson (1988). "Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract)". In: *20th ACM STOC*. ACM Press, pp. 1–10. DOI: 10.1145/62212.62213.

Gentry, Craig, Allison B. Lewko, and Brent Waters (2014). "Witness Encryption from Instance Independent Assumptions". In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, pp. 426–443. DOI: 10.1007/978-3-662-44371-2_24.

Boyle, Elette, Kai-Min Chung, and Rafael Pass (2014). "On Extractability Obfuscation". In: *TCC 2014*. Ed. by Yehuda Lindell. Vol. 8349. LNCS. Springer, Heidelberg, pp. 52–73. DOI: 10.1007/978-3-642-54242-8_3.

Sahai, Amit (1999). "Non-Malleable Non-Interactive Zero Knowledge and Adaptive Chosen-Ciphertext Security". In: *40th FOCS*. IEEE Computer Society Press, pp. 543–553. DOI: 10.1109/SFFCS.1999.814628.

De Santis, Alfredo, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai (2001). "Robust Non-interactive Zero Knowledge". In: *CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. LNCS. Springer, Heidelberg, pp. 566–598. DOI: 10.1007/3-540-44647-8_33.

Oren, Yair (1987). "On the Cunning Power of Cheating Verifiers: Some Observations about Zero Knowledge Proofs (Extended Abstract)". In: *28th FOCS*. IEEE Computer Society Press, pp. 462–471. DOI: 10.1109/SFCS.1987.43.

Goldreich, Oded and Yair Oren (1994). "Definitions and Properties of Zero-Knowledge Proof Systems". In: *Journal of Cryptology* 7.1, pp. 1–32. DOI: 10.1007/BF00195207.

Goldreich, Oded and Hugo Krawczyk (1996). "On the Composition of Zero-Knowledge Proof Systems". In: *SIAM J. Comput.* 25.1, pp. 169–192. ISSN: 0097-5397. DOI: 10.1137/S0097539791220688. URL: https://doi.org/10.1137/S0097539791220688.

Canetti, Ran and Marc Fischlin (2001). "Universally Composable Commitments". In: *CRYPTO 2001*. Ed. by Joe Kilian. Vol. 2139. LNCS. Springer, Heidelberg, pp. 19–40. DOI: 10.1007/3-540-44647-8_2.

Camenisch, Jan, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven (2018). "The Wonderful World of Global Random Oracles". In: *EUROCRYPT 2018, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. LNCS. Springer, Heidelberg, pp. 280–312. DOI: 10.1007/978-3-319-78381-9_11.

Information technology âĂŤ Trusted platform module library (2015). Standard. Geneva, CH: International Organization for Standardization.

Intel Software Guard Extensions (Intel SGX) (2018). https://software.intel.com/en-us/sgx.

ARM Consortium. *ARM Trustzone*. Available at https://developer.arm.com/technologies/trustzone.

Van Bulck, Jo, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx (2018). "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, pp. 991–1008.

Lipp, Moritz, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg (2018). "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, pp. 973–990.

Kocher, Paul, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom (2019). "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, pp. 1–19. DOI: 10.1109/SP.2019.00002.

Weichbrodt, Nico, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza (2016). "AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves". In: *ESORICS 2016, Part I*. Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Vol. 9878. LNCS. Springer, Heidelberg, pp. 440–457. DOI: 10.1007/978-3-319-45744-4_22.

Moghimi, Ahmad, Gorka Irazoqui, and Thomas Eisenbarth (2017). "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. LNCS. Springer, Heidelberg, pp. 69–90. DOI: 10.1007/978-3-319-66787-4_4.

Wang, Wenhao, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter (2017). "Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX". In: *ACM CCS 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM Press, pp. 2421–2434. DOI: 10.1145/3133956.3134038.

Lee, Sangho, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado (2017). "Inferring Fine-grained Control Flow Inside SGX

Enclaves with Branch Shadowing". In: *USENIX Security 2017*. Ed. by Engin Kirda and Thomas Ristenpart. USENIX Association, pp. 557–574.

Biondo, Andrea, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi (2018). "The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, pp. 1213–1227.

Dall, Fergus, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom (2018). "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks". In: *IACR TCHES* 2018.2, pp. 171–191. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i2.171-191.

Shih, Ming-Wei, Sangho Lee, Taesoo Kim, and Marcus Peinado (2017). "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs". In: *NDSS 2017*. The Internet Society.

Seo, Jaebaek, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim (2017). "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs". In: *NDSS 2017*. The Internet Society.

Chandra, Swarup, Vishal Karande, Zhiqiang Lin, Latifur Khan, Murat Kantarcioglu, and Bhavani M. Thuraisingham (2017). "Securing Data Analytics on SGX with Randomization". In: *ESORICS 2017, Part I*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Vol. 10492. LNCS. Springer, Heidelberg, pp. 352–369. DOI: 10.1007/978-3-319-66402-6_21.

Chen, Guoxing, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin (2018). "Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races". In: *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, pp. 178–194. DOI: 10.1109/SP.2018.00024.

Pass, Rafael, Elaine Shi, and Florian Tramèr (2017). "Formal Abstractions for Attested Execution Secure Processors". In: *EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. LNCS. Springer, Heidelberg, pp. 260–289. DOI: 10.1007/978-3-319-56620-7_10.

Chapter 3

Enclave Ledger Interaction

3.1 Introduction

In recent years a new class of distributed system has evolved. Loosely categorized as *decentralized ledgers*, these systems construct a virtual "bulletin board" to which nodes may publish data. Many protocols, including cryptocurrencies such as Bitcoin (Nakamoto, 2008), construct such a ledger to record financial transactions. More recent systems target other specific applications, such as identity management (Handshake, 2018; *Namecoin* 2016), or the execution of general, user-defined programs, called "smart contracts" (*The Ethereum Project* 2018). Some companies have also deployed centralized public ledgers for specific applications; for example, Google's Certificate Transparency (*Certificate Transparency* 2018) provides a highly-available centralized ledger for recording issued TLS certificates.

While the long-term success of specific systems is uncertain, two facts seem clear: (1) centralized and decentralized ledger systems are already in widespread deployment, and this deployment is likely to continue. Moreover

(2) the decentralized nature of these systems makes them potentially long-lived and resilient to certain classes of network-based attack. This provides a motivation to identify new ways that these technologies can be used to enhance the security of distributed systems.

In this chapter we focus on one such application: using ledgers to enhance the security of Trusted Execution Environments (TEE). In the context of this work, we use TEE to refer to any limited, secure computing environment that is dependent on a (possibly malicious) host computer for correct operation. Examples of such environments include Hardware Security Modules, smart cards (Poulsen, 2001), and the "secure element" co-processors present in many mobile devices (Apple Computer, 2016), as well as virtualized TEE platforms such as Intel's Software Guard Extensions (SGX), ARM TrustZone, and AMD SEV (*Intel Software Guard Extensions (Intel SGX)* 2018; ARM Consortium, 2017; Advanced Microchip Devices, 2018). While many contemporary examples rely on hardware, it is conceivable that future trusted environments may be implemented using pure software virtualization, general-purpose hardware obfuscation (Garg et al., 2013; Döttling et al., 2011; Nayak et al., 2017), or even cryptographic program obfuscation (Lewi et al., 2016).

While TEEs have many applications in computing, they (like all secure co-processors) have fundamental limitations. A trusted environment operating perfectly depends on the host computer for essential functionality, creating an opportunity for a malicious host to manipulate the TEE and its view of the world. For example, an attacker may:

1. Tamper with network communications, censoring certain inputs or outputs and preventing the TEE from communicating with the outside world.

2. Tamper with stored non-volatile data, *e.g.* replaying old stored state to the TEE in order to *reset* the state of a multi-step computation.

We stress that these attacks may have a catastrophic impact *even if the TEE itself operates exactly as designed*. For example, many interactive cryptographic protocols are vulnerable to "reset attacks," in which an attacker rewinds or resets the state of the computation (Bellare et al., 2001; *TPM Reset Attack* 2018; Giller, 2015). State reset attacks are not merely a problem for cryptographic protocols; they are catastrophic for many typical applications such as limited-attempt password checking (Skorobogatov, 2016).

When implemented in hardware, TEE systems can mitigate reset attacks by deploying a limited amount of tamper-resistant non-volatile storage (Parno et al., 2011).[1] However, such countermeasures increase the cost of producing the hardware and are simply not possible in software-only environments. Moreover, these countermeasures are unavailable to environments where a single state transition machine is run in a *distributed* fashion, with the transition function executed across different machines. In these environments, which include private smart contract systems (Hyperledger, 2017) and "serverless" cloud step-function environments (Amazon, 2018; Google Inc., 2018), state

[1] The literature affords many examples of attackers bypassing such mechanisms (Skorobogatov and Anderson, 2003; Kauer, 2007; Skorobogatov, 2016) using relatively inexpensive physical and electronic attacks.

protections cannot be enforced locally by hardware. Similarly, hardware countermeasures cannot solve the problem of enforcing a secure channel to a public data network.

A hypothetical solution to these problems is to delegate statekeeping and network connectivity to a remote, trusted server or small cluster of peers, as discussed in (Matetic et al., 2017). These could keep state on behalf of the enclave and would act as conduit to the public network. However, this approach simply shifts the root of trust to a different physical location, failing to solve our problem because this new system is vulnerable to the same attacks. Moreover, provisioning and maintaining the availability of an appropriate server can be a challenge for many applications, including IoT deployments that frequently outlive the manufacturer.

Combining TEEs with Ledgers In this work we consider an alternative approach to ensuring the statefulness and connectivity of trusted computing devices. Unlike the strawman proposals above, our approach does not require the TEE to include secure internal non-volatile storage, nor does it require a protocol-aware external server to keep state. Instead, we propose a model in which parties have access to an append-only public ledger, or *bulletin board* with certain properties. Namely, upon publishing a string S on the ledger, a party receives a copy of the resulting ledger – or a portion of it – as well as a proof (*e.g.* a signature) to establish that the publication occurred. Any party, including a trusted device, can verify this proof to confirm that the received ledger data is authentic. The main security requirement we require from the ledger is that its contents cannot be modified or erased and proofs of

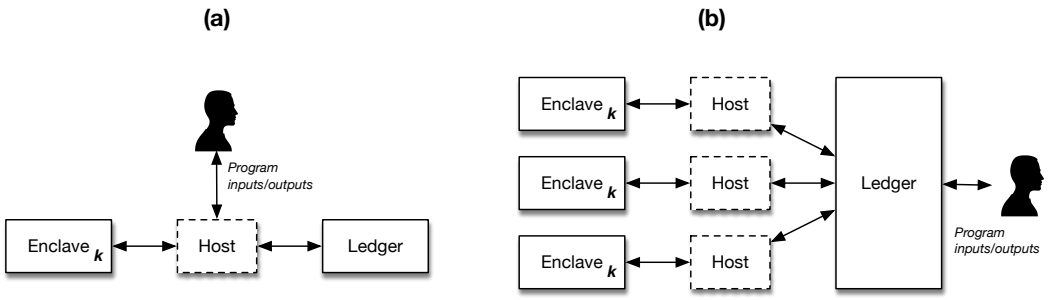

Figure 3.1: Two example ELI deployments. In the basic scenario (a) a single TEE (with a hard-coded secret key K) interacts with a ledger functionality via a (possibly adversarial) host application. Program inputs are provided by a user via the host machine. In scenario (b) multiple copies of the same enclave running on different host machines interact with the ledger (*e.g.*, as in a private smart contract system), which allows them to synchronize a multi-step execution across many different machines without the need for direct communication. Program inputs and outputs may be provided via the ledger.

publication cannot be (efficiently) forged.

Our contributions In this work we propose a new general protocol, which we refer to as an *Enclave-Ledger Interaction* (ELI). This proposal divides any multi-step interactive computation into a protocol run between three parties: a stateless client-side TEE, which we refer to as an *enclave* (for the rest of this work, we use these two terms interchangeably) that contains a secret key; a *ledger* that logs posted strings and returns a proof of publication; and a (possibly adversarial) *host application* that facilitates all communications between the two preceding parties. Users may provide inputs to the computation via the host, or through the ledger itself. We illustrate our model in Figure 3.1.

We assume that the enclave is a trustworthy computing environment, such as a tamper-resistant hardware co-processor, SGX enclave, or a cryptographically obfuscated circuit (Nayak et al., 2017; Lewi et al., 2016). Most notably, the enclave need not store persistent state or possess a secure random number

generator; we only require that the enclave possesses a single secret key K that is not known to any other party. We similarly require that the host application can publish strings to the ledger; access the ledger contents; and receive proofs of publication.

As a first contribution, we show how this paradigm can facilitate secure state management for randomized *multi-step* computations run by the enclave, even when the enclave has no persistent non-volatile storage or access to trustworthy randomness.

Building such a protocol is non-trivial, as it requires simultaneously that the computation cannot be rewound or forked, even by an adversarial host application that controls all state and interaction with the ledger.

As a second contribution, we show that the combination of enclave plus ledger can achieve properties that may not be achievable even when the enclave uses stateful trusted hardware. In particular, we show how the enclave-ledger interaction allows us to condition program execution on the *publication* of particular messages to the ledger, or the receipt of messages from third parties. For example, an application can require that devlopers be alerted on the ledger that user activity is anomolous, perhaps even dangerous, before it continues execution.

As a third contribution, we describe several practical applications that leverage this paradigm. These include private smart contracts, limited-attempt password checking (which is known to be difficult to enforce without persistent state (Skorobogatov, 2016)), enforced file access logging, and new forms of encryption that ensure all parties receive the plaintext, or that none do.

As a practical matter, we demonstrate that on appropriate ledger systems that support payments, execution can be conditioned on other actions, such as *monetary payments* made to the ledger. In malicious hands, this raises the specter of *autonomous ransomware* that operates verifiably and without any need for a command-and-control or secret distribution center.

Previous and concurrent work In previous work, Memoir (Parno et al., 2011) leverages hashchains, NVRAM, and monotonic counters to efficiently prevent state rollback in the presence of a malicious host. While Memoir's protocol has many similarities to our ELI protocols, the system design is quite different. In Memoir, each TEE device uses its internal NVRAM to checkpoint state, while our systems rely on a public ledger and communication through an untrustworthy host. A second proposal, ROTE (Matetic et al., 2017), uses a consensus between a cluster of distributed enclaves to in order to address the rollback problem. Neither Memoir nor ROTE deals with the problem of conditioning execution on data publication, which is a second contribution of our work. Several early works have focused on the problem of preventing reset attacks via *de-randomization*, *i.e.*, by deriving (pseudo)random coins from the computation input (Canetti et al., 2000). Unfortunately this approach does not generalize to multi-step calculations where the adversary can adaptively select the input prior to each step.

Two concurrent research efforts have considered the use of ledgers to achieve secure computation. In late 2017, Goyal and Goyal proposed the use of blockchains for implementing *one time programs* (Goyal and Goyal, 2017) using cryptographic obfuscation techniques. While our work has a similar

focus, we aim for a broader class of functionalities and a more practical set of applications. Also in 2017, the authors of the present work, along with others, proposed to use ledgers to obtain fairness for MPC protocols, an application that is discussed in later sections of this work (Choudhuri et al., 2017). Bowman et al. of Intel Corporation (Bowman et al., 2018) independently proposed "Private Data Objects" for smart contract systems that use ideas related to this work, and have begun to implement them in production smart contract systems that support private computation. We believe Bowman's effort strongly motivates the formal analysis we include in this work. There have also been a number of attempts to combine trusted execution environments and public ledgers, but aimed at slightly different goals (Kosba et al., 2016; Juels, Kosba, and Shi, 2016; Zhang et al., 2016). Finally, the Ekiden system (Cheng et al., 2018), proposed in April 2018, builds on the ideas proposed in this work and (Choudhuri et al., 2017) to achieve goals similar to those of Intel's Private Data Objects.

3.1.1 Intuition

We now briefly present the intuition behind our construction. Our goal is to securely execute a *multi-step* interactive, probabilistic program P,

which we will define as having the following interface:

$$P(\mathsf{I}_i, \mathsf{S}_i; \bar{r}_i) \rightarrow (\mathsf{O}_i, \mathsf{S}_{i+1})$$

At each step of the program execution, the program takes a user input I_i, an (optional) state S_i from the previous execution step, along with some

random coins $\bar{r}_i$. It produces an output O_i as well as an updated state S_{i+1} for subsequent steps.

(Looking forward, we will add *public* ledger inputs and outputs to this interface as well, but we now omit these for purposes of exposition.) For this initial exposition, we will assume a simple ledger that, subsequent to each publication, returns the full ledger contents L along with a proof of publication π_{publish}.[2] We also require a stateless *enclave* with no native random number generator, that stores a single, hardcoded, secret key K.

Figure 3.1 illustrates the way the user, host, ledger and enclave can interact. We now discuss several candidate approaches, beginning with obviously insecure ideas, and building on them to describe a first version of our main construction.

Attempt #1: Encrypt program state. An obvious first step is for the enclave to simply encrypt each output state using its internal secret key, and to send the resulting ciphertext to the host for persistent storage. Assuming that we use a proper authenticated encryption scheme (and pad appropriately), this approach should guard both the confidentiality and authenticity of state values even when they are held by a malicious host.[3]

It is easy to see that while this prevents tampering with the contents of stored state, it does not prevent a malicious host from *replaying* old state ciphertexts to the enclave along with new inputs. In practice, such an attacker can rewind and fork execution of the program.

[2] In later sections we will discuss improvements that make this Ledger response *succinct*.

[3] For the moment we will ignore the challenge of preventing re-use of nonces in the encryption scheme; these issues will need to be addressed in our main construction, however.

Attempt #2: Use the ledger to store state. A superficially appealing idea is to use the ledger itself to store an encrypted copy of the program state. As we will show, this does not mitigate rewinding attacks.

For example, consider the following strawman protocol: after the enclave executes the program P on some input, the enclave sends the resulting encrypted state to the ledger (via the host). The enclave can then condition future execution of P on receiving valid ledger contents L, as well as a proof of publication π_{publish}, and extracting the encrypted state from L.

Unfortunately this does nothing to solve the problem of adversarial replays. Because the enclave has no trusted source of time and relies on the host to communicate with the ledger, a malicious host can simply replay old versions of L (including the associated proofs-of-publication) to the enclave, while specifying different program inputs. As before, this allows the host to fork the execution of the program.

Attempt #3: Bind program inputs on the ledger. To address the replay problem, we require a different approach. As in our first attempt, we will have the enclave send encrypted state to the host (and not the ledger) for persistent storage. As a further modification, we will add to this encrypted state an iteration counter i which identifies the next step of the program to be executed.

To execute the i^{th} invocation of the program, the host first commits its next program input I_i to the ledger. This can be done in plaintext, although for privacy we will use a secure commitment scheme. It labels the resulting commitment C with a unique identifier CID that identifies the enclave, and sends the pair (C, CID) to the ledger.

Following publication, the host can obtain a copy of the full ledger L as well as the proof of publication π_{publish}. It sends *all* of the above values (including the commitment randomness R) to the enclave, along with the most recent value of the encrypted state (or ε if this is the first step of the program).

The enclave decrypts the encrypted state internally to obtain the program state and counter (S, i).[4] It verifies the following conditions:

1. π_{publish} is a valid proof of publication for L.
2. The ledger L contains exactly i tuples $(\cdot, \mathsf{CID})$.
3. The most recent tuple embeds (C, CID).
4. C is a valid commitment to the input I using randomness R.

If all conditions are met, the enclave can now execute the program on state and input (S, I). Following execution, it encrypts the new output state and updated counter $(\mathsf{S}_{i+1}, i+1)$ and sends the resulting ciphertext to the host for storage.

Remark. Like our previous attempts, the protocol described above does *not* prevent the host from replaying old versions of L (along with the corresponding encrypted state). Indeed, such replays will still cause the enclave to execute P and produce an output. Rather, our purpose is to prevent the host from replaying old state with *different inputs*. By forcing the host to commit to its input on the ledger before L is obtained, we prevent a malicious host from changing its

[4] If no encrypted state is provided, then i is implicitly set to 0 and $\mathsf{S} = \varepsilon$.

program input during a replay, ensuring that the host gains no new information from such attacks. However, there remains a single vulnerability in the above construction that we must still address.

Attempt #4: Deriving randomness. While the protocol above prevents the attacker from changing the inputs provided to the program, there still remains a vector by which the malicious host could fork program execution. Specifically, even if the program input is fixed for a given execution step, the program execution may fork if the *random coins* provided to P change between replays. This might prove catastrophic for certain programs.[5]

To solve this problem, we make one final change to the construction of the enclave code. Specifically, we require that at each invocation of P, the enclave will derive the random coins used by the program in a deterministic manner from the inputs, using a pseudorandom function (similar to the classical approach of Canetti *et al.* (Canetti et al., 2000)).

This approach fixes the random coins used at each computation step and effectively binds them to the ledger and the host's chosen input.

Limitations of our pedagogical construction. The construction above is intended to provide an intuition, but is not the final protocol we describe in this work.

An astute reader will note that this pedagogical example has many limitations, which must be addressed in order to derive a practical ELI protocol. We discuss several extensions below.

Extension #1: Reducing ledger bandwidth. The pedagogical protocol above

[5]For example, many interactive identification and oblivious RAM protocols become insecure if the program can be rewound and executed different randomness.

requires the host and enclave to parse *the entire ledger* L on each execution step. This is quite impractical, especially for public ledgers that may contain millions of transactions.

A key contribution of this work is to show that the enclave need not receive the entire ledger contents, provided that the ledger can be given only modest additional capabilities: namely (1) the ability to organize posted data into sequences (or chains), where each posted string contains a unique pointer to the preceding post, and (2) the ability for the ledger to calculate a collision-resistant hash chain over these sequences. As we discussed in Section 2.1 and Section 2.1.2.1, these capabilities are already present in many candidate ledger systems such as public (and private) blockchain networks.

Extension #2: Adding public input and output. A key goal of our protocol is to allow P to condition its execution on inputs and outputs drawn from (resp. sent to) the ledger. This can be achieved due to the fact that the enclave receives an authenticated copy of L. Thus the enclave (and P) can be designed to condition its operation on *e.g.*, messages or public payment data found on the ledger.

To enforce public output, we modify the interface of P to produce a "public output string" as part of its output to the host, and we record this string with the program's encrypted state. By structuring the enclave code (or P) appropriately, the program can *require* the host to post this string to the ledger as a condition of further program execution. Of course, this is not an absolute guarantee that the host will publish the output string. That is, the enclave cannot force the host to post such messages. Rather, we achieve a best-possible

guarantee in this setting: the enclave can simply disallow *further* execution if the host does not comply with the protocol.

Extension #3: Specifying the program. In the pedagogical presentation above, the program P is assumed to be fixed within the enclave. As a final extension, we note that the enclave can be configured to provide an environment for running arbitrary programs P, which can be provided as a separate input at each call. Achieving this involves recording (a hash) of P within the encrypted state, although the actual construction requires some additional checks to allow for a security proof. We include this capability in our main construction.

Modeling the ledger Several recent works have also used ledgers (or bulletin boards) to provide various security properties (Goyal and Goyal, 2017; Choudhuri et al., 2017). In these works, the ledger is treated as possessing an *unforgeable* proof of publication. The protocols in this work can operate under this assumption, however our construction is also motivated by real-world decentralized ledgers, many of which do not possess such a property. Instead, many "proof-of-work" blockchains provide a weaker security property, in that it is merely *expensive* to forge a proof that a message has been posted to the blockchain. This notion may provide sufficient security in many real-world applications, and we provide a detailed analysis of the costs in Section 2.1.2.1

3.1.2 Applications

To motivate our techniques, we describe a number of practical applications that can be implemented using the ELI paradigm, including both constructive

and potentially destructive techniques. Here we provide several example applications, and provide a more complete discussion in Section 3.5.

Synchronizing private smart contracts and step functions. Smart contract systems and cloud "step functions" (Ethereum White Paper, 2017; Hyperledger, 2017) each employ a distributed network of compute nodes that perform a multi-step interactive computation. To enable private computation, some production smart contract systems (Hyperledger, 2017) have recently proposed incorporating TEEs. Such distributed systems struggle to synchronize state as the computation migrates across nodes. Motivated by an independent effort of Bowman *et al.* (Bowman et al., 2018) we show that our ELI paradigm achieves the necessary guarantees for security in this setting.

Mandatory logging for local file access. Corporate and enterprise settings often require users to log access to sensitive files, usually on some online system. We propose to use the ELI protocol to *mandate* logging of each file access before the necessary keys for an encrypted file can be accessed by the user.

Limiting password guessing. Cryptographic access control systems often employ passwords to control access to encrypted filesystems (Apple Computer, 2016; Project, 2017) and cloud backup images (*e.g.,* Apple's iCloud Keychain (Krstić, 2016)). This creates a tension between the requirement to support easily memorable passwords (such as device PINs) while simultaneously preventing attackers from simply *guessing* users' relatively weak passwords (Bonneau, 2012; Ur et al., 2015).[6] Attempts to address this with

[6]This is made more challenging due to the fact that manufacturers have begun to design

tamper-resistant hardware (Apple Computer, 2016; Project, 2017; ARM Consortium, 2017; Krstić, 2016) lead to expensive systems that provide no security against rewind attacks. [7] We show that ELI can safely enforce *passcode guessing limits* using only inexpensive hardware without immutable state (Skorobogatov, 2016).

Autonomous ransomware. Modern ransomware, malware that encrypts a victim's files, is tightly integrated with cryptocurrencies such as Bitcoin, which act as both the ransom currency and a communication channel to the attacker (Sinegubko, 2016). Affected users must transmit an encrypted key package along with a ransom payment to the attacker, who responds with the necessary decryption keys.

The ELI paradigm could potentially enable the creation of ransomware that operates autonomously – from infection to decryption – with no need for remote parties to deliver secret keys. This ransomware employs local trusted hardware or obfuscation to store a decryption key for a user's data, and conditions decryption of a user's software on payments made on a public consensus network.

3.2 Definitions

Protocol Parties: A Enclave-Ledger Interaction is a protocol between three parties: the enclave $\mathcal{E}$, the ledger $\mathcal{L}^{\mathsf{Verify}}$, and a host application $\mathcal{H}$. We now

systems that do not include a trusted party – due to concerns that trusted escrow parties may be compelled to unlock devices (Apple Computer, 2017).

[7] See (Skorobogatov, 2016) for an example of how such systems can be defeated when state is recorded in standard NAND hardware, rather than full tamper-resistant hardware.

describe the operation of these components:

The ledger $\mathcal{L}^{\mathsf{Verify}}$. The ledger functionality provides a public append-only ledger for storing certain public data. Our main requirement is that the ledger is capable of producing a publicly-verifiable authentication tag π_{publish} over the entire ledger contents, or a portion of the ledger.

The enclave $\mathcal{E}$. The trusted enclave models a cryptographic obfuscation system or a trusted hardware co-processor configured with an internal secret key K. The enclave may contain the program P, or this program may be provided to it by the host application. Each time the enclave is invoked by the host application $\mathcal{H}$ on some input, it calculates and returns data to the host.

The host application $\mathcal{H}$. The host application is a (possibly adversarial) party that invokes both the enclave and the ledger functionalities. The host determines the inputs to each round of computation — perhaps after interacting with a user — and receives the outputs of the computation from the enclave.

3.2.1 The Program Model

Our goal in an ELI is to execute a multi-step interactive computation that runs on inputs that may be chosen *adaptively* by an adversary. Expanding on our initial description, we define this program $P : \mathcal{I} \times \mathcal{S} \times \mathcal{R} \to \mathcal{O} \times \mathcal{P} \times \mathcal{S}$ as having the following input/output interface:

$$P(\mathsf{I}_i, \mathsf{S}_i; \bar{r}_i) \to (\mathsf{O}_i, \mathsf{Pub}_i, \mathsf{S}_{i+1}).$$

When a user inputs I_i, the current program state S_i, and random coins $\bar{r}_i$, this algorithm produces a program output O_i, as well as an optional public

broadcast message Pub_i and new state S_{i+1}.

In our main construction, we will allow the host application to provide the program P that the enclave will run. This is useful in settings such as smart contract execution, where a given enclave may execute multiple distinct smart contract programs. As a result of this change, we will assume that P is passed as input to each invocation of the enclave.

Maximum program state size and runtime We assume in this work that the runtime of each P can be upper bounded by a polynomial function of the security parameter. We also require that for any program P used in our system there exists an efficiently-computable function $\mathsf{Max}(\cdot)$ such that $\mathsf{Max}(P)$ indicates the *maximum* length in bits of any output state S_i produced by P, and that $\mathsf{Max}(P)$ is polynomial in the security parameter.

One-Time vs. Multi-Use Programs In this work we consider two different classes of program. While all of our programs may involve multiple execution steps, *one-time programs* can be initiated only once by a given enclave. Once such a program has begun its first step of execution, it can never be restarted. By contrast, *multi-use programs* can be executed as many times as the user wishes, and different executions may be interwoven. However each execution of the program is independent of the others, receives different random coins and holds different state. In our model, an execution of a program will be uniquely identified by a session identifier, which we denote by CID. Thus, the main difference between a one-time and many-time program is whether the enclave will permit the re-initiation of a given program P under a different identifier CID.

$\mathsf{Setup}(1^\lambda) \to (K, \mathsf{PP}_{\mathsf{com}})$. This trusted setup algorithm is executed once to configure the enclave. On input a security parameter λ, it samples a long-term secret K which is stored securely within the enclave, and the (non-secret) parameters $\mathsf{PP}_{\mathsf{com}}$ which are provided to the enclave and the host.

$\mathsf{ExecuteApplication}(\mathsf{PP}_{\mathsf{com}}, P)$**.** This algorithm is run on the host. It proceeds in an infinite loop, invoking the ledger operations and enclave operations. In each iteration of the loop, the user selects a step input, commits to it and posts it to the ledger. It then sends that input and the Ledger's output into the enclave to actually execute the next step.

$\mathsf{ExecuteEnclave}_{K,\mathsf{PP}_{\mathsf{com}}}((P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \pi_{\mathsf{publish},i}, \mathsf{post}_i)) \to (\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_i)$. This algorithm is run by the enclave, which is configured with $K, \mathsf{PP}_{\mathsf{com}}$. At the i^{th} computation step it takes as input a program P, an encrypted previous state $\mathcal{S}_i$, a program input I_i, commitment randomness r_i, a ledger output post_i and a ledger authentication tag $\pi_{\mathsf{publish},i}$. The enclave invokes P and produces a public output O_i, as well as a new encrypted state $\mathcal{S}_{i+1}$ and a public output Pub_i.

Figure 3.2: Definition of an Enclave Ledger Interaction (ELI) scheme.

We note that it is possible to convert any multi-time program to a one-time program by having the enclave generate the value CID deterministically from its internal key K and the program P (*e.g.,* by calculating a pseudorandom function on these values), and then to enforce that each execution of the program P is associated with the generated CID. This enforcement algorithm can be instantiated as a "meta-program" P' that takes as input a second program P and is executed using our unmodified ELI protocol.

While our pedagogical example in the introduction discussed one-time programs, in the remainder of this work we will focus on multi-use programs, as these are generally sufficient for our proposed applications in Section 3.5.

3.3 Security Definitions for Enclave-Ledger Interaction

In this section we present formal definitions of security for an ELI scheme.

3.3.1 Ledger Unforgeability

As noted in Section 2.1.1, we require that it is difficult to construct a pair $(S, \pi_{\mathsf{publish}})$ such that $\mathsf{Ledger.Verify}(S, \pi_{\mathsf{publish}}) = 1$ except as the result of a call to $\mathsf{Ledger.Post}$. Intuitively we refer to this definition as SUF-AUTH. This definition is analogous to the SUF-CMA definition used for signatures. Indeed, in many practical instantiations of ledgers, these signatures will be one of the techniques used to construct the authentication tag.

3.3.2 Simulation security

Intuition. Our definition specifies two experiments: a **Real** experiment in which an adversarial host application runs the real ELI protocol with oracles that implement honest enclave and ledger functionalities respectively, and an **Ideal** experiment that models the correct and stateful execution of the underlying program P by a trusted party. In broad strokes, our security definition requires that for every p.p.t. adversary $\mathcal{H}$ that runs the **Real** experiment, there must exist an ideal adversary $\hat{\mathcal{H}}$ that runs the **Ideal** experiment such that the output of $\mathcal{H}$ is computationally indistinguishable from that of $\hat{\mathcal{H}}$.

Our first experiment, which we term the **Real** experiment, we define an interaction where an adversarial host user $\mathcal{H}$ interacts with an honest ledger

and honest enclave to execute the ELI protocol. These interfaces are provided to the host in the form of oracles that the host may call an unbounded number of times. In the **Ideal** experiment, we consider an adversarial *ideal* host $\hat{\mathcal{H}}$ that interacts with a trusted functionality. This functionality takes as input a program P and a program input provided by $\hat{\mathcal{H}}$, and runs the program using real random coins. The trusted functionality stores the resulting state internally, broadcasts public outputs to all parties, and returns user outputs to the user. This ideal model is intuitively what we wish to accomplish from a secure multi-step interactive computing system.

Our security definition requires that for every p.p.t. adversarial real-world hosts $\mathcal{H}$, there must exist a p.p.t. ideal-world host $\hat{\mathcal{H}}$ that does "as well" in the ideal experiment as the real host does in the real experiment. More formally, we define this last notion via an indistinguishability-based definition: we require that the output distributions of $\mathcal{H}$ and $\hat{\mathcal{H}}$ are computationally indistinguishable between the two experiments.

What if the ledger is forgeable? In some of our proposed instantiations, the ledger authentication tags are *economically* secure, rather than cryptographically secure. In this setting feasible for an attacker to forge an authentication tag, but doing so is extremely costly. This setting raises the possibility that a motivated attacker might be able to forge some limited number of tags. In this situation, it is reasonable to consider how security *degrades* in the face of these forgeries.

For example, it is easy to imagine some ELI protocol that operates securely when forgeries are infeasible, but that becomes catastrophically insecure when

the enclave is presented with a *single* forged authentication tag. For example, a contrived ELI protocol might be constructed to reveal a share of the secret K with each output, such that a single adversarial rewind allows the adversary to obtain K. This would clearly be catastrophic for the security of an ELI! While we cannot prevent an attacker from obtaining some advantage using forgeries, we wish to more clearly bound the attacker's capability in this setting.

Our approach to this problem is to extend both the **Real** and **Ideal** experiments with an additional capability. In the **Real** experiment, the adversary $\mathcal{H}$ is permitted to query a **Forgery** oracle that, on input a string S, provides a valid pair $(S, \pi_{\mathsf{publish}})$ but *does not* update the ledger. In the **Ideal** experiment we provide a **Fork** oracle that allows the ideal adversary $\hat{\mathcal{H}}$ to run a single step of the computation using an older state (of their choosing). In each experiment, the adversaries are restricted to querying these respective oracles a maximum of q_{forge} times, where q_{forge} is a parameter provided as input to the experiment. (Clearly $q_{\mathsf{forge}} = 0$ implies a definition with no forgeries.)

Intuitively, this extended definition models the ability of an attacker to rewind a single step of the computation (per forgery) but without giving this attacker the ability to catastrophically break the security of the system, or even to run additional *legitimate* execution steps from this execution step. We stress that this definition represents a form of *best possible* security in our model, when forgeries are allowed.

3.3.3 Formal Definitions

We now formally describe the experiments and present our definition of security.

The **Real** *experiment.* The real-world experiment is parameterized by a security parameter λ, an adversarial host $\mathcal{H}$ and a non-negative integer q_{forge}. At the start of the experiment, compute $(K, \mathsf{PP}_{\mathsf{com}}) \leftarrow \mathsf{Setup}(1^{\lambda})$ and output $\mathsf{PP}_{\mathsf{com}}$ to $\mathcal{H}$. Subsequently, the adversary $\mathcal{H}$ is given access to three oracles: an *enclave* oracle; a *ledger* oracle, and a *forgery* oracle. The adversary can make an unlimited number of calls to the first two oracles, but is restricted to making at most q_{forge} queries to the forgery oracle. The oracles operate as follows:

The Ledger Oracle. The ledger oracle honestly implements the ledger interface used in the real protocol (see Section 2.1).

The Enclave Oracle. The enclave oracle is initialized with the secret K and parameters $\mathsf{PP}_{\mathsf{com}}$. Each time the user executes this oracle on some input $(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \pi_{\mathsf{publish},i}, \mathsf{post}_i)$, the oracle runs $\mathsf{ExecuteEnclave}_{K,\mathsf{PP}_{\mathsf{com}}}(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \pi_{\mathsf{publish},i}, \mathsf{post}_i)$. It returns the resulting output to $\mathcal{H}$.

The Forgery Oracle. This special oracle allows up to q_{forge} queries by $\mathcal{H}$. On input a string S, the forgery oracle computes a valid ledger authenticator π_{publish} over S *but does not place this data on the ledger*. It returns $(S, \pi_{\mathsf{publish}})$.

At the conclusion of the experiment, the adversary produces an output. This output is the result of the experiment.

The **Ideal** *experiment.* This experiment is parameterized by a security parameter λ, an ideal-world adversarial host $\hat{\mathcal{H}}$ and a non-negative integer q_{forge}. The adversary is given access to two oracles: a *compute* oracle and a *fork* oracle. These oracles operate as follows:

The Compute Oracle. When $\hat{\mathcal{H}}$ makes the i^{th} query to this oracle on input $(P, \mathsf{I}, \mathsf{CID})$, the oracle first checks an internal table to find the tuple labeled $(j, \mathsf{real}, \mathsf{S}, \mathsf{CID})$ where j has the highest value of all matching tuples in the table. If no such tuple is found, it sets $\mathsf{S} \leftarrow \varepsilon$. Now it samples random coins r_i uniformly at random and computes:

$$(\mathsf{O}, \mathsf{Pub}, \mathsf{S}') \leftarrow P(\mathsf{I}, \mathsf{S}; \bar{r})$$

The oracle stores $(i, \mathsf{real}, \mathsf{S}', \mathsf{CID})$ in its internal table, places Pub on a list of "public outputs" and returns $(\mathsf{O}, \mathsf{Pub})$ to the caller.

The Fork Oracle. This specialized oracle may be called by $\hat{\mathcal{H}}$ at most q_{forge} times. It operates similarly to the **Compute** oracle, except that it allows the caller execute the program on any *any past state* in that oracle's table. On input $(P, \mathsf{I}_i, \mathsf{CID}, j, l \in \{\mathsf{real}, \mathsf{fork}\})$, this oracle finds the tuple $(j, l, \mathsf{S}, \mathsf{CID})$ in the **Compute** oracle's internal table, samples random coins and computes:

$$(\mathsf{O}, \mathsf{Pub}, \mathsf{S}') \leftarrow P(\mathsf{I}, \mathsf{S}; \bar{r})$$

The oracle now stores $(i, \mathsf{fork}, \mathsf{S}', \mathsf{CID})$ in its internal table. It discards Pub and returns O to the caller.

At the conclusion of the experiment, the adversary $\hat{\mathcal{H}}$ produces an output, which is the output of the experiment. With these considerations in mind, we now present our main security definition:

Definition 15 (Simulation security for ELI) An ELI scheme

$$\Pi = (\mathsf{ExecuteApplication}, \mathsf{ExecuteEnclave})$$

is *simulation-secure* if for every *p.p.t.* adversary $\mathcal{H}$, sufficiently large λ, and non-negative q_{forge}, there exists a p.p.t. $\hat{\mathcal{H}}$ such that the following holds:

$$\mathbf{Real}(\mathcal{H}, \lambda, q_{\mathsf{forge}}) \overset{c}{\approx} \mathbf{Ideal}(\hat{\mathcal{H}}, \lambda, q_{\mathsf{forge}})$$

Discussion. The **Forgery** and **Fork** oracles in the experiments above have similar purposes. Each is designed to mimic the degradation in security that occurs in the event of a ledger forgery. If $q_{\mathsf{forge}} = 0$, neither oracle may ever be called; this allows us to model the case where the ledger operates perfectly. If $q_{\mathsf{forge}} > 0$ then we wish our ELI protocol to minimize the damage caused by such forgeries. In the ideal world, we define that damage as *the ability to execute one additional computation* on pre-existing state, without updating the current state (or producing a public output). An important implication of this definition is that the resulting output states produced by the **Fork** oracle may *not* be fed to the **Compute** oracle. This prevents the attacker from spawning an entirely new execution branch from a single forgery.

3.3.4 Third Party Privacy

Our main security definition above considers the integrity of ELI computations in the face of an adversarial host application. However, a useful scheme should also provide privacy for an honest host (and user) against curious third parties, who may view the ledger in order to recover confidential user inputs.

For space reasons we leave a formal definition of this property to the full version of this work. However, we provide an informal discussion here. We address this concern via a separate security definition that models an honest enclave, an honest host application and an *honest-but-curious* ledger that acts as the observer. Intuitively we wish to that the ledger cannot learn any information about the I_i, even when she can specify the distribution of inputs that $\mathcal{H}$ executes on.

In practice we address this property via a real/ideal world definition. In the *real-world experiment*, the adversarial ledger $\mathcal{L}^{\mathsf{Verify}}$ is provided with an oracle that allows her to request that host run an instance of the computation on a given input and CID. In response, $\mathcal{L}^{\mathsf{Verify}}$ receives all data posted to the ledger. In the *ideal-world experiment*, we require the existence of a simulator $\mathcal{S}$ that, on input a valid query from $\mathcal{L}^{\mathsf{Verify}}$ (*i.e.,* a query that would produce a valid output from P), makes the necessary post to $\mathcal{L}^{\mathsf{Verify}}$. The key element of this definition is that $\mathcal{S}$ does receive the input I chosen by $\mathcal{L}^{\mathsf{Verify}}$, and so it necessarily cannot leak information about the user's input. Our definition requires that no p.p.t. adversary $\mathcal{L}^{\mathsf{Verify}}$ can distinguish the output of these two experiments.

3.4 ELI Construction

In this section we present a specific construction of an Enclave-Ledger Interaction scheme. Our construction makes black box use of public ledger $\mathcal{L}^{\mathsf{Verify}}$ with ledger hash H_{L}, commitment schemes Π_{com}, authenticated symmetric encryption Π_{AE}, collision-resistant hash function H that maps to $\{0,1\}^{2\ell}$, for $\ell = \text{poly}(\lambda)$ and and pseudorandom function PRF.

3.4.1 Main Construction

We now present our main construction for a Enclave-Ledger Interaction scheme and address its security. Recall that an ELI consists of the three algorithms with the interface described in Figure 3.2. We present pseudocode for our construction in Algorithms 1, 2 and 3 below.

Discussion. The scheme we present in this section differs somewhat from the pedagogical scheme we discussed in the introduction. Many of these differences address minor details that affect efficiency or simplify our security analysis: for example we do not encrypt state directly using the fixed key K, but instead derive a unique per-execution key k using a pseudorandom function (PRF). This simplifies our analysis by allowing us to instantiate with a single-message authenticated encryption scheme (*e.g.*, an AE scheme with a hard-coded nonce) without concerns about how to deal with encrypting multiple messages on a single key.

A second modification from our pedagogical construction is that we evaluate a pseudorandom function on the hash of the *structure* returned by the

Algorithm 1: Setup

Data: Input: 1^λ
Result: Secret K for the enclave and public commitment parameters $\mathsf{PP}_{\mathsf{com}}$

$K \xleftarrow{\$} \{0,1\}^\lambda$
$\mathsf{PP}_{\mathsf{com}} \leftarrow \Pi_{\mathsf{com}}.\mathsf{Setup}(1^\lambda)$
Output $(K, \mathsf{PP}_{\mathsf{com}})$

Algorithm 2: ExecuteApplication

Data: Input: $\mathsf{PP}_{\mathsf{com}}, P$
// Set counter to 0 and state to ε
$\mathcal{S}_0 \leftarrow \varepsilon$
$i \leftarrow 0$
$\mathsf{Pub}_0 \leftarrow \varepsilon$
// Loop and run the program
while *true* **do**
  Obtain I_i from the user
  if $\mathsf{I}_i = \perp$ **then**
    Terminate
  $r_i \xleftarrow{\$} \{0,1\}^\ell$
  $C_i \leftarrow \Pi_{\mathsf{com}}.\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, (i, \mathsf{I}_i, \mathcal{S}_i, P); r_i)$
  $(\pi_{\mathsf{publish},i}, \mathsf{post}_i) \leftarrow \mathsf{Ledger.Post}\left((\mathsf{Pub}_i, C_i)\right)$
  $(\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_{i+1}) \leftarrow \mathsf{ExecuteEnclave}(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \pi_{\mathsf{publish},i}, \mathsf{post}_i)$
  Output $(\mathsf{O}_i, \mathsf{Pub}_{i+1})$ to the user
  $i \leftarrow i + 1$

Algorithm 3: ExecuteEnclave

Data: Input: $(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \pi_{\mathsf{publish},i}, \mathsf{post}_i)$
Internal values: $K, \mathsf{PP}_{\mathsf{com}}$
Result: $(\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_{i+1})$ or $\perp$
// Verify and parse the inputs
$\mathsf{Assert}\left(\mathsf{Ledger.Verify}(\mathsf{post}_i, \pi_{\mathsf{publish},i})\right)$
$\mathsf{Assert}(\mathsf{post}_i.\mathsf{Hash} = \mathsf{H}_{\mathsf{L}}(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash}))$
$\mathrm{Parse}\ (\mathsf{Pub}_i, C_i) \leftarrow \mathsf{post.Data}$
$\mathsf{Assert}(C_i = \Pi_{\mathsf{com}}.\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, (i, \mathsf{I}_i, \mathcal{S}_i, P); r_i))$
// Compute the i^{th} state encryption key
$(k_i, \cdot) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{PrevHash})$
if $\mathcal{S}_i = \varepsilon$ **then**
  // First execution step, no state.
  $\mathsf{Assert}(i = 0)$
  $S_i = \varepsilon$
else
  $(S_i, H_P) \leftarrow \Pi_{AE}.\mathsf{Decrypt}^{\mathsf{unpad}}(k_i, \mathcal{S}_i)$
  $\mathsf{Assert}((S_i, H_P) \neq \perp)$
  $\mathsf{Assert}(H_P = \mathsf{H}(P \| i \| \mathsf{Pub}_i))$
// Compute randomness and $i + 1^{th}$ encryption key
$(k_{i+1}, \bar{r}_i) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{Hash})$
// Run the program and abort if it fails
$(S_{i+1}, \mathsf{Pub}_{i+1}, \mathsf{O}_i) \leftarrow P(S_i, \mathsf{I}_i; \bar{r}_i)$
$\mathsf{Assert}((S_{i+1}, \mathsf{Pub}_{i+1}, \mathsf{O}_i) \neq \perp)$
// Encrypt the resulting state
$\mathcal{S}_{i+1} \leftarrow \Pi_{AE}.\mathsf{Encrypt}^{\mathsf{pad}}(k_{i+1}, (S_{i+1}, \mathsf{H}(P \| i + 1 \| \mathsf{Pub}_{i+1}))$
$\mathrm{Output}\ (\mathcal{S}_{i+1}, \mathsf{O}_i, \mathsf{Pub}_{i+1})$

ledger. By the nature of our ledger abstraction, this data structure enforces a hash chain over all previous transactions; as a result this ensures that all random coins and keys are themselves a function of the *full execution history of the program*. This ensures that an attacker – even a powerful one that can forge some ledger outputs – cannot use the state resulting from those forgeries to continue normal execution via the real ledger, since the execution history on the real ledger will not contain these forgeries.

We remark again that this more powerful ledger abstraction does not truly represent a stronger assumption when compared to our pedagogical construction, since the more powerful ledger can be "simulated" by enclave itself, provided the enclave has access to the full contents of a simple ledger.

Security We now present our main security theorem.

Theorem 2 Assuming a secure commitment scheme, a secure authenticated encryption scheme (in the sense of (Rogaway, 2002)); that H and H_{L} are collision resistant; PRF is pseudorandom; and that ledger authentication tags are unforgeable, then the scheme $\Pi = (\mathsf{Setup}, \mathsf{ExecuteEnclave}, \mathsf{ExecuteApplication})$ presented in Algorithms 1, 2 and 3 satisfies Definition 15.

3.4.2 Proof of Security

Let $\mathcal{H}$ be an adversarial host that plays the **Real** experiment. We now construct an ideal-world adversarial host $\hat{\mathcal{H}}$ such that, for sufficiently large λ and all non-negative q_{forge},

$$\mathbf{Real}(\mathcal{H}, \lambda, q_{\mathsf{forge}}) \overset{c}{\approx} \mathbf{Ideal}(\hat{\mathcal{H}}, \lambda, q_{\mathsf{forge}}).$$

Proof. [Proof of Theorem 2] To prove this statement, we first describe the operation of $\hat{\mathcal{H}}$. We then proceed to demonstrate that if there exists a p.p.t. distinguisher algorithm that distinguishes the output of the two experiments with non-negligible advantage, then one or more of the following are true: (1) there exists an adversary that breaks the SUF-AUTH security of the ledger authentication tags, (2) there is an attack on the binding property of the commitment scheme, (3) there exists a distinguisher for the pseudorandom function PRF, (4) there is an adversary that succeeds against the authenticated encryption scheme, or (5) there exists an attacker that breaks the collision-resistance of a hash function.

The operation of $\hat{\mathcal{H}}$. The ideal-world adversary $\hat{\mathcal{H}}$ runs the real adversary $\mathcal{H}$ internally, and simulates for it the oracles of the **Real** experiment. We will assume that $\hat{\mathcal{H}}$ also has access to an oracle that simulates the real ledger and produces authentication tags, in addition to a *ledger forgery* oracle that allows for the production of "forged" ledger signatures (see Section 2.1 for a discussion of this model.)

$\hat{\mathcal{H}}$ runs the **Ideal** experiment. Internally, it maintains the tables T_{enclave}, and T_{forge} as well as a view of the ledger L. The table T_{enclave} contains entries of the following form:

$$(P, i, \mathsf{I}_i, r_i, \mathcal{S}_{in}, \mathcal{S}_{out}, \mathsf{post}, \mathsf{O}, \mathsf{Pub}, \mathsf{CID})$$

At the start of the experiment, our simulation generates $\mathsf{PP}_{\mathsf{com}} \leftarrow \Pi_{\mathsf{com}}.\mathsf{Setup}(1^\lambda)$. Each query $\mathcal{H}$ makes to the **Real** oracles is handled as follows:

The Ledger Oracle. When $\mathcal{H}$ queries the ledger oracle on $(\mathsf{Data}, \mathsf{CID})$, $\hat{\mathcal{H}}$ executes Ledger.Post algorithm (see Section 2.1). It stores $(\mathsf{Data}, \mathsf{CID}, \pi_{\mathsf{publish}})$ in a table, and returns $(\mathsf{post}, \pi_{\mathsf{publish}})$ to $\mathcal{H}$. If two distinct outputs share the same value post.Hash, the simulation aborts with an error.

The Forgery Oracle. This oracle responds to at most q_{forge} queries made by $\mathcal{H}$. When $\mathcal{H}$ queries on an arbitrary string S, $\hat{\mathcal{H}}$ queries the authentication oracle for a tag π_{publish} and records the pair $(S, \pi_{\mathsf{publish}})$ in T_{forge}.

The Enclave Oracle. When $\mathcal{H}$ queries the enclave oracle on input $(P, i, \mathcal{S}_i, \mathsf{I}_i, r_i, \pi_{\mathsf{publish},i}, \mathsf{post}_i)$, $\hat{\mathcal{H}}$ performs the following steps:

1. It performs all of the publicly-verifiable checks in the ExecuteEnclave algorithm (*i.e.,* all checks that do not rely on the secret K.) It will abort and output a defined error message if $\mathcal{H}$ outputs an authenticator/message pair that was not produced by the **Ledger** or **Forgery** oracles, a hash collision in H or H_{L}, or a collision in the commitment scheme. $\mathsf{Event}_{\mathsf{hashcoll}}$.
2. It verifies that $(\mathsf{post}, \pi_{\mathsf{publish}})$ has either been posted to the ledger L, or exists in T_{forge}.
3. If $\mathcal{S}_i = \epsilon$ it looks up CID in the table T_{enclave} and aborts if any matching entry is found.
4. It searches T_{enclave} for an entry where $\mathcal{S}^*_{out} = \mathcal{S}_i$ and aborts if no result is found. Otherwise it obtains the tuple

$$(P^*, i^*, \mathsf{I}^*_i, {r_i}^*, \mathcal{S}^*_{in}, \mathcal{S}^*_{out}, \mathsf{post}^*, \mathsf{O}^*, \mathsf{Pub}^*, \mathsf{CID}^*)$$

in T_{enclave} where $\mathcal{S}^*_{out} = \mathcal{S}_i$.

5. It checks that $i = i^* + 1$.
6. It checks that $\mathsf{CID} = \mathsf{post}.\mathsf{CID} = \mathsf{post}^*.\mathsf{CID}$.
7. It checks that $\mathsf{post}.\mathsf{PrevHash} = \mathsf{post}^*.\mathsf{Hash}$.
8. It checks that $\mathsf{post}.\mathsf{Pub} = \mathsf{Pub}^*$.
9. It checks that $P = P^*$.

   If any of the above checks fail, $\hat{\mathcal{H}}$ aborts and outputs $\perp$ to $\mathcal{H}$.

10. $\hat{\mathcal{H}}$ first samples $k_{i+1} \leftarrow \{0,1\}^\ell$ uniformly at random, and computes a "dummy" ciphertext

$$\mathcal{S}'_{out} \leftarrow \Pi_{AE}.\mathsf{Encrypt}(k_{i+1}, (1^{\mathsf{Max}(P)}, 1^\ell)).$$

    $\hat{\mathcal{H}}$ now selects one of the following three cases. The first case handles *repeated* inputs that have already been previously seen. The second case handles inputs that contain valid authenticators from the **Ledger** oracle. The final case handles inputs that have been authenticated by the **Forge** oracle.

11. **Process repeated program inputs.** Whenever $\hat{\mathcal{H}}$ receives an input, it searches its table for any entry that matches $(P, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i)$. (Note that we explicitly exclude $\pi_{\mathsf{publish},i}$ from this check.[8]) If a

[8] Recall that our simulation has already verified that $\pi_{\mathsf{publish},i}$ passes the value passes the

matching entry exists, it obtains $(\mathsf{O}, \mathcal{S}_{\mathsf{out}}, \mathsf{Pub}_{\mathsf{out}})$ from the same entry in the table and outputs the tuple $(\mathcal{S}_{out}, \mathsf{O}, \mathsf{Pub}_{\mathsf{out}})$ to $\mathcal{H}$ and halts.

12. **Process new, and unforged inputs.** Otherwise, if $(\mathsf{post}_i, \pi_{\mathsf{publish},i})$ is contained within the table maintained by the **Ledger** oracle, then $\hat{\mathcal{H}}$ looks up $\mathsf{post}_i.\mathsf{Hash}$ in the **Ledger**'s table to obtain CID. If this produces multiple possible matches, it aborts and outputs $\mathsf{Event}_{\mathsf{ledgerrepeat}}$.

    Next, $\hat{\mathcal{H}}$ calls the **Compute** oracle on $(P, \mathsf{I}_i, \mathsf{CID})$ to obtain $(\mathsf{O}', \mathsf{Pub}')$. $\hat{\mathcal{H}}$ now stores $(P, i, \mathsf{I}_i, r_i, \mathcal{S}_i, \mathcal{S}'_{out}, \mathsf{post}, \mathsf{O}', \mathsf{Pub}')$ in T_{enclave}. It returns $(\mathcal{S}'_{out}, \mathsf{O}', \mathsf{Pub}')$ to $\mathcal{H}$.

13. **Process new, forged inputs.** Otherwise, if $(\mathsf{post}, \pi_{\mathsf{publish}})$ is contained within T_{forge}, then $\hat{\mathcal{H}}$ identifies the call j at which this value was added to the table, and calls the **Fork** oracle on input $(P, \mathsf{I}, \mathsf{CID}, j)$ to obtain $(\mathsf{O}', \mathsf{Pub}')$. As in the previous step, it stores

$$(P, i||j, \mathsf{I}_i, r_i, \mathcal{S}_i, \mathcal{S}'_{out}, \mathsf{post}, \mathsf{O}', \mathsf{Pub}', \mathsf{CID})$$

    in T_{enclave}. It returns $(\mathcal{S}'_{out}, \mathsf{O}', \mathsf{Pub}')$ to $\mathcal{H}$.

Discussion. Note that there are three main cases in the simulation above. Whenever the adversary queries the enclave on an state/counter that has previously been queried, $\hat{\mathcal{H}}$ simply returns the same output as it did during the previous query (this occurs at step 11). This is possible because the output

check $\mathsf{Assert}\left(\mathsf{Ledger.Verify}(\mathsf{post}_i, \pi_{\mathsf{publish},i})\right)$. Provided that this check succeeds, it is easy to see that the value $\pi_{\mathsf{publish},i}$ has no further influence on the output of ExecuteEnclave.

of the enclave in the protocol is determined solely by the given inputs (if all public checks pass), and thus repeated inputs cause the enclave to produce the same behavior.

When the adversary queries on a new input that has been posted to the ledger, $\hat{\mathcal{H}}$ queries the ideal **Compute** oracle to obtain the appropriate output, and generates a simulated ("dummy ciphertext") encrypted state to return to $\mathcal{H}$. Similarly, when $\mathcal{H}$ queries the enclave on a forged (but otherwise correct) ledger output, $\hat{\mathcal{H}}$ queries the **Fork** oracle to obtain the correct output and also returns a dummy ciphertext to $\mathcal{H}$.

Indistinguishability of $\hat{\mathcal{H}}$'s simulation. Let $\mathcal{Z}$ be a p.p.t. distinguisher that succeeds in distinguishing $\hat{\mathcal{H}}$'s output in the **Ideal** experiment from $\mathcal{H}$'s output in the **Real** experiment with non-negligible advantage. We now show that such an adversary violates one of our assumptions above. The proof proceeds via a series of hybrids, where in each hybrid $\mathcal{H}$ interacts as in the **Real** experiment. The first hybrid (**Game 0**) is identically distributed to the **Real** experiment, and the final hybrid represents $\hat{\mathcal{H}}$'s simulation above. For notational convenience, let $\mathbf{Adv}\left[\,\mathbf{Game\ i}\,\right]$ be $\mathcal{Z}$'s absolute advantage in distinguishing the output of **Game i** from **Game 0**, *i.e.,* the **Real** distribution.

Game 0. In this hybrid, $\mathcal{H}$ interacts with the **Real** experiment.

Game 1 (**Abort on [adversary-]forged authenticators.**) This hybrid modifies the previous as follows: in the event that $\mathcal{H}$ queries the **Enclave** oracle on any pair $(\mathsf{post}, \pi_{\mathsf{publish}})$ such that (1) $\mathsf{Ledger.Verify}(\mathsf{post}, \pi_{\mathsf{publish}}) = 1$, and yet (2) the pair was not the input (resp. output) of a previous call

to either the **Ledger** or **Forgery** oracles, then abort and output $\mathsf{Event}_{\mathsf{forge}}$. We note that if $\mathcal{H}$ causes this event to occur with non-negligible probability, then we can trivially use $\mathcal{H}$ to construct an attack on the SUF-AUTH security of the contract authenticator with related probability. Since by assumption the probability of such an event is negligible, we bound $\mathbf{Adv}\left[\,\mathbf{Game\ 1}\,\right] \leq \mathsf{negl}_1(\lambda)$.

Game 2 (Abort on hash collisions.) This hybrid modifies the previous as follows: if at any point during the experiment $\mathcal{H}$ causes the functions $\mathsf{H}, \mathsf{H}_{\mathsf{L}}$ to be evaluated on inputs $s_1 \neq s_2$ such that $\mathsf{H}(s_1) = \mathsf{H}(s_2)$ or $\mathsf{H}_{\mathsf{L}}(s_1) = \mathsf{H}_{\mathsf{L}}(s_2)$, then abort and output $\mathsf{Event}_{\mathsf{hashcoll}}$. Under the assumption that the hash functions $\mathsf{H}, \mathsf{H}_{\mathsf{L}}$ are collision-resistant, we have that $|\mathbf{Adv}\left[\,\mathbf{Game\ 2}\,\right] - \mathbf{Adv}\left[\,\mathbf{Game\ 1}\,\right]| \leq \mathsf{negl}_2(\lambda)$.

Game 3 (Abort on commitment collisions.) This hybrid modifies the previous as follows: in the event that $\mathcal{H}$ queries the **Enclave** oracle at steps i, j where

$$C_i = C_j =$$

$$\Pi_{\mathsf{com}}.\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, (i, \mathsf{I}_i, \mathcal{S}_i, P, \mathsf{post}_i.\mathsf{CID}); r_i) = \Pi_{\mathsf{com}}.\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, j \| \mathsf{I}_j \| \mathcal{S}_j \| \mathsf{post}_i.\mathsf{CID}; r_j$$

and yet $(i, \mathsf{I}_i, \mathcal{S}_i, P_i, \mathsf{post}_i.\mathsf{CID}) \neq (j, \mathsf{I}_j, \mathcal{S}_j, P_j, \mathsf{post}_j.\mathsf{CID})$, then abort and output $\mathsf{Event}_{\mathsf{binding}}$. We note that if the commitment scheme is binding, this event will occur with at most negligible probability, hence $|\mathbf{Adv}\left[\,\mathbf{Game\ 3}\,\right] - \mathbf{Adv}\left[\,\mathbf{Game\ 2}\,\right]| \leq \mathsf{negl}_3(\lambda)$.

Game 4 (**Duplicate Enclave calls give identical outputs.)** This hybrid modifies the previous as follows: when $\mathcal{H}$ queries the **Enclave** oracle repeatedly on the same values $(P_i, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i)$ (here we exclude $\pi_{\mathsf{publish},i}$) and the oracle (as implemented in the previous hybrid) does not output $\perp$, replace the response to all repeated queries subsequent to the first query with the same result as the first query. Recall that by definition the **Enclave** oracle's ExecuteEnclave calculation is deterministic and solely based on the inputs provided above. Thus, repeated queries on the same input will always produce the same output. Hence by definition $|\mathbf{Adv}\,[\,\mathbf{Game\ 4}\,] - \mathbf{Adv}\,[\,\mathbf{Game\ 3}\,]| = 0$.

Game 5 (**Abort on colliding ledger hashes.)** This hybrid modifies the previous as follows: when $\mathcal{H}$ calls the **Enclave** oracle on two distinct inputs $(P_i, i, \mathsf{I}_i, \mathcal{S}_i, \mathsf{post}_i, \pi_{\mathsf{publish},i})$ and $(P_j, j, \mathsf{I}_j, \mathcal{S}_j, \mathsf{post}_j, \pi_{\mathsf{publish},j})$, and if the two inputs do *not* represent repeated inputs (according to **Game 4**), then: if both $(\mathsf{post}_i, \pi_{\mathsf{publish},i})$ and $(\mathsf{post}_j, \pi_{\mathsf{publish},j})$ are valid outputs of the **Ledger** oracle and yet $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$ then abort and output $\mathsf{Event}_{\mathsf{ledgercoll}}$. By Lemma 1 this event will occur with at most negligible probability if H_L is collision-resistant and the ledger is implemented correctly. This bounds $|\mathbf{Adv}\,[\,\mathbf{Game\ 5}\,] - \mathbf{Adv}\,[\,\mathbf{Game\ 4}\,]| \leq \mathsf{negl}_4(\lambda)$.

Game 6 ($\hat{\mathcal{H}}$ **can always uniquely identify** CID**.)** This hybrid modifies the previous as follows: if at step i the adversary $\mathcal{H}$ calls the **Enclave** the oracle (as implemented in the previous hybrid) and (1) the oracle does not return $\perp$, (2) the inputs to the two calls are not identical (this would be excluded by the earlier hybrids), and (3) the pair $(\mathsf{post}_i, \pi_{\mathsf{publish},i})$ are

in the **Ledger** table, and (4) $\mathsf{post}_i.\mathsf{Hash}$ matches two distinct entries in the **Ledger** table, then abort and output $\mathsf{Event}_{\mathsf{ledgerrepeat}}$. By Lemma 2 this event will occur with at most negligible probability. This bounds $|\mathbf{Adv}[\mathbf{Game\ 6}] - \mathbf{Adv}[\mathbf{Game\ 5}]| \leq \mathsf{negl}_5(\lambda)$.

Game 7 (**Replace the session keys and pseudorandom coins with random strings.)** This hybrid modifies the previous as follows: if **Enclave** does not abort or is not called on repeated inputs, then the pair $(k_{i+1}, \bar{r}_i)$ is sampled uniformly at random and recorded in a table for later use. Recall that in the preceding hybrid, this pair is generated as $(k_{i+1}, \bar{r}_i) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{Hash})$ where (by the previous hybrids) $\mathsf{post}_i.\mathsf{Hash}$ is guaranteed not to repeat. Similarly, the output of each call $(k_i, \cdot) \leftarrow \mathsf{PRF}_K(\mathsf{post}_i.\mathsf{PrevHash})$ is also replaced with a random value when the input $\mathsf{post}_i.\mathsf{PrevHash}$ has not been queried to PRF_K previously, or the appropriate value (drawn from the table) when this value has been previously queried. If PRF is pseudorandom then if $|\mathbf{Adv}[\mathbf{Game\ 5}] - \mathbf{Adv}[\mathbf{Game\ 4}]|$ is non-negligible, then we can construct an algorithm that succeeds in distinguishing the PRF from a random function. This bounds $|\mathbf{Adv}[\mathbf{Game\ 7}] - \mathbf{Adv}[\mathbf{Game\ 6}]| \leq \mathsf{negl}_6(\lambda)$.

Game 8 (**Reject inauthentic ciphertexts.)** This hybrid modifies the previous as follows: if $\mathcal{H}$ queries the **Enclave** oracle on an input $\mathcal{S}_i \neq \varepsilon$ such that (1) the oracle does not reject the input, (2) $\Pi_{AE}.\mathsf{Decrypt}(k_i, \mathcal{S}_i)$ does not output $\perp$, and yet (3) the pair $(\mathcal{S}_i, k_i)$ was not generated during a previous query to **Enclave**, then abort and output $\mathsf{Event}_{\mathsf{auth}}$. We note that that in the previous hybrid each key k_i is generated at random, and only used

to encrypt one ciphertext. Thus if $\mathcal{H}$ is able to produce a second ciphertext that satisfies decryption under this key, then we can construct an attacker that forges ciphertexts in the authenticated encryption scheme with non-negligible advantage. Because we assumed the AE scheme was secure, we have that $|\mathbf{Adv}[\mathbf{Game\ 8}] - \mathbf{Adv}[\mathbf{Game\ 7}]| \leq \mathsf{negl}_7(\lambda)$.

Game 9 (**Abort if inputs are inconsistent.**) This hybrid modifies the previous as follows: on $\mathcal{H}$'s the i^{th} query to the **Enclave** oracle, when the input $\mathcal{S}_i \neq \varepsilon$, let $(\mathsf{post\_CID}', P', i', \mathsf{Pub}_{i-1}{}')$ be the inputs/outputs associated with the previous **Enclave** call that produced $\mathcal{S}_i$. If (1) the experiment has not already aborted due to a condition described in previous hybrids and (2) if the **Enclave** oracle as implemented in the previous hybrid does not reject the input, and (3) any of the provided inputs $(\mathsf{post\_CID}, P, i-1, \mathsf{Pub}_{i-1}) \neq (\mathsf{post\_CID}', P', i', \mathsf{Pub}_{i-1}{}')$ differ from those associated with the previous call to the **Enclave**, abort and output $\mathsf{Event}_{\mathsf{mismatch}}$. By Lemma 3 we have that $|\mathbf{Adv}[\mathbf{Game\ 9}] - \mathbf{Adv}[\mathbf{Game\ 8}]| = 0$.

Game 10 (**Replace ciphertexts with dummy ciphertexts.**) This hybrid modifies the previous as follows: we modify the generation of each ciphertext $\mathcal{S}'_{out}$ to encrypt the unary string $(1^{\mathsf{Max}(P)}, 1^{\ell})$. We first note that the length of the (padded) plaintext is identical between this and the previous hybrid, by the definition of the padding function. Thus if $\mathcal{H}$'s output is significantly different between this and the previous hybrid, we can construct an attacker that succeeds against the confidentiality of the authenticated encryption scheme. By the assumption that the AE scheme is secure, we have that $|\mathbf{Adv}[\mathbf{Game\ 10}] - \mathbf{Adv}[\mathbf{Game\ 9}]| \leq \mathsf{negl}_8(\lambda)$.

We note that **Game 10** is distributed identically to the simulation provided to $\mathcal{H}$ by $\hat{\mathcal{H}}$. By summation over the hybrids we have that $\mathbf{Adv}\left[\mathbf{Game\ 9}\right] \leq \mathsf{negl}_1(\lambda) + \cdots + \mathsf{negl}_8(\lambda)$ and thus is also negligible in λ. This implies that no p.p.t. distinguisher can distinguish the output of **Real** and **Ideal** experiments with non-negligible advantage.

Lemma 1 (Uniqueness of Ledger Identifiers) If H_L is collision resistant and the ledger operates as described in Section 2.1, then for *non-repeated* inputs (those not caught in **Game 4**) with all but negligible probability no two calls $i \neq j$ to the **Enclave** oracle in **Game 5** will have $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$.

Proof. [Proof of Lemma 1] Let us imagine that with some non-negligible probability ϵ, $\mathcal{H}$ is able to play **Game 5** such that two distinct calls $i \neq j$ to the **Enclave** oracle have $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$ and yet the two calls were not identified as repeated inputs (according to the conditions of **Game 4**). We show that this implies a collision in the hash function H_L.

Let $(P_j, j, \mathsf{l}_j, \mathcal{S}_j, \mathsf{post}_j), (P_i, i, \mathsf{l}_i, \mathcal{S}_i, \mathsf{post}_i)$ be the inputs to the **Enclave** oracle (we exclude π_{publish}). Let

$$(\mathsf{Data}_i, \mathsf{CID}_i, \mathsf{post}_i.\mathsf{Hash}), (\mathsf{Data}_j, \mathsf{CID}_j, \mathsf{post}_j.\mathsf{Hash})$$

represent $\mathcal{H}$'s input (resp. output) from the distinct calls to the **Ledger** oracle. We now show that if two such calls produce identical $\mathsf{post}_i.\mathsf{Hash}$, then the inputs to H_L are not equal with probability related to ϵ.

If the two calls to **Enclave** are *not* repeated inputs, then for $i \neq j$ it holds that:

$$(P_j, j, \mathsf{l}_j, \mathcal{S}_j, \mathsf{post}_j) \neq (P_j, j, \mathsf{l}_j, \mathcal{S}_j, \mathsf{post}_j)$$

Recall as well that:

$$\mathsf{post}_i.\mathsf{Hash} = \mathsf{H}_\mathsf{L}(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash})$$

$$\mathsf{post}_j.\mathsf{Hash} = \mathsf{H}_\mathsf{L}(\mathsf{post}_j.\mathsf{Data} \| \mathsf{post}_j.\mathsf{PrevHash})$$

And:

$$\mathsf{post}_i.\mathsf{Data} = \Pi_{\mathsf{com}}.\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, (i, \mathsf{I}_i, \mathcal{S}_i, P_i); r_i)$$

$$\mathsf{post}_j.\mathsf{Data} = \Pi_{\mathsf{com}}.\mathsf{Commit}(\mathsf{PP}_{\mathsf{com}}, (j, \mathsf{I}_j, \mathcal{S}_j, P_j); r_j)$$

Thus note that if $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$ then this would imply that the simulation would abort with either $\mathsf{Event}_{\mathsf{hashcoll}}$ or $\mathsf{Event}_{\mathsf{binding}}$. Since this has not occurred, then the probability of such a collision is 0.

Lemma 2 (No duplicate Ledger identifiers) If H_L is collision-resistant, then for all $\mathcal{H}$, $\mathbf{Pr}\left[\mathsf{Event}_{\mathsf{ledgerrepeat}}\right] \leq \mathsf{negl}_6(\lambda)$.

Proof. [Proof of Lemma 2] Let us assume by contradiction that $\mathcal{H}$ is able to query the **Ledger** oracle on two distinct inputs such that the oracle returns (and records in its table) two distinct transactions post_i and post_j such that $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$. Then we construct a second adversary $\mathcal{A}$ that outputs a collision in the hash function H_L. $\mathcal{A}$ conducts the experiment of **Game 6** with $\mathcal{H}$.

Let i, j be any two integers. If at any point $\mathcal{H}$, at the j^{th} call to the **Ledger** oracle, submits a pair $(\mathsf{Data}, \mathsf{CID})$ such that $\mathsf{post}_j.\mathsf{Hash} = \mathsf{post}_i.\mathsf{Hash}$, then $\mathcal{A}$ terminates and outputs the collision pair $(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash}), (\mathsf{post}_j.\mathsf{Data} \| \mathsf{post}_j.\mathsf{PrevHash})$.

Clearly if $\mathsf{post}_j.\mathsf{Hash} = \mathsf{post}_i.\mathsf{Hash}$ but

$$(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash}) \neq (\mathsf{post}_j.\mathsf{Data} \| \mathsf{post}_j.\mathsf{PrevHash})$$

then $\mathcal{A}$ has identified a collision in H_L. It remains only to show that when $\mathcal{H}$ succeeds in triggering this event, the latter inequality must hold. Our proof proceeds inductively.

1. If $j = 0$ then this condition cannot occur, as there is no previous entry in the table.

2. If $j > 0$ then there are two subconditions:

   (a) If $\mathcal{H}$ has *not* previously called the **Ledger** oracle on CID then $\mathsf{post}_j.\mathsf{PrevHash}$ is set to a unique identifier based on CID. Because there has been no previous call on input CID, there cannot exist a second value $\mathsf{post}_i.\mathsf{PrevHash}$ in the table that shares the same value. Thus $\forall i$ it holds that $\mathsf{post}_j.\mathsf{PrevHash} \neq \mathsf{post}_i.\mathsf{PrevHash}$ and thus the main inequality holds.

   (b) If $\mathcal{H}$ has previously called the **Ledger** oracle on input CID, then (by the definition of the Ledger interface) there must exist some i such that $\mathsf{post}_j.\mathsf{PrevHash} = \mathsf{post}_i.\mathsf{Hash} = \mathsf{H}_\mathsf{L}(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash})$. We now consider two subcases:

      i. If the i^{th} call to the **Ledger** oracle was the first call made on input CID, then by definition $\mathsf{post}_i.\mathsf{PrevHash} \neq \mathsf{post}_j.\mathsf{PrevHash}$ because as the root of a new chain, $\mathsf{post}_i.\mathsf{PrevHash}$ has a special structure and cannot be equal to the output of H_L.

ii. If the i^{th} call to **Ledger** was *not* the first call on input CID, and if $\mathsf{post}_j.\mathsf{PrevHash} = \mathsf{post}_i.\mathsf{PrevHash}$, then by definition there must exist a third integer $k < j$ such that the k^{th} call to the **Ledger** oracle was also on input CID and $\mathsf{post}_i.\mathsf{PrevHash} = \mathsf{post}_k.\mathsf{Hash} = \mathsf{H}_\mathsf{L}(\mathsf{post}_k.\mathsf{Data} \| \mathsf{post}_k.\mathsf{PrevHash})$. However, this event cannot occur, as this would imply that $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_k.\mathsf{Hash}$ and thus $\mathcal{A}$ would have already terminated and output a collision prior to reaching this point.

In all of the above cases

$$(\mathsf{post}_i.\mathsf{Data} \| \mathsf{post}_i.\mathsf{PrevHash}) \neq (\mathsf{post}_j.\mathsf{Data} \| \mathsf{post}_j.\mathsf{PrevHash})$$

and so if $\mathcal{H}$ triggers this condition, then $\mathcal{A}$ finds a collision in H_L. Because we assume that H_L is collision-resistant, we can bound the probability of $\mathsf{Event}_{\mathsf{ledgerrepeat}}$ to be negligible in λ.

Lemma 3 (Consistency of encrypted state) For all p.p.t. $\mathcal{H}$, $\mathbf{Pr}\left[\mathsf{Event}_{\mathsf{mismatch}}\right] = 0$.

Proof. [Proof of Lemma 3] For $\mathsf{Event}_{\mathsf{mismatch}}$ to occur, one of the following conditions must occur. First, it must be the case that $\mathcal{H}$ has previously called **Enclave** on a set of values $(\mathsf{post}_{\cdot}\mathsf{CID}', P', i', \mathsf{Pub}_{i-1}{}')$, which produced a state ciphertext $\mathcal{S}$ that embeds

$$H_P = \mathsf{H}(\mathsf{post}_{\cdot}\mathsf{CID}' \| P' \| i' \| \mathsf{Pub}'_{i-1}).$$

And simultaneously it must be the case that, on input the state ciphertext $\mathcal{S}$,

the assertion $\mathsf{Assert}(H_P = \mathsf{H}(\mathsf{CID}\|P\|i\|\mathsf{Pub}_{i-1}))$ would not fail and cause the oracle to return $\perp$.

However, for this event to occur, it would require one of the following to be true: either the state ciphertext would have to be inauthentic, resulting in a previous abort $\mathsf{Event}_{\mathsf{auth}}$ (or another abort). Or the hash function would have to include a collision, resulting in $\mathsf{Event}_{\mathsf{hashcoll}}$. Since we require that the simulation would abort on these other events before it aborts with $\mathsf{Event}_{\mathsf{mismatch}}$, this event can never occur.

Lemma 4 (Uniqueness of PRF inputs) No adversary playing **Game 5** will (with non-negligible probability) call the **Enclave** oracle at steps i, j that (1) the oracle does not return $\perp$ or the experiment aborts in one of the calls, (2) the input values $(P_1, i, \mathsf{I}_1, \mathcal{S}_1, \mathsf{CID}_1)$ and $(P_2, j, \mathsf{I}_2, \mathcal{S}_2, \mathsf{CID}_2)$ to the two oracle calls are distinct, and (3) during these invocations the evaluation $\mathsf{PRF}_K(\mathsf{post}_i.\mathsf{Hash})$ and $\mathsf{PRF}_K(\mathsf{post}_j.\mathsf{Hash})$ uses $\mathsf{post}_i.\mathsf{Hash} = \mathsf{post}_j.\mathsf{Hash}$.

Proof. [Proof of Lemma 4] Let $\mathcal{H}'$ be an adversary that succeeds in triggering the above condition with non-negligible probability ϵ. We now show this violates one of the assumptions given above.

Let us define the two (partial) input tuples provided to the **Enclave** oracle as

$$\mathsf{Input}_1 = (P_1, i, \mathsf{I}_1, \mathcal{S}_1, \mathsf{CID}_1) \text{ and } \mathsf{Input}_2 = (P_2, j, \mathsf{I}_2, \mathcal{S}_2, \mathsf{CID}_2)$$

where $\mathsf{Input}_1 \neq \mathsf{Input}_2$ (we also separately define $\mathsf{post}_1, \mathsf{post}_2$ as inputs to the oracle.) Recall that in the first instance, PRF is evaluated on $(K, \mathsf{post}_i.\mathsf{Hash})$ and in the second it is evaluated on $(K, \mathsf{post}_j.\mathsf{Hash})$. We now argue that if the

oracle does not output $\perp$ (and the experiment itself has not aborted), and if $\mathsf{Input}_i \neq \mathsf{Input}_j$ then this must imply that $\mathsf{post}_i.\mathsf{Hash} \neq \mathsf{post}_j.\mathsf{Hash}$.

Note that each of the fields $\mathsf{post}_1.\mathsf{Data}, \mathsf{post}_2.\mathsf{Data}$ embeds a commitment computed on $(\mathsf{I}_1, i, \mathcal{S}_1, P_1, \mathsf{CID}_1), (\mathsf{I}_2, j, \mathcal{S}_2, P_2, \mathsf{CID}_2)$ respectively. If the **Enclave** oracle does not abort and the experiment does not abort, then if $(\mathsf{I}_1, i, \mathcal{S}_1, P_1, \mathsf{CID}_1) \neq (\mathsf{I}_2, j, \mathcal{S}_2, P_2, \mathsf{CID}_2)$ then it must hold that (1) $\mathsf{post}_1.\mathsf{Data} \neq \mathsf{post}_2.\mathsf{Data}$ (if this were not the case then it would imply a violation of the binding property of the commitment scheme, and hence an abort with $\mathsf{Event}_{\mathsf{binding}}$), or (2) $\mathsf{post}_1.\mathsf{Hash} \neq \mathsf{post}_2.\mathsf{Hash}$ (because each Hash is computed by applying H_L to the Data field, and a collision in these values would imply an abort with $\mathsf{Event}_{\mathsf{hashcoll}}$).

Since we required that these aborts do *not* occur, it must hold that if $(\mathsf{I}_1, i, \mathcal{S}_1, P_1, \mathsf{CID}_1) \neq (\mathsf{I}_2, j, \mathcal{S}_2, P_2, \mathsf{CID}_2)$ then $\mathsf{post}_i.\mathsf{Hash} \neq \mathsf{post}_j.\mathsf{Hash}$. This concludes the proof.

3.5 Applications

We now describe several applications that use Enclave-Ledger Interaction and present the relevant implementations for each. Each application employs the main construction we presented in Section 3.4 to implement a specific functionality. Except where explicitly noted, these applications are implemented as *many-time execution* programs: this means the host can re-launch the same program P many times, but each execution thread is independent and threads do not share state.

3.5.1 Private Smart Contracts

Smart contract systems comprise a network of volunteer nodes that work together to execute multi-step interactive programs called *contracts*. These systems, which are exemplified by Ethereum and the Hyperledger platforms (*The Ethereum Project* 2018; Hyperledger, 2017) as well as research systems like Ekiden (Cheng et al., 2018) maintain a shared ledger that records both the previous and updated state of the contract following each execution of a contract program. These platforms are designed for flexibility: they are capable of executing many different contracts on a single network. Smart contract systems come in two varieties: *public* contract networks (exemplified by Ethereum (*The Ethereum Project* 2018)), where all state and program code is known to the world; and *private* contract systems where some portion of this data is held secret. In both settings the computation (and verification) is conducted by a set of nodes who are not assumed to always be trustworthy. In the public setting (*e.g.*, (*The Ethereum Project* 2018)) a single node performs each contract execution, and the remaining nodes simply verify the (deterministic) output of this calculation. This approach does not work in the private setting, where some of the program inputs are unknown to the full network.

Platforms such as Hyperledger Sawtooth (Intel Corporation, 2018) have sought to address this concern by employing trusted execution technology (Hyperledger, 2017). In these systems, contract code executes within a trusted enclave on a single node, and the TEE system generate public *attestation* signatures proving the correctness of the resulting output. The network then verifies the attestation to ensure that the execution was correct. A challenge in

these systems is to ensure that a smart contract remains *synchronized*, despite the fact that execution migrates from one host to another between steps. A secondary challenge is to ensure that the enclave only executes contract code on valid inputs from the ledger, and cannot be forced to run on arbitrary input or perform additional steps by a malicious host.

Bowman *et al.* of Intel corporation (Bowman et al., 2018) independently proposed a solution reminiscent of an ELI for securing contracts in this setting. (Figure 3.1 presents an illustration of this model.) This setting is also a natural solution for our ELI system, given that the contract system – with many distinct enclave copies – can be viewed as merely being a special case of our main construction.

To instantiate an ELI in this setting, we require the contract author to pre-position a key K within each enclave.[9] Encrypted state outputs can now be written to the ledger as a means to distribute them. Given these modifications, each enclave can simply read the current state from the ledger in order to obtain the most recent encrypted state and commitment to the next contract input (which may be transmitted by users to the network).[10] Other enclaves can *verify* the correctness of the resulting output state by either (1) verifying an attestation signature, or (2) deterministically re-computing the new state and comparing it to the encrypted state on the ledger.

[9]This can be accomplished using a broadcast encryption scheme or peer-to-peer key sharing mechanism.

[10]We assume that the ledger is authenticated using signatures. Systems such as Hyperledger propose to use TEE enclaves to construct the ledger as well as execute contracts; in these systems the ledger blocks are authenticated using digital signatures that can be publicly verified.

3.5.2 Logging and Reporting

Algorithm 4: File Access Logging $P_{logging}$

Data: Input: I_i , S_i; Constants: pk_{auditor}
Parse $(\mathsf{phase}, sk, CT, \mathsf{filename}) \leftarrow \mathsf{S}_i$
if $\mathsf{S}_i = \varepsilon$ **then** `// Generate master keypair`
  $(pk, sk) \leftarrow \mathsf{PKKeyGen}(1^\lambda)$
  $\mathsf{S}_{i+1} = (\mathsf{PUBLISH}, sk, \cdot, \cdot)$
  $(\mathsf{Pub}_{i+1}, \mathsf{O}_{i+1}) \leftarrow (\varepsilon, pk)$
else if $\mathsf{phase} = \mathsf{PUBLISH}$ **then** `// Send filename`
  Parse $\mathsf{filename} \leftarrow \mathsf{I}_i$
  $CT_{\mathsf{auditor}} \leftarrow \mathsf{PKEnc}(pk_{\mathsf{auditor}}, \mathsf{filename})$
  $\mathsf{S}_{i+1} \leftarrow (\mathsf{DECRYPT}, sk, CT, \mathsf{I}_i)$
  $(\mathsf{Pub}_{i+1}, \mathsf{O}_{i+1}) \leftarrow (CT, \mathsf{post})$
else `// Decrypt a given file`
  Parse $C \leftarrow \mathsf{I}_i$
  $(\mathsf{filename}', M) \leftarrow \mathsf{PKDec}(sk, C)$
  if $\mathsf{filename} = \mathsf{filename}'$ **then**
    $\mathsf{S}_{i+1} = (\mathsf{PUBLISH}, sk, \cdot, \cdot)$
    $(\mathsf{Pub}_{i+1} \| \mathsf{O}_{i+1}) = (\varepsilon \| M)$
  else
    Abort and output $\perp$.
output $(\mathsf{S}_{i+1}, \mathsf{Pub}_{i+1}, \mathsf{O}_{i+1})$

Several cryptographic access control systems require participants to actively log file access patterns to a remote and immutable network location (Foundation, 2018). A popular approach to solving this problem in cryptographic access control systems, leveraged by systems like Hadoop (Foundation, 2018), is to assign a unique decryption key to each file and to require that clients individually request each key from an online server, which in turn logs each request. This approach requires a trusted online server that holds decryption keys and cannot be implemented using a public ledger.

In place of a trusted server, we propose to use ELI to implement mandatory logging for protected files. In this application, a local enclave is initialized (in the first step of a program) and stores (or generates) a master key for some collection of files, *e.g.*, a set of files stored on a device.[11] The enclave then employs the *public output* field of the ELI scheme to ensure that prior to each file access the user must post a statement signaling that the file is to be accessed.[12] The logging program is presented as Algorithm 4 and consists of three phases. When the program is launched, the enclave generates a keypair for a public-key encryption scheme $(\mathsf{PKKeyGen}, \mathsf{PKEnc}, \mathsf{PKDec})$ and outputs the public key.[13] Next the user provides a filename they wish to decrypt, and the program encrypts this filename using a hard-coded public key for an *auditor*. When the user posts this key to the ledger, the program decrypts the given file.

3.5.3 Limited-attempt Password Guessing

Device manufacturers have widely deployed end-to-end file encryption for devices such as mobile phones and cloud backup data (Apple Computer, 2016; Android Project, 2017). These systems require users to manage their own secrets rather than trusting them to the manufacturer.

Encryption requires high-entropy cryptographic keys, but users are prone

[11] If the Enclave is implemented using cryptographic techniques such as FWE, a unique Enclave can be shipped along with the files themselves. If the user employs a hardware token, the necessary key material can be delivered to the user's Enclave when the files are created or provisioned onto the user's device.

[12] To provide confidentiality of file accesses, the enclave may encrypt the log entry under the public key of some auditing party.

[13] Here we require the encryption scheme to be CCA-secure.

to lose or forget high-entropy passwords. To address this dilemma, manufacturers are turning to *trusted hardware*, including on-device cryptographic co-processors (Apple Computer, 2016). trusted enclaves (ARM Consortium, 2017), and cloud-based HSMs (LastPass, 2017; Krstić, 2016) for backup data. A user authenticates with a relatively weak passcode such as a PIN and the hardware will release a strong encryption key. To prevent brute force attacks, this *stateful* hardware must throttle or limit the number of login attempts.[14]

Enclave-Ledger Interaction provides an alternative mechanism for limiting the number of guessing attempts on password-based encryption systems. A manufacturer can employ an inexpensive stateless hardware token to host a simple enclave, with an internal (possible hard-wired) secret key K. In the initial step, the enclave takes in a password uses the random coins to produce a master encryption key k_{enc} that it outputs to the user. The Enclave is constructed to release k_{enc} only when it is given the proper passcode *and* the step counter is below some limit. Note that if the host restarts the execution, this simply re-runs the setup step which will generate a new key unrelated to the original. Rate limiting can be accomplished if the ledger has some approximation of a clock, like number of blocks between login attempts in Bitcoin. In practice the decryption process in such a system can be fairly time consuming if the ledger has significant lag. This system may be useful for low frequency applications such as recovering encrypted backups or emergency password recovery.

[14]This approach led to the famous showdown between Apple and the FBI in the Spring of 2016. The device in question used a 4-character PIN, and was defeated in a laboratory using a state rewinding attack, and in practice using an estimated $1 million software vulnerability(Paletta, 2015; Weaver, 2015).

3.5.4 Paid Decryption and Ransomware

ELI can also be used to condition program execution on *payments* made on an appropriate payment ledger such as Bitcoin or Ethereum. Because in these systems payment transactions are essentially just transactions written to a public ledger, the program P can take as input a public payment transaction and condition program execution on existence of this transaction. This feature enables pay-per-use software with no central payment server. Not all of the

Algorithm 5: Ransomware $P_{ransomware}$

Data: Input: I_i , S_i; Randomness r_i;
Parse $(K, \mathsf{R}, pk) \leftarrow \mathsf{S}_i$
if $\mathsf{S}_i = \varepsilon$ **then** `// Generate Key, Set Ransom`
  Parse $(\mathsf{R}, pk) \leftarrow \mathsf{I}_i$
  $K \leftarrow \mathsf{KDF}(r_i)$
  output $\mathsf{S}_{i+1} \leftarrow (K, \mathsf{R}, pk)$
else `// Release Key on Payment`
  Parse $(t, \sigma) \leftarrow \mathsf{I}_i$
  if $(\mathsf{BlockchainVerify}(t, \sigma) = 1)$ **then**
    if $(t.amount > \mathsf{R}$ *and* $t.target = pk)$ **then**
      output $\mathsf{O}_i = K$
  output $\mathsf{O}_i = \perp$

applications of this primitive are constructive. The ability to condition software execution on payments may enable new types of destructive application such as ransomware (Zetter, 2016). In current ransomware, the centralized system that deliver keys represent a weak point in the ransomware ecosystem. Those systems exposes ransomware operators to tracing (Technology.org, 2016). As a result, some operators have fled without delivering key material, as in the famous WannaCry outbreak (Kan, 2017).

In the remainder of this section we consider a potential *destructive* application of the ELI paradigm: the development of *autonomous* ransomware that guarantees decryption without the need for online C&C. We refer to this malware as autonomous because once an infection has occurred it requires no further interaction with the malware operators, who can simply collect payments issued to a Bitcoin (or other cryptocurrency) address.

In this application, the malware portion of the ransomware samples an encryption key $K \in \{0,1\}^{\ell}$ and installs this value along with the attackers public address within a Enclave. The Enclave will only release this encryption key if it is fed a validating blockchain fragment containing a transaction paying sufficient currency to the attacker's address. Algorithm 5 presents a simple example of the functionality.

We note that the Enclave may be implemented using trusted execution technology that is becoming available in commercial devices, *e.g.,* an Intel SGX enclave, or an ARM TrustZone trustlet. Thus, autonomous ransomware should be considered a threat today – and should be considered in the threat modeling of trusted execution systems. Even if the methods employed for securing these trusted execution technologies are robust, autonomous ransomware can be realized with software-only cryptographic obfuscation techniques, if such technology becomes practical(Lewi et al., 2016).

This application can be extended by allowing a ransomware instance to prove to a skeptical victim that it contains the true decryption key without allowing the victim to regain all their files. The victim and the ransomware can together select a random file on the disk to decrypt, showing the proper

key is embedded. Additionally, the number of such files that can be decrypted can be limited using similar methodology as in Section 3.5.2.

3.6 Realizing the Enclave

In Section 2.1 we discussed how the ledger interface we require can be realized in practice. We now proceed to consider realizations of the enclave

3.6.1 Realizing the Enclave

Trusted cryptographic co-processors The simplest approach to implement the enclave is using a secure hardware or trusted execution environment such as Intel's SGX(*Intel Software Guard Extensions (Intel SGX)* 2018), ARM Trustzone (ARM Consortium, 2017), or AMD SEV (Advanced Microchip Devices, 2018). When implemented using these platforms, our techniques can be used immediately for applications such as logging, fair encryption and ransomware.

While these environments provide some degree of hardware-supported immutable statekeeping, this support is surprisingly limited. For example, Intel SGX-enabled processors provide approximately 200 monotonic counters to be shared across all enclaves. On shared systems these counters could be maliciously reserved by enclaves such that they are no longer available to new software. Finally, these counters do not operate across enclaves operating on different machines, as in the smart contract setting.

Many simpler computing devices such as smart cards lack any secure

means of keeping state. In our model, even extremely lightweight ASICs and FPGAs could be used to implement the enclave for stateful applications using our ELI constructions. Along these lines, Nayak *et al.* (Nayak et al., 2017) recently showed how to build trusted non-interactive Turing Machines from minimal *stateless* trusted hardware. Such techniques open the way for the construction of arbitrary enclave functionalities on relatively inexpensive hardware.

Remark. Several recent attacks against trusted co-processors, particularly Intel SGX (Van Bulck et al., 2018) highlight the possibility that an enclave breach could reveal the key K. These attacks would have catastrophic implications for our protocol. We note that there are several potential mitigations for these attacks. For example, we recommend that an enclave should not directly expose the key K to a given program, but should instead *derive* a separate key for each program P in case the program contains a vulnerability. Similarly, we emphasize that even in the event of key leakage, industrial systems may be able to renew security through *e.g.*, a microcode update, which will allow the system to derive a new key K from some well-protected internal secret (as Intel did in response to the Foreshadow attack on SGX). Finally, to ensure that a processor is using the most recent microcode, the microcode maintainer can list the most recent microcode hash on the ledger and an ELI "bootloader" could use ELI to enforce that the current microcode is up to date. We leave exploration of these ideas to future work.

Software-Only Options A natural software-only equivalent of the enclave is to use pure-software techniques such as virtualization, or cryptographic

program obfuscation (Barak et al., 2001). While software techniques may be capable of hiding secrets from an adversarial user during execution, interactive multi-step obfuscated functionalities are *implicitly* vulnerable to being run on old state. Unfortunately, there are many negative results in the area of program obfuscation (Barak et al., 2001), and current primitives are not yet practical enough for real-world use (Lewi et al., 2016). However, for specific functionalities this option may be feasible: for example, Choudhuri *et al.* (Choudhuri et al., 2017) and Jager *et al.* (Liu et al., 2018) describe protocols based on the related Witness Encryption primitive.

3.7 Prototype Implementation

To validate our approach we implemented our ELI construction using Intel SGX (*Intel Software Guard Extensions (Intel SGX)* 2018; McKeen et al., 2013; Johnson et al., 2016; Anati et al., 2013; B, 2016; Rao, 2016; Intel Corporation, 2016)to implement the enclave and the Bitcoin blockchain to implement the ledger. We embedded a lightweight Javascript engine called Duktape (*Duktape.org* 2018) into our enclave, as similar projects have done in the past (Milutinovic et al., 2017). Source code can be found at `https://github.com/JHU-ARC/state_for_the_stateless/`.

The host application communicates with a local Bitcoin node via RPC to receive blockchain (ledger) fragments for delivery to the enclave, and to sends transaction when requested by the enclave. The enclave requires an independent (partial) Bitcoin implementation to verify proof-of-work tags used as ledger authenticators. We based this on the C++ SGX-Bitcoin implementation

Computation Section	Running Time	Percentage
Bitcoin Operations	**7764 μs**	**100%**
Proof Preparation	7094 μs	92.8%
Proof Verification	550 μs	7.2%
Protocol Operations	**2006 μs**	**100%**
Ciphertext Decryption	4 μs	0.2%
Javascript Invocation	1920 μs	95.7%
Ciphertext Encryption	82 μs	4.0%
SGX Overhead	**1153348 μs**	**100%**
Enclave Initialization	1153308 μs	100.0%
Ecall Entry and Exit	40 μs	0.0%

Figure 3.3: Measured computation overhead for different elements of our ELI experiment using a simple string concatenation program P. Because SGX does not support internal time calls, these times were measured by the application code. The table above shows averaged results over 100 runs on a local Bitcoin regtest network

in the Obscuro project (Tran et al., 2017).

At startup, the host application loads the Javascript program from a file, initializes the protocol values as in Algorithms 1, 2 and 3. and launches the SGX enclave. At first initialization the enclave generates a random, long term, master key K, which can be sealed to the processor using SGX's data sealing interface, protecting the key from power fluctuations. In each iteration of the protocol, the untrusted application code prompts the user for the next desired input. It then generates a transaction T using `bitcoin-tx` RPC. The first "input" $T.\mathsf{vin}[0]$ is set to be an unspent transaction in the local wallet. The first "output" $T.\mathsf{vout}[0]$ spends the majority of the input transaction to a new address belonging to the local wallet. The second output $T.\mathsf{vout}[1]$ embeds $\mathsf{SHA256}(i\|\mathsf{I}_i\|\mathcal{S}_i\|P\|\mathsf{CID}\|r_i)$ in an `OP_RETURN` script. The third output $T.\mathsf{vout}[2]$ embeds the public output Pub emitted by the previous step. This transaction is signed by a secret key in the local wallet and submitted for confirmation.

The host application now monitors the blockchain until T has been confirmed by 6 blocks.[15] The host then sets (1) $\mathsf{post}_i.\mathsf{Data} \leftarrow T.\mathsf{vout}$, (2) $\mathsf{post}_i.\mathsf{PrevHash} \leftarrow T.\mathsf{vin}[0].\mathit{Hash}$, (3) $\mathsf{post}_i.\mathsf{CID} \leftarrow$ chain of transactions from T back to the transaction with hash $\mathsf{post}_0.\mathsf{PrevHash}$, (4) $\mathsf{post}_i.\mathsf{Hash} \leftarrow T.\mathit{Hash}$, and (5) $\pi_{\mathsf{publish},i} \leftarrow 6$ blocks confirming T.

The host then submits $(\mathsf{post}_i, \pi_{\mathsf{publish},i})$ to the enclave which then performs the following checks: (1) verifies that $\pi_{\mathsf{publish},i}$ is valid and has sufficiently high block difficulty (2) the blocks in $\pi_{\mathsf{publish},i}$ are consecutive (3) $T.\mathsf{vout}[0], T.\mathsf{vout}[1]$ embed the correct data and (4) the transactions in $\mathsf{post}_i.\mathsf{CID}$ are well formatted.

If $i = 0$ and there is no input state, the enclave generates a zero initial state. Otherwise it generates the decryption key as described in the protocol using C-MAC to implement the PRF. The state along with the inputs and random coins are passed to the Javascript interpreter. All hashes computed in the enclave are computed using SHA256. One note is that instead of hashing all of CID into the ciphertext, we include only $\mathsf{post}_0.\mathsf{PrevHash}$, which keeps CID constant throughout the rounds.

Implementation Limitations. We chose to use Intel SGX to implement our enclave because it is a widely accepted, secure execution environment. However, SGX is significantly more powerful than the enclaves we model, including access to trusted time and monotonic counters. Although we use SGX, we do not leverage any of these additional features to make sure our implementation matches our model. Our Bitcoin implementation of the ledger is slow and

[15] In general, six blocks is considered sufficiently safe for normal Bitcoin payment operations; however the number of confirmations blocks can be tweaked as an implementation parameter.

would likely not be suitable for production release. Finally, we implement our applications in Javascript so the Javascript virtual machine will insulate the enclave code from host tampering.

Measurements To avoid spending significant money on the Bitcoin main network, we tested our implementation on a private regression regtest. This also allows us to control the rate at which blocks are mined. The most time-consuming portion of an implementation using the mainnet or testnet is waiting for blocks to be confirmed; blocks on the main bitcoin network take an average of 10 minutes to be mined, or an average of 70 minutes to mine a block and its 6 confirmation blocks. If an application requires faster execution, alternative blockchains can be used, such as Litecoin (2.5 minutes per block) or Ethereum (approximately 10-19 seconds).

Our experiments used a simple string concatenation program P. For our experiments we measured three specific operations: (1) the execution time of the Bitcoin operations (on the host, enclave and regtest network, (2) ELI protocol execution time, (3) the time overhead imposed by Intel SGX operations. Figure 3.3 shows the running times of these parts of our implementation. It is worth noting that SGX does not provide access to a time interface, and there is no way for an SGX enclave to get trustworthy time from the operating system. The times in Figure 3.3 were measured from the application code.

Discussion. Note that initializing an SGX enclave is a one-time cost that must be paid when the enclave is first loaded into memory. It is a comparatively expensive operation because the SGX driver must verify the code integrity and perform other bookkeeping operations. An additional computationally

expensive operation is obtaining the proof-of-publication to be delivered to the enclave. This process relies on `bitcoin-cli` to retrieve the proper blocks, which can be slow depending on the status of the `bitcoind` daemon. We note that these tests were run using the regression blockchain regtest, and retrieving blocks from testnet or mainnet may produce different results.

3.8 Conclusion

In this work we considered the problem of constructing secure stateful computation from limited computing devices. This work leaves several open questions. First, while we discussed the possibility of using cryptographic obfuscation schemes to construct the enclave, we did not evaluate the specific assumptions and capabilities of such a system. Additionally, there may be other capabilities that the enclave-ledger combination can provide that are not realized by this work. Finally, while we discussed a number of applications of the ELI primitive, we believe that there may be many other uses for these systems.

Enclave Ledger Interaction References

Nakamoto, S. (2008). "Bitcoin: A peer-to-peer electronic cash system, 2008". In: URL: http://www.bitcoin.org/bitcoin.pdf.

Handshake (2018). *Handshake protocol*. Available at https://handshake.org/.

Namecoin (2016). https://namecoin.org/. URL: https://namecoin.org/.

The Ethereum Project (2018). https://www.ethereum.org/.

Certificate Transparency (2018). Available at https://www.certificate-transparency.org.

Poulsen, Kevin (2001). "DirecTV attacks hacked smart cards". In: *The Register*.

Apple Computer (2016). *iOS Security: iOS 9.3 or later*. Available at https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

Intel Software Guard Extensions (Intel SGX) (2018). https://software.intel.com/en-us/sgx.

ARM Consortium (2017). *ARM Trustzone*. Available at https://www.arm.com/products/security-on-arm/trustzone.

Advanced Microchip Devices (2018). Available at https://developer.amd.com/sev/.

Garg, Sanjam, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters (2013). *Candidate Indistinguishability Obfuscation and Functional Encryption for all circuits*. Cryptology ePrint Archive, Report 2013/451.

Döttling, N., T Mie, J. Müller-Quade, and T. Nilges (2011). "Basing Obfuscation on Simple Tamper-Proof Hardware Assumptions". In: *TCC '11*. Springer.

Nayak, Kartik, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya Lokam, Elaine Shi, and Vipul Goyal (2017). "HOP: Hardware makes Obfuscation Practical". In: *NDSS '17*.

Lewi, Kevin, Alex J. Malozemoff, Daniel Apon, Brent Carmer, Adam Foltzer, Daniel Wagner, David W. Archer, Dan Boneh, Jonathan Katz, and Mariana Raykova (2016). "5Gen: A Framework for Prototyping Applications Using

Multilinear Maps and Matrix Branching Programs". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, pp. 981–992. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978314. URL: http://doi.acm.org/10.1145/2976749.2978314.

Bellare, Mihir, Marc Fischlin, Shafi Goldwasser, and Silvio Micali (2001). "Identification Protocols Secure against Reset Attacks". In: *EUROCRYPT '01*. Ed. by Birgit Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 495–511. ISBN: 978-3-540-44987-4. DOI: 10.1007/3-540-44987-6_30. URL: https://doi.org/10.1007/3-540-44987-6_30.

TPM Reset Attack (2018). Available at http://www.cs.dartmouth.edu/~pkilab/sparks/.

Giller, Brett (2015). "Implementing Practical Electrical Glitching Attacks". In: *BlackHat '15*.

Skorobogatov, Sergei (2016). "The bumpy road towards iPhone 5c NAND mirroring". In: *CoRR* abs/1609.04327. URL: http://arxiv.org/abs/1609.04327.

Parno, Bryan, Jacob R. Lorch, John R. Douceur, James Mickens, and Jonathan M. McCune (2011). "Memoir: Practical State Continuity for Protected Modules". In: *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. SP '11. Washington, DC, USA: IEEE Computer Society, pp. 379–394. ISBN: 978-0-7695-4402-1. DOI: 10.1109/SP.2011.38. URL: https://doi.org/10.1109/SP.2011.38.

Skorobogatov, Sergei P. and Ross J. Anderson (2003). "Optical Fault Induction Attacks". In: *CHES '02*. Ed. by Burton S. Kaliski, çetin K. Koç, and Christof Paar. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 2–12. ISBN: 978-3-540-36400-9. DOI: 10.1007/3-540-36400-5_2. URL: https://doi.org/10.1007/3-540-36400-5_2.

Kauer, Bernhard (2007). "OSLO: Improving the Security of Trusted Computing". In: *Usenix '07*. Boston, MA: USENIX Association. ISBN: 111-333-5555-77-9. URL: http://dl.acm.org/citation.cfm?id=1362903.1362919.

Hyperledger (2017). *Hyperledger Architecture, Volume 1*. Available at https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

Amazon (2018). *AWS Step Functions*. Available at https://aws.amazon.com/step-functions/.

Google Inc. (2018). *Google Cloud Functions*. Available at https://cloud.google.com/functions/.

Matetic, Sinisa, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun (2017). *ROTE: Rollback Protection for Trusted Execution*. Cryptology ePrint Archive, Report 2017/048.

Canetti, Ran, Oded Goldreich, Shafi Goldwasser, and Silvio Micali (2000). "Resettable Zero-knowledge (Extended Abstract)". In: *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*. STOC '00. Portland, Oregon, USA: ACM, pp. 235–244. ISBN: 1-58113-184-4. DOI: 10.1145/335305.335334. URL: http://doi.acm.org/10.1145/335305.335334.

Goyal, Rishab and Vipul Goyal (2017). *Overcoming Cryptographic Impossibility Results using Blockchains*. Cryptology ePrint Archive, Report 2017/935.

Choudhuri, Arka Rai, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers (2017). "Fairness in an Unfair World: Fair Multiparty Computation from public Bulletin Boards". In: *CCS '17*.

Bowman, M., A. Miele, M. Steiner, and B. Vavala (2018). "Private Data Objects: an Overview". In: *ArXiv e-prints*. arXiv: 1807.05686 [cs.CR].

Kosba, A., A. Miller, E. Shi, Z. Wen, and C. Papamanthou (2016). "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 839–858. DOI: 10.1109/SP.2016.55.

Juels, Ari, Ahmed Kosba, and Elaine Shi (2016). "The Ring of Gyges: Investigating the Future of Criminal Smart Contracts". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, pp. 283–295. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978362. URL: http://doi.acm.org/10.1145/2976749.2978362.

Zhang, Fan, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi (2016). "Town Crier: An Authenticated Data Feed for Smart Contracts". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, pp. 270–282. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978326. URL: http://doi.acm.org/10.1145/2976749.2978326.

Cheng, Raymond, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah M. Johnson, Ari Juels, Andrew Miller, and Dawn Song (2018). "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution". In: *CoRR* abs/1804.05141. arXiv: 1804.05141. URL: http://arxiv.org/abs/1804.05141.

Ethereum White Paper (2017). *Ethereum White Paper*. Available at https://github.com/ethereum/wiki/wiki/White-Paper.

Project, Android (2017). *Full-Disk Encryption*. Available at https://source.android.com/security/encryption/full-disk.html.

Krstić, Ivan (2016). *Behind the Scenes with iOS Security*. In BlackHat. Available at https://www.blackhat.com/docs/us-16/materials/us-16-Krstic.pdf.

Bonneau, Joseph (2012). "The Science of Guessing: Analyzing an Anonymized Corpus of 70 Million Passwords". In: *IEEE S&P (Oakland) '12*. SP '12. Washington, DC, USA: IEEE Computer Society, pp. 538–552. ISBN: 978-0-7695-4681-0. DOI: 10.1109/SP.2012.49. URL: http://dx.doi.org/10.1109/SP.2012.49.

Ur, Blase, Sean M. Segreti, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Saranga Komanduri, Darya Kurilova, Michelle L. Mazurek, William Melicher, and Richard Shay (2015). "Measuring Real-World Accuracies and Biases in Modeling Password Guessability". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, pp. 463–481. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ur.

Apple Computer (2017). *Answers to your questions about Apple and security*. Available at http://www.apple.com/customer-letter/answers/.

Sinegubko, Denis (2016). *Website Ransomware - CBT-Locker Goes Blockchain*. Sucuri Blog. Available at https://blog.sucuri.net/2016/04/website-ransomware-ctb-locker-goes-blockchain.html.

Rogaway, Phillip (2002). "Authenticated Encryption with Associated Data". In: *CCS '02*. ACM Press.

Intel Corporation (2018). *Hyperledger Sawtooth*. Available at http://hyperledger.org/projects/sawtooth.

Foundation, Apache (2018). *Hadoop Key Management Server (KMS) - Documentation Sets*. Available at https://hadoop.apache.org/docs/stable/hadoop-kms/index.html.

Android Project (2017). *File-Based Encryption for Android*. Available at https://source.android.com/security/encryption/file-based.

LastPass (2017). *How is LastPass secure and how does it encrypt/decrypt my data safely?* Available at https://lastpass.com/support.php?cmd=showfaq&id=6926.

Paletta, Damian (2015). "FBI Chief Punches Back on Encryption". en-US. In: *Wall Street Journal*. URL: http://www.wsj.com/articles/fbi-chief-punches-back-on-encryption-1436217665.

Weaver, Nicholas (2015). "iPhones, the FBI, and Going Dark". In: *Lawfare Blog*. URL: https://www.lawfareblog.com/iphones-fbi-and-going-dark.

Zetter, Kim (2016). *Why Hospitals are the Perfect Targets for Ransomware*. Available at https://www.wired.com/2016/03/ransomware-why-hospitals-are-the-perfect-targets/.

Technology.org (2016). *Ransomware Authors Arrest Cases*. Available at http://www.technology.org/2016/11/21/ransomware-authors-arrest-cases/.

Kan, Michael (2017). "Paying the WannaCry ransom will probably get you nothing. Here's why." In: *PCWorld*.

Van Bulck, Jo, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx (2018). "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution". In: *Proceedings of the 27th USENIX Security Symposium*. USENIX Association.

Barak, Boaz, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang (2001). *On the (Im)possibility of Obfuscating Programs*. Cryptology ePrint Archive, Report 2001/069.

Liu, Jia, Tibor Jager, Saqib A. Kakvi, and Bogdan Warinschi (2018). "How to build time-lock encryption". In: *Designs, Codes and Cryptography* 86.11, pp. 2549–2586. ISSN: 1573-7586. DOI: 10.1007/s10623-018-0461-x. URL: https://doi.org/10.1007/s10623-018-0461-x.

McKeen, Frank, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar (2013). "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA*, p. 10.

Johnson, Simon, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen (2016). *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*.

Anati, Ittai, Shay Gueron, Simon Johnson, and Vincent Scarlata (2013). "Innovative technology for CPU based Attestation and Sealing". In: *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. Vol. 13.

B, Alexander (2016). *Introduction to Intel SGX Sealing*. Available at https://software.intel.com/en-us/blogs/2016/05/04/introduction-to-intel-sgx-sealing.

Rao, Dinesh (2016). *Intel SGX Product Licensing*. Available at https://software.intel.com/en-us/articles/intel-sgx-product-licensing.

Intel Corporation (2016). *Product Licensing FAQ*. Available at https://software.intel.com/en-us/sgx/product-license-faq.

Duktape.org (2018). Available at http://duktape.org.

Milutinovic, Mitar, Warren He, Howard Wu, and Maxinder Kanwal (2017). *Proof of Luck: an Efficient Blockchain Consensus Protocol*. Cryptology ePrint Archive, Report 2017/249.

Tran, Muoi, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena (2017). *Obscuro: A Bitcoin Mixer using Trusted Execution Environments*. Cryptology ePrint Archive, Report 2017/974.

Chapter 4

Achieving Fairness in MPC

4.1 Introduction

Secure multiparty computation (MPC) allows a collection of mutually distrusting parties to jointly compute a function on their private inputs while revealing nothing beyond the function output. Since its conception three decades ago (Yao, 1982; Goldreich, Micali, and Wigderson, 1987), MPC has found wide applicability to important tasks such as electronic auctions, voting, valuation of assets, and privacy-preserving data mining.

Fairness. Over the years, several security definitions for MPC have been studied. One natural and desirable definition for MPC stipulates that either all parties receive the protocol output or no party does. This is referred to as *fair* MPC.

The notion of fairness is very important (and necessary) in applications such as auctions and contract signing. For example, if Alice is the first to learn she did not win an auction, she may abort, claim a network failure, and try again with a new bid that just exceeds the previous winning bid. More

generally when the "value" of the function output may be enhanced by an information asymmetry, e.g., if Alice is better off exclusively knowing the true value of a financial asset than all parties knowing it, fairness is an issue.

In a seminal work, Cleve (Cleve, 1986) proved that fair MPC is impossible to realize for general functions when a majority of the parties are dishonest. This result even holds when the parties have access to a trusted setup such as a common reference string.

The pursuit of fairness. In light of Cleve's impossibility result, a vast amount of research effort has been dedicated towards the study of mitigations to the fairness problem. In particular, two prominent lines of research have emerged over the years. The first research direction considers the problem of achieving fairness in the standard model for a *restricted* classes of functions (Gordon et al., 2008; Gordon and Katz, 2009; Asharov, Lindell, and Rabin, 2013; Asharov, 2014; Asharov et al., 2015).

The second research direction studies fairness for *general* functions by augmenting the computation model and/or by relaxing the definition of fairness. The prominent examples in this direction range from using a trusted party to restore fairness (Cachin and Camenisch, 2000), to weaker models where the honest parties can recover the output at computational cost or time at most Δ-times that of the adversary (Even, Goldreich, and Lempel, 1982; Beaver and Goldwasser, 1989; Goldwasser and Levin, 1991; Pinkas, 2003; Garay et al., 2006; Pass, Shi, and Tramèr, 2017) (where Δ is a constant), to penalizing aborting parties monetarily (Andrychowicz et al., 2014; Bentov and Kumaresan, 2014; Kumaresan, Moran, and Bentov, 2015; Kumaresan and

Bentov, 2016). (See Section 4.2 for a more elaborate discussion.)

While these mitigations are helpful, they fall short of solving the problem in many circumstances. In particular, they either require appointing trusted parties for very specific tasks (related to the protocol) that can be hard to find, or require that the parties' possess precise estimates of the adversary's resources and incentives. If the adversary values exclusive knowledge of the output very highly, it may not be practical to have a large enough computational differential or penalty to deter aborts. Indeed, in many cases it may be impossible value the MPC output at all.

4.1.1 Our Results

In this work, we construct theoretical and practical fair MPC protocols for general functions in the bulletin board model. We, in fact, provide general transformations from any (possibly unfair) n-party MPC protocol that supports $t < n$ corruptions to a fair MPC protocol secure against the same number of corruptions. Crucially, the assumptions used in our transformations affect fairness only: the correctness and privacy properties of the underlying MPC scheme are completely preserved even if the assumptions were not to hold.

I. Fair MPC from Witness Encryption. Our first contribution is a fair MPC protocol in the bulletin board model assuming the existence of witness encryption (WE) (Garg et al., 2013) and injective one-way functions. In order to rely on the standard security of WE, we require the bulletin board's proof of publish to be implemented via unique signatures (Goldwasser and Ostrovsky, 1993; Lysyanskaya, 2002). If the bulletin board is implemented via standard

signatures (e.g., in Google Transparency Certificates) or proofs of stake (e.g., in Etherium), then we require the stronger assumption of extractable witness encryption (Boyle, Chung, and Pass, 2014).

Candidate constructions of WE for NP (Garg et al., 2013; Gentry, Lewko, and Waters, 2014) are known from multilinear maps (Garg, Gentry, and Halevi, 2013). Since present constructions (Garg, Gentry, and Halevi, 2013; Coron, Lepoint, and Tibouchi, 2013; Gentry, Gorbunov, and Halevi, 2015; Coron, Lepoint, and Tibouchi, 2015) of multilinear maps are quite inefficient, we view our first construction as a feasibility result. We note, however, that our construction requires WE for a specific NP language for which constructing efficient schemes from simpler assumptions might be easier. Indeed, a fascinating open question for future work is whether WE for the specific language used in our constructions can be implemented from existing constructions for the related notion of hash proof systems (Cramer and Shoup, 1998).

II. Fair MPC from Secure Processors. Our second contribution is a fair MPC protocol in the bulletin board where all the parties have access to secure processors. In fact, Cleve's impossibility result holds even in the presence of secure processors, and was proved recently in (Pass, Shi, and Tramèr, 2017). For concreteness, we work with Intel SGX as a secure processor, following the formalization of (Pass, Shi, and Tramèr, 2017). For this result, we only need standard cryptographic assumptions such as secret-key authenticated encryption and signatures. We provide an implementation of this protocol in Section 4.6.

Comparison with recent works. Recently, (Andrychowicz et al., 2014; Bentov

and Kumaresan, 2014) showed how block-chain based decentralized cryptocurrencies such as Bitcoin can be used to achieve a notion of *fairness with penalties* where aborting parties are forced to pay a pre-agreed financial penalty. We note that while we also use blockchain based bulletin boards in our work, our end result is quite different in that we achieve the standard notion of fairness – either all parties get the output or none do.

Very recently, (Pass, Shi, and Tramèr, 2017) studied fairness in the model where each party has access to a secure hardware equipped with secure clock. They achieve a notion of Δ-fairness which guarantees that if an aborting adversary can learn the output in time T, then the honest party can also learn the output in time $\Delta \cdot T$ for $\Delta = 2$. A disadvantage of this model is that T is controlled by the adversary, who can set it arbitrarily to create large delay (e.g., in the order of several minutes or hours) between the times when it gets the output and when the honest party does.

We note that while we also use secure hardware for our second result, we do not require them to implement secure clocks.[1] More importantly, we achieve the standard notion of fairness.

4.1.2 Technical Overview

We now describe the main ideas used in our constructions. For simplicity of exposition, we restrict this discussion to the two-party case. It is easy to generalize the ideas presented below to the multiparty case.

[1] In the specific case where the bulletin board is implemented using a proof of work blockchain, we can use secure clocks to achieve stronger security guarantees. This is unnecessary when the bulletin board uses signatures. We discuss this further in Section 4.6.

Starting Ideas. Our starting idea is to run an unfair MPC protocol to compute an encryption of the function output as opposed to computing it in the clear. We then design a special decryption procedure such that either no party is able to perform the decryption or both parties can. In other words, we reduce the fairness problem in MPC to the problem of fair decryption.

At first, it may seem that we haven't made any progress because it is unclear why fair decryption would be any easier than achieving fairness for general functions. Indeed, fair decryption was shown to be a *complete* functionality for fair MPC in (Gordon et al., 2010).

Our *key insight is that a public bulletin board can be used to implement a fair decryption protocol for a witness encryption scheme*. We elaborate on this idea below.

Fairness from Witness Encryption. A witness encryption scheme for a language L can be used to encrypt a message m with a statement x in such a manner that the resulting ciphertext can only be decrypted using a witness w for x. We now explain how we use witness encryption to implement our fair MPC protocol.

In order to securely compute a function f with complete fairness, the parties first run a standard (possibly unfair) MPC protocol to compute a randomized function that takes the private inputs say (y_1, y_2) of the parties and returns a witness encryption ciphertext CT of the desired output $\mathcal{F}(y_1, y_2)$. The statement x associated with CT is set to be such that a valid witness for x corresponds to the proof of posting a "release token" α (to be determined later) on the bulletin board.

The only way for any party to obtain such a witness is to post α on the bulletin board and obtain the corresponding proof of posting σ. However, in doing so, the pair (α, σ) is made public, and therefore, anyone can obtain it. Thus, if a malicious adversary learns the witness for decrypting CT, then so can the honest party since it can simply read the public bulletin board. This mechanism puts the honest party and the adversary on equal footing and resolves the fairness problem.

While the above constitutes the core idea behind our work, we run into several technical issues in implementing this idea. We discuss these next, together with the solutions.

Issue #1: Setting the release token. An immediate issue with implementing the above idea is that we cannot set the release token α to be an a priori fixed value that is known to the adversary. Indeed, if this is the case, then the adversary can simply abort during the execution of the unfair MPC protocol so that it learns the ciphertext CT, but the honest party does not. Now, even if the honest party can obtain (α, σ) once the adversary has posted it on the bulletin board, it cannot learn the output $\mathcal{F}(y_1, y_2)$ since it does not have CT to decrypt.

To address this issue, we set α to be a pair of random values (α_1, α_2) where α_i is chosen by the i-th party. During the initial MPC phase, each party uses α_i as an additional input such that the output of the MPC is (β, CT) where $\beta_i = f(\alpha_i)$ for some one-way function f and $\beta = (\beta_1, \beta_2)$. Now, even given (β, CT), the value α is not completely known to the adversary. Therefore, if it aborts prematurely, then the honest party aborts as well, knowing that the

adversary would not be able to recover the output.

On the other hand, if the first phase is successfully completed, then the parties execute a second phase where each party i simply sends over α_i to the other party. Of course, the adversary may abort in this phase after learning α. However, in order to decrypt CT, it will have to post α on the bulletin board which means the honest party would learn it as well. This restores the balance between the honest party and the adversary.

Issue #2: Security of WE. The standard definition of witness encryption only guarantees semantic security for a ciphertext CT if the statement x associated with it is *false*. In our case, the statement is always true. The only way to argue security in this case is to use a stronger notion of extractable witness encryption (Boyle, Chung, and Pass, 2014) which guarantees that for any statement x, if an adversary can distinguish between witness encryption of m from an encryption of $m' \neq m$, then one can efficiently recover from that adversary a witness w for x. Now, if the witness w is computationally hard to find, then we can get a contradiction.

It was shown in (Boyle, Chung, and Pass, 2014) that for languages with statements that have only polynomially many witnesses, the standard definition of WE implies the stronger definition of extractable WE. We note that if we set f to be an *injective* one-way function and implement the proof of posting on the bulletin board via unique signatures (Goldwasser and Ostrovsky, 1993; Lysyanskaya, 2002), then we can bound the number of valid witnesses. In this case, we can rely on the standard definition of WE.

Issue #3: Rewinding. We run into yet another issue while arguing security of

the above construction. Recall that in order to prove security of a fair MPC protocol, we must construct a simulator who can "force" the correct output on the real adversary, provided that the adversary did not abort prematurely. In our protocol, the only opportunity for the simulator to "program" the output is inside the ciphertext CT computed during the initial MPC phase. However, this point in our overall protocol is "too early" for the simulator to determine with enough confidence whether the real adversary is going to later abort or not. If the simulator's decision to program the output turns out to be wrong, then it would immediately lead to a distinguisher between the outputs of the real and ideal experiments.

To deal with this issue, we use a rewinding strategy previously used in (Goldreich and Kahan, 1996a; Gordon et al., 2010; Gordon, 2010) to determine the aborting probability of the adversary with enough accuracy, while still ensuring (expected) polynomial running time for the simulator. In order to ensure indistinguishability of the adversary's view in the real and ideal experiments, we allow the simulator to also rewind the bulletin board to a previous state, as and when necessary. Indeed, without this capability, the simulator cannot prevent an adversary from "detecting rewinding" by continuously posting on the bulletin board. A consequence of this is that we must model the bulletin board as a "local" functionality as opposed to a "global" functionality (Canetti et al., 2007; Canetti, Jain, and Scafuro, 2014). Furthermore, since our simulator performs rewinding, we only achieve stand-alone security.

Fairness from Secure Hardware. Roughly, the main idea in our second protocol is to replace the witness encryption in the plain model with a secure hardware that implements (essentially) the same functionality as witness encryption. We require that each party is equipped with such a secure hardware (e.g., Intel SGX). While much of the details in this protocol are similar to the previous one, there are some key differences. We explain them below.

Once the parties have "installed" an appropriate program P (discussed below) in their own local secure hardware and attestation of the same is successfully performed by everyone, they run (as in the previous protocol) an execution of a standard MPC protocol to compute an encryption CT of the desired output. Unlike the previous scheme where CT was computed using witness encryption, here we use a regular secret-key encryption scheme. The secret key K used for encryption is secret-shared amongst the parties who use their respective shares as additional inputs to the MPC. The key K is also loaded in each party's secure hardware, and is in fact computed by the secure hardware devices during an initial key-exchange phase.

As in the previous protocol, we require that the ciphertext CT can only be decrypted if the release token α has been posted on the bulletin board. The program P loaded in each party's secure hardware implements such a conditional decryption mechanism. Specifically, upon receiving a ciphertext CT, a release token α and a corresponding proof of posting σ, the program P verifies the validity of α and σ. If the verification succeeds, then it decrypts CT and returns the output; otherwise it returns $\perp$.

We remark upon two security issues: first, in order to prevent malleability

attacks, we require that an authenticated encryption scheme is used in order to compute CT. Further, to prevent an adversary from performing a related key attack (by changing its input key share in the MPC), we require that the secure hardware also provide commitments C_i of each key share K_i to all the parties upon generation of K. A party i is required to input the decommitment to C_i in the MPC protocol, and the MPC functionality checks that all the input key shares are valid by verifying the decommitment information.

Second, for this protocol, we can completely dispense with rewinding and instead construct a black-box, non-rewinding simulator. This is because the use of secure hardware allows the simulator to "program" the output at the very end, when the adversary makes a decryption query to its secure hardware.[2] Indeed, in the secure hardware model, the simulator has the ability to observe (and modify) the queries made by the adversary to its secure hardware. This means that when the adversary makes a final decryption query, the simulator can check if it is valid. If this is the case, then it queries the trusted party to obtain the function output. At this point, the simulator sends a "fake" decryption query to the secure hardware that already contains the desired output. Upon receiving this query, the secure hardware returns the programmed output to the adversary. We note that this programming technique for secure hardware was recently used in (Pass, Shi, and Tramèr, 2017).

Because of the above modifications, in this protocol, we can model the

[2]We also use an MPC in the common random string (CRS) model (e.g., (Canetti et al., 2002)) to implement the first phase of the protocol. By using the CRS trapdoor, the simulator for this phase can avoid any rewinding of the adversary.

bulletin board as a global functionality. In this manuscript, however, we do not prove UC security of our protocol and leave it for future work.

Realizing the Bulletin Board. Our constructions assume a public bulletin board that is capable of producing an unforgeable proof that a string has been published to the bulletin board. Such bulletin boards can easily be constructed practice if one is willing to instantiate the board using a single trusted party. While this seems a strong assumption, the advantage of this approach is that such systems already exist and have been widely deployed in practice for applications such as Certificate Transparency (*Certificate Transparency* 2017). Re-using them to achieve fairness in *arbitrary* MPC protocols requires no specific to the existing systems.

Alternatively, a bulletin board can be realized using a decentralized systems such as proof of stake blockchains (*e.g.*, (Kiayias et al., 2017)). These systems allow a quorum of honest users – who together possess a majority ownership "stake" in a cryptocurrency – to securely authenticate an append-only log using signatures. Finally, a weaker notion of security can be achieved using a proof of work blockchain. In the latter case, the "proof" of publication is not a cryptographically unforgeable signature, but rather the solution to a sequence of one or more computational puzzles which may be, in practice, prohibitively expensive for an attacker to forge.[3] We explore this approach in our experimental implementation, although we stress that this is merely an

[3]In practice, such proof of work blockchains provide a slightly weaker security that is related to Δ-fairness. An attacker, given enough time, may be able to forge the proof of work necessary to prove publication. However, in the trusted hardware setting we are able to mitigate this concern to some extent by requiring the attacker to provide a proof in a limited period of time, as judged by the hardware.

implementation detail. Our bulletin board could easily be replaced with one of the alternatives above.

Optimizations. We mention a few optimizations to the above protocols to improve efficiency. First, we can add an *optimistic decryption phase* in the above protocols that allows the parties to learn the output using a simple decryption process, without using the bulletin board, provided that all the parties are honest. Roughly, the MPC protocol executed in the first phase now additionally computes another encryption CT' of the function output, where CT' is implemented using a regular encryption scheme. The decryption key K' corresponding to CT' is secret-shared between the parties. Now, if the release-token exchange performed in the second phase is successful, then the parties execute a third phase (that we refer to as the optimistic decryption phase) where they exchange the key shares corresponding to K'. If all the parties are honest, then they all learn K' and use it to decrypt CT', without using the bulletin board. However, if one or more parties are adversarial and abort in this phase, then the honest parties can still post the release token α (that they learned in the second phase) on the bulletin board and then use the proof of posting to decrypt CT as before.

We remark that in order to avoid related key attacks by an adversary, we would need a slight modification to the above protocol where the MPC in the first phase outputs commitments to each key share K'_i to both the parties. During the optimistic decryption phase, each party must reveal the decommitment value together with K'_i. A party only accepts the key share as valid if the associated decommitment information is correct.

Finally, we note that the size of the release token $\alpha = (\alpha_1, \alpha_2)$ used in the above described protocols grows with the number of parties N. However, it is easy to make it independent of N by setting $\alpha = \oplus_i \alpha_i$ and using $\beta = f(\alpha)$ to verify the correctness of release token. An advantage of this modification is that the witness length for the witness encryption used in our construction, as well as the length of the string that is posted on the bulletin board becomes independent of the number of parties.

4.2 Related work

A large body of research work has addressed the problem of fairness in secure protocols over the years. Below, we provide a non-exhaustive summary of prior works. A more elaborate summary can be found, e.g., in (Bentov and Kumaresan, 2014).

Fairness in Standard Model. Assuming an honest majority of parties, fair MPC can be achieved in both computational (Goldreich, Micali, and Wigderson, 1987) and information-theoretic setting (Rabin and Ben-Or, 1989). Cleve (Cleve, 1986) proved the impossibility of MPC for general functions n the dishonest majority setting. Subsequently, an exciting sequence of works (Gordon et al., 2008; Gordon and Katz, 2009; Asharov, Lindell, and Rabin, 2013; Asharov, 2014; Asharov et al., 2015) have shown that complete fairness can still be achieved for a restricted class of functions. The works of (Gordon and Katz, 2010; Beimel et al., 2011; Alon and Omri, 2016) study the problem of partial fairness.

Optimistic Models. Starting from the early work of (Ben-Or et al., 1985), optimistic models for fair exchange have been studied in a long sequence of works (Asokan, Schunter, and Waidner, 1997; Asokan, Shoup, and Waidner, 1998; Garay, Jakobsson, and MacKenzie, 1999; Micali, 2003; Dodis, Lee, and Yum, 2007; Küpçü and Lysyanskaya, 2010). An optimistic model for fair two-party computation using a semi-trusted third party was studied in (Cachin and Camenisch, 2000; Kilinç and Küpçü, 2016).

Gradual Release Mechanisms. A different approach to fairness that avoids trusted third parties was considered in a long sequence of works (Boneh and Naor, 2000; Garay and Jakobsson, 2003; Garay and Pomerance, 2003; Pinkas, 2003), following the early works of (Even, Goldreich, and Lempel, 1982; Beaver and Goldwasser, 1990; Goldwasser and Levin, 1991). The protocols in these works employ a "gradual release" mechanism where the parties take turns to release their secrets in a bit-by-bit fashion. The intuitive security guarantee (formalized in (Garay et al., 2006)) is that even if an adversary aborts prematurely, the honest party can recover the output in time comparable to that of the adversary by investing equal (or more) computational effort.

Δ-Fairness. Very recently, (Pass, Shi, and Tramèr, 2017) considered a notion of Δ-fairness with the guarantee that if an adversary aborts, then the honest party can learn the output in time $\Delta \cdot T$, where T is the time in which the adversary would learn the output. They propose a fair two-party computation protocol with $(\Delta = 2)$-fairness assuming that all the parties have secure hardware equipped with secure clocks.

Fairness with Penalties. Recently, with the popularity of decentralized cryptocurrencies such as Bitcoin, a sequence of works (Andrychowicz et al., 2014; Bentov and Kumaresan, 2014; Kumaresan, Moran, and Bentov, 2015; Kumaresan and Bentov, 2016) have shown how to implement a fairness-with-penalties model for MPC where adversarial parties who prematurely abort are forced to pay financial fines. Prior works in similar spirit considered fairness with reputation systems (Asharov, Lindell, and Zarosim, 2013) and legally enforced fairness (Chen, Kudla, and Paterson, 2004; Lindell, 2009).

4.3 Fair MPC from Witness Encryption

Overview. We start by giving an overview of our protocol. Our protocol builds on an MPC protocol that achieves the weaker notion of security with abort, where the fairness condition is not required to hold. The initial phase constitutes of the parties using this unfair MPC protocol to compute a witness encryption ciphertext of the function value they wish to compute. To decrypt, a party must post messages of a specific form (referred to as "release tokens") on to the bulletin board which the bulletin board validates with an authentication tag. The idea then is that any party can use this posted information and authentication tag to decrypt the witness encryption ciphertext. The release token must include shares of all parties that are secret prior to the completion of this initial phase. These shares must also be easily verifiable. Our construction uses injective one-way functions, where the images of these shares are sent out during the initial phase.

The next phase, on completion of the initial phase, constitutes of parties

sending these secrets to every other party. Once a party releases its share, it must not abort until it is sure that the other parties cannot post to the bulletin board, and hence decrypt the message thereafter. Otherwise, the adversary on receiving the secret shares will wait for the honest parties to abort before posting to the bulletin board. This is resolved by parameterizing the protocol by a cut-off period which once elapsed, effectively ends the protocol. If there isn't a valid post to the bulletin board at this time, no party gets the output.

To argue security, we require each statement in the language corresponding to the witness encryption to have only a polynomial size witness set. To do so, we use an injective one-way function and a unique signature scheme. The witness for the statement are the pre-images of the values sent during the initial phase, and the corresponding tag from the bulletin board. This pair is unique for a given statement. But we need to incorporate the cut-off period into the witness. This is enforced by the counter in the bulletin board as described in Section 2.1. In the protocol, this translates to a window (set) of counter values which qualify as the additional variable in the witness. To ensure that the number of witnesses are still polynomial, the window size has to be polynomial. We parameterize the protocol with the size of this window, and the parties choose the start point of the window.

As discussed in the introduction, for the proof in this model, it is essential that the simulator is able to reset the bulletin board to a prior point (in essence, rewinding).

– rewind: This functionality is reserved for the simulator in the ideal world. On receiving additional input t, the bulletin board internally resets its

counter to t and clears all data stored beyond the counter value t. The simulator gets no output on this query.

$$\perp \leftarrow \mathsf{Ledger.rewind}(t)$$

We want to stress that this additional capability is only limited to the construction in this section and the construction in the next section (using trusted hardware) we will not require this.

We additionally discuss an extension to an optimistic phase where the parties can share some additional secrets (different from before) that enable them to decrypt a (different) ciphertext containing the output, without having to post to the bulletin board. Of course, the adversary can prematurely abort in this phase and obtain the output for itself. To protect against this, the optimistic phase is reached only once it has been established that the parties have enough information that would enable them to use the bulletin board, to decrypt to the output, in case the adversary aborts in this phase.

Construction. We now proceed to describe our protocol Π_{fair}. It uses the cryptographic primitives and a bulletin board as described below. The formal protocol description is given in Figure ??.

1. A injective one-way functions f.

2. An authentication scheme with public verification $(\mathsf{Gen}, \mathsf{Tag}, \mathsf{Verify}_{\mathsf{Ledger}})$ such that the authentication tags are unique for a given message.

3. A witness encryption WE for the language

$$L_{\mathsf{WE},\Delta t} = \Big\{ \Big(\{y_i\}_{i\in[n]}, T \Big) \;\Big|\; \exists \Big(t, \sigma, \{\rho_i\}_{i\in[n]} \Big) \text{ s.t.}$$

$$(\forall i \in [n],\ y_i = f(\rho_i)) \text{ AND}$$

$$t \in \{T, T+1, \cdots, T+\Delta t\} \text{ AND}$$

$$\mathsf{Verify}_{\mathsf{Ledger}}((t||\rho_1||\cdots||\rho_n), \sigma) = 1 \Big\}$$

For a given $x \in L_{\mathsf{WE},\Delta t}$, if f is an injective one-way function and $(\mathsf{Gen}, \mathsf{Tag}, \mathsf{Verify}_{\mathsf{Ledger}})$ is a scheme that generates unique authentication tags, it is easy to see that there are only $\Delta t + 1$ witnesses for x. If Δt is set to be polynomial in the size of x, there are only polynomially many witnesses for any given statement, and thus $L_{\mathsf{WE},\Delta t}$ is a polynomial witness language (see Definition 10). From Theorem 1, given $L_{\mathsf{WE},\Delta t}$ is a polynomial witness language, we know that a witness encryption for $L_{\mathsf{WE},\Delta t}$ is also an extractable witness encryption for $L_{\mathsf{WE},\Delta t}$.

4. An MPC protocol that computes:

$$\mathcal{F}'_{\Delta t}((x_1, \rho_1, t_1), \cdots, (x_n, \rho_n, t_n)) = \Big(c, \{f(\rho_i)\}_{i\in[n]}, T \Big)$$

where $T = \mathsf{max}(t_1, \cdots, t_n)$ and $c = \Pi_{\mathsf{WE}}.\mathsf{Enc}(x_{\mathsf{WE},\Delta t}, \mathcal{F}(x_1, \cdots, x_n))$ for $x_{\mathsf{WE},\Delta t} = (\{f(\rho_i)\}_{i\in[n]}, T)$. We do not require this protocol to be fair. Importantly, we use the MPC protocol in the common random string (CRS) model. This allows for black-box simulation of the adversary without the necessity of rewinding. For this section, we shall drop the

CRS notation, but it will be implicit.

Protocol Π_{fair} in the $\mathcal{F}'_{\Delta t}$-Hybrid model

Inputs: Each party P_i has an input x_i.
Common input: The verification key for the bulletin board $\mathsf{vk}_{\mathsf{Ledger}}$.
The protocol:

1. **Computation of $\mathcal{F}'_{\Delta t}$.**
   - P_i samples token $\rho_i \xleftarrow{\$} \{0,1\}^{\mathrm{poly}(\lambda)}$.
   - P_i queries the bulletin board to get the current counter value, i.e. $t_i \leftarrow \mathsf{Ledger}(\mathsf{getCurrentCounter})$.
   - P_i sends (x_i, ρ_i, t_i) to the ideal functionality $\mathcal{F}'_{\Delta t}$ and receives $\left(c, \{y_i\}_{i\in[n]}, T\right)$. It aborts if it receives $\perp$ from the ideal functionality.
2. **Exchange of tokens.** P_i broadcasts ρ_i to all other parties, and receives $\{\widehat{\rho}_l\}_{l\in[n]\setminus\{i\}}$.
3. **Obtaining the output.** We split this into three cases, where either (i) P_i can post on the bulletin board to receive a valid witness; or (ii) P_i waits for another party to post to the bulletin board; or (iii) no party posts to the bulletin board.
   (a) P_i received ρ_j from all the other parties, such that $\forall j \in [n] \setminus \{i\} : f(\rho_j) = y_j$. In this case, P_i waits for the counter to get to T before posting to the bulletin board. Prior to posting, it check to see if another party has already posted the same. This could be done either by observing the broadcasts sent (to the bulletin board), or querying the bulletin board at most Δt times. On obtaining the appropriate authentication tag, the witness encryption can be decrypted to get the output.
   (b) P_i received a ρ_j such that $f(\rho_j) \neq y_j$, or $\rho_j = \perp$ (i.e. a party didn't send its token). In this case, P_i checks if the right message is posted to the bulletin board for counter values between T and $T + \Delta T$. If it finds the right value, it obtains the authentication tag and decrypts the witness encryption to get the output.
   (c) If there are no posts on the bulletin board satisfying the given requirements, and the counter has progressed beyond $T + \Delta t$, P_i aborts.

Figure 4.1: Π_{fair} in the $\mathcal{F}'_{\Delta t}$-Hybrid model. The protocol relies on the security of witness encryption for a polynomial witness language, injective one-way functions and authentication scheme with public verification and unique tags.

Remark 1 *In the construction described above, the size of the witness encryption circuit is dependent on the number of parties in the protocol. This can be remedied*

by using the XOR of the ρ_i values as the release token, and applying the injective one-way function on this. The rest of the protocol remains the same.

4.3.1 Proof of Security

We prove the security of our construction in the $\mathcal{F}'_{\Delta t}$-hybrid model.

Simulator $\mathcal{S}$. We start by constructing a simulator $\mathcal{S}$. Our simulator uses rewinding strategy similar to the one described in (Gordon, 2010) (which in turn builds on (Goldreich and Kahan, 1996b)). The simulator has access to an ideal functionality for computing $\mathcal{F}$, and simulates $\mathcal{F}'_{\Delta t}$ for the real world adversary. In addition, for the proof in this model, the simulator reserves the right to reset the bulletin board to prior point (in essence, rewinding). (We will not require this property in the protocol based on secure hardware.) Further, $\mathcal{S}$ forwards any queries the adversary makes to the bulletin board, and returns the corresponding response from the bulletin board.

1. $\mathcal{S}$ receives inputs $\{(x_a, \rho_a, t_a)\}_{a \in \mathcal{A}}$ sent by the adversary that are intended for $\mathcal{F}'_{\Delta t}$.

2. Mark the current value of the counter so that $\mathcal{S}$ can rewind the bulletin board to this point.

$$t_{\mathsf{mark}} \leftarrow \mathsf{Ledger.getCurrentCounter}()$$

3. $\mathcal{S}$ simulates the output of ideal functionality computing $\mathcal{F}'_{\Delta t}$ as follows:

   (a) Set $T = \max\{\{t_a\}_{a \in \mathcal{A}}, t_{\mathsf{mark}}\}$.

   (b) Randomly pick $\{\rho_h\}_{h \in \mathcal{H}}$ for the honest parties.

(c) $\forall i \in [n]$, $y_i \coloneqq f(\rho_i)$.

(d) Compute $\widehat{\mathsf{out}} \leftarrow \mathcal{F}(\widehat{x}_1, \cdots, \widehat{x}_n)$ where $\widehat{x}_h = 0$ for all $h \in \mathcal{H}$.

(e) Set $x_{\mathsf{WE}} \coloneqq \left(\{y_i\}_{i\in[n]}, T\right)$ and compute

$$c \leftarrow \Pi_{\mathsf{WE}}.\mathsf{Enc}\left(x_{\mathsf{WE}}, \widehat{\mathsf{out}}\right).$$

(f) Send $(c, \{y_i\}_{i\in[n]}, T)$ to the adversarial parties.

(g) If the adversary responds with an abort, $\mathcal{S}$ sends abort to the ideal functionality computing $\mathcal{F}$, and exits. For our analysis, we denote this by abort_1.

4. $\mathcal{S}$ sends values $\{\rho_h\}_{h\in\mathcal{H}}$ to the adversary. If the adversary sends values $\{\rho_a\}_{a\in\mathcal{A}}$ such that $\forall a, y_a = f(\rho_a)$; or sends a post query to the bulletin board with value $(\rho_1||\cdots||\rho_n)$ when counter value is between T and $T + \Delta t$ such that $\forall i, y_i = f(\rho_i)$, the adversary has not aborted.

5. If the adversary aborted in the previous step, $\mathcal{S}$ sends abort to the ideal functionality computing $\mathcal{F}$, and exits. For our analysis, we denote this by abort_2.

6. If the adversary didn't abort prior to this, we need to estimate the probability of the adversary not aborting. Let q represents the true of probability of this event, where the randomness is over random coins used in step 3(b) and 3(e). The estimated probability will be denoted by $\widetilde{q}$.

   (a) $\mathcal{S}$ fixes some number $t = \mathrm{poly}(\lambda)$.

   (b) $\mathcal{S}$ rewinds the adversary to step 3, rewinds the bulletin board $\mathsf{Ledger.rewind}(t_{\mathsf{mark}})$ and repeats steps 3 and 4 (other than 3(g)) with

fresh randomness each time. Repeat till the adversary has not aborted t times.

(c) $\mathcal{S}$ estimates q as $\widetilde{q} = \frac{t}{\#\text{ of repetitions}}$. The polynomial defining t is chosen to be large enough that

$$\mathsf{Pr}\left[\frac{1}{2} \leq \frac{q}{\widetilde{q}} \leq 2\right] > 1 - 2^{\lambda}.$$

7. The simulator sends $\{x_a\}_{a\in\mathcal{A}}$ to the ideal functionality for $\mathcal{F}$ and receives out. $\mathcal{S}$ repeats the following at most $\frac{t}{\widetilde{q}}$ times.

   (a) With fresh randomness each time, $\mathcal{S}$ rewinds the adversary to step 3, rewinds the bulletin board $\mathsf{Ledger.rewind}(t_{\mathsf{mark}})$ and repeats steps 3 and 4 (other than 3(g)) replacing $\widehat{\mathsf{out}}$ with out.

   (b) If the adversary does not abort, we output its view and the simulator terminates.

8. If $\mathcal{S}$ has not terminated yet, output fail and terminate the simulation.

Claim 1 *If simulator $\mathcal{S}$ does not outputs* fail*, the hybrid world and the ideal world are indistinguishable.*

Proof. [Proof of Claim 1] We split the analysis into two cases:

– *Case 1: The adversary does not abort.* Since the simulator does not output fail, it has successfully got the adversary to accept the transcript for the right output. In this case, the main thread of the adversary is statistically indistinguishable from the real execution. Additionally, since the simulator is able to rewind the bulletin board, the adversary's view of

the bulletin board is that of a straight line execution. Thus the joint distribution consisting of the view of the adversary and the honest party outputs is indistinguishable.

– *Case 2: The adversary aborts.* As noted in the simulator, the adversary can abort in two phases of the protocol. We deal with the two case separately: abort_1**:** The adversary aborts immediately after running the MPC for $\mathcal{F}'_{\Delta t}$. In both the real and ideal world, honest parties do not get any output. Thus, we need to argue that the adversary's view is indistinguishable when he receives a witness encryption of the actual output as opposed to when he received the witness encryption of a random string. To the contrary, assume that the adversary can distinguish between these two cases. Since there only polynomially many witnesses, we use the extractor for the adversary to recover the witness. Since the honest parties aborts without revealing its share of the token, we can use the extractor to construct an adversary that breaks the security of the injective one-way function. abort_2**:** The adversary aborts on receiving the honest party's tokens without posting to the bulletin board. We use the same technique as above, leveraging the extractor for witness encryption, to construct an adversary that breaks the unforgability of the authentication tags issued by the bulletin board.

Claim 2 *The simulator $\mathcal{S}$ outputs* fail *with only negligible probability.*

Proof. [Proof of Claim 2] The analysis for the proof below is taken from (Gordon, 2010; Goldreich and Kahan, 1996b). The simulator $\mathcal{S}$ outputs fail only if it has reached step 7 and then fails in producing an accepting transcript. $\mathcal{S}$ fails to reach step 7 with probability q.

We denote by p, the probability that the adversary does not aborts when given the witness encryption of the correct functionality, i.e., p is the probability when the adversary is given the witness encryption of $\mathcal{F}(x_1, \cdots, x_n)$. Recollect that q is the probability of the adversary not aborting when given the witness encryption of $\mathcal{F}(\widehat{x}_1, \cdots, \widehat{x}_n)$ where $\forall a, \widehat{x}_a = x_a$ and $\forall h, \widehat{x}_h = 0$. From the security of witness encryption, we require $|q - p|$ is negligible in the security. (probability is taken over the random coins used to generate the output of $\mathcal{F}'_{\Delta t}$.)

$$\begin{aligned}
\mathsf{Pr}[\mathcal{S} \text{ outputs fail}] &= q \sum_i \left(\mathsf{Pr}\left[\frac{1}{\widetilde{q}} = i\right] \right) (1-p)^{t \cdot i} \\
&\leq q\mathsf{Pr}\left[\frac{q}{\widetilde{q}} \geq \frac{1}{2}\right] (1-p)^{\frac{t}{\widetilde{q}}} + q\mathsf{Pr}\left[\frac{q}{\widetilde{q}} < \frac{1}{2}\right] \\
&\leq q(1-p)^{\frac{t}{2q}} + \mathsf{negl}(\lambda) \qquad (4.1)
\end{aligned}$$

To show that the above equation is negligible in λ, we split the analysis into two cases:

– **Case 1:** $p \geq \frac{q}{2}$. Substituting, we get

$$(1-p)^{\frac{t}{2q}} \leq \left(1 - \frac{q}{2}\right)^{\frac{t}{2q}} < e^{-\frac{t}{4}}$$

which is negligible is λ since t is polynomial in λ.

– **Case 1:** $p < \frac{q}{2}$. To the contrary, let us assume Equation 4.1 is non-negligible. Then, there is a polynomial poly and infinitely many values λ such that

$$q \geq q(1-p)^{\frac{t}{2q}} + \mathsf{negl}(\lambda) > \frac{1}{\mathrm{poly}(\lambda)}.$$

Thus $q > \frac{1}{\mathrm{poly}'(\lambda)}$ for some polynomial poly$'$. This gives us

$$|q-p| > \left|\frac{q}{2}\right| > \frac{1}{2\mathrm{poly}'(\lambda)}.$$

This breaks the security of the witness encryption scheme.

Thus $\mathcal{S}$ outputs fail with only negligible probability.

We assume that the value of T chosen in the protocol is such that the real execution of the protocol ends in time bounded above by a polynomial $g(\lambda)$. Otherwise $\mathcal{F}'_{\Delta t}$ implements an additional check to ensure this.

Claim 3 *The simulator $\mathcal{S}$ runs in expected polynomial time.*

Proof. [Proof of Claim 3] With probability $1-q$, the simulator aborts prior to step 6. With probability q, the simulator goes through the estimation phase and then attempts to force an accepting transcript onto the adversary. The expected number of iterations for the estimation phase is $\frac{t}{q}$ and the cut-off point for forcing the transcript is $\frac{t}{\tilde{q}} < \frac{t}{2q}$. Hence the total expected running time is bounded by

$$g(\lambda) \cdot q \cdot \left(\frac{t}{q} + \frac{2t}{q}\right) = g'(\lambda)$$

Thus $\mathcal{S}$ runs in expected polynomial time.

Given the above claims, the following theorem follows.

Theorem 3 *Assuming the security of injective one-way functions, witness encryption for polynomial witness language and the unforgeability of the authentication scheme, the above protocol satisfies Definition 7 in the $\mathcal{F}'_{\Delta t}$-hybrid model.*

As mentioned in the introduction, in order to rely on the standard security of WE, we require the bulletin board's proof of publish to be implemented via unique signatures (Goldwasser and Ostrovsky, 1993; Lysyanskaya, 2002). If the bulletin board is implemented via standard signatures (e.g., in Google Transparency Certificates) or proofs of stake (e.g., in Etherium), then we require the stronger assumption of extractable witness encryption (Boyle, Chung, and Pass, 2014).

4.4 Fairness from Secure Hardware

A key limitation of our previous constructions is the need to use Witness Encryption (WE) to protect the output of the MPC protocol. Unfortunately, current proposed WE construction are inefficient, due to the high overhead of current constructions of multilinear maps. Moreover, the Witness Encryption paradigm requires the parties to compute a new WE ciphertext for each invocation of the MPC protocol.

In this section we investigate an alternative paradigm that uses *secure hardware*. Our work is motivated by the recent deployment of commodity virtualization technologies such as Intel's Software Guard Extensions (SGX). These technologies allow for the deployment of secure "enclave" functionalities that can store secrets and perform correct computation even when executed in an adversarial environment. Moreover, these systems allow an

enclave to remotely *attest* to their correct functioning, which allows for the establishment of trustworthy communications between enclaves running on different machines.

Model. Following the approach of Pass *et al.* (Pass, Shi, and Tramèr, 2017) we model all available trusted hardware processors from a given manufacturer as a single, globally shared functionality denoted $\mathcal{G}_{\mathsf{att}}$ (see Figure 2.5). We describe the functionalities required for our construction, and refer the reader to (Pass, Shi, and Tramèr, 2017) for details. `install` loads the program prog onto the attested hardware. It returns an enclave identifier *eid*. (For simplicity, we skip the session identifier used in (Pass, Shi, and Tramèr, 2017).) The enclave identifier is be used to identify the enclave upon resume. `resume` allows for a stateful resume using the unique enclave identifier generated. On running over a given input, the output produced is signed to attest that the enclave with identifier *eid* was installed with a program prog, which was then executed to produce the output. The program's input is not included in the attestation.

Description. We describe here the main ideas in this construction that differ from the previous construction. Upon loading the program onto the attested hardware (enclaves), there is an initial key exchange to establish a secure authenticated channel between the enclaves. Any information passed over this channel is hidden from the parties. It is important that enclaves attest to the fact that they are running the correct programs prior to the key exchange.

Next, the shares of the release token and the key are input to the enclave. The enclaves use the established secure authenticated channel to exchange

this information and set up consistent parameters (over all enclaves) for the decryption circuit. The parties then run an MPC protocol external to the enclaves to compute an encrypted version of the output. As in our previous construction, the players exchange shares of the release token that they are required to post onto the bulletin board in order to decrypt.

For technical reasons, we need to ensure that the key share that a party sends to the enclave is the one used in the MPC. This is ensured by using a commitment scheme which the MPC computation verifier before returning the output.

Our protocol requires the following primitives:

1. A one-way function f.[4]
2. A signature schemes $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Verify})$.
3. An authentication scheme with public verification $(\mathsf{Gen}, \mathsf{Tag}, \mathsf{Verify}_{\mathsf{Ledger}})$.
4. A multi-party computation in the CRS model for computing $\mathcal{F}'$ defined as

$$\mathcal{F}'\left(\left\{x_i, k_i, \{\mathsf{com}_{ij}\}_{j\in[n]}, r_i\right\}_{i\in[n]}\right) =$$

$$\begin{cases} \perp & \text{if } \exists i, i' \text{ s.t. } \left(\{\mathsf{com}_{ij}\}_{j\in[n]}\right) \neq \left(\{\mathsf{com}_{i'j}\}_{j\in[n]}\right) \\ \perp & \text{if } \exists i \text{ s.t. } \mathsf{com}_{ii} \neq \mathsf{Com}(k_i; r_i) \\ y & \text{otherwise} \end{cases}$$

where $y = \mathsf{AE.Enc}_{\bigoplus_{i=1}^{n} k_i}(\mathcal{F}(x_1, \cdots, x_n))$. Essentially the MPC takes in a the input, key share, a commitment tuple and a decommitment from

[4] In practice we suggest using a hash function.

each party. It checks if the tuple pairs received are the same throughout and the commitment linked to each party decommits to the key share. If this check fails it just returns $\perp$, or it returns the output y.

5. Two instances of AE scheme $(\mathsf{Enc}, \mathsf{Dec})$ with INT-CTXT security for authentication and semantic security.

6. A commitment scheme (Com) with computational hiding and statistically binding.

We describe and prove the protocol in the two party setting. Both extended naturally to the multi-party setting. The protocol is described in Figure **??**.

We note that there are two trapdoors installed into functionalities of $\mathsf{prog}_{\mathsf{fair}}$. These are used for the security reduction of the one-way function, and to program the output correspondingly. Specifically, the trapdoor is used to get the enclave to attest to a value of choice. These trapdoors can be used by an adversarial party, but this makes no difference to the security since these values are not sent across to the other party.

Theorem 4 *Assume that F is one-way, the signature scheme is existentially unforgeable under chosen message attacks, the authentication scheme satisfies standard notion of unforgeability, the encryption scheme is perfectly correct, authenticated encryption scheme that is perfectly correct and satisfies standard notions of INT-CTXT and semantic security, decisional Diffie-Hellman assumption holds in the algebraic group adopted. Then, the above protocol satisfies Definition 7 in the* $(\mathcal{G}_{\mathsf{att}}, \mathcal{F}')$*-hybrid model.*

4.4.1 Proof of Security

We now prove Theorem 4. We consider the two party setting where P_1 is corrupted. The other case is symmetric. The simulator $\mathcal{S}$ works as follows:

1. Unless otherwise mentioned, $\mathcal{S}$ passes through messages between adversary $\mathcal{A}(P_1)$ and $\mathcal{G}_{\mathsf{att}}$.

2. $\mathcal{S}$ loads the program to get the corresponding eid_0, i.e.

$$eid_0 \leftarrow \mathcal{G}_{\mathsf{att}}.\texttt{install}(\mathsf{prog}_{\mathsf{fair}}[\Delta t, \mathcal{P}_0, \mathcal{P}_1, vk_{\mathcal{L}}^{\mathsf{Verify}}, 0])$$

3. Next, $\mathcal{S}$ initiates the key exchange phase $(g^a, \sigma_0) \leftarrow \mathcal{G}_{\mathsf{att}}.\texttt{resume}(eid_0, \text{“keyex”})$ and sends (eid_0, g^a, σ_0) message to $\mathcal{A}$.

4. $\mathcal{S}$ waits to receive $\left(eid_1, g^b, \sigma_1\right)$ from $\mathcal{A}$.

   The simulator sees messages between $\mathcal{A}$ and $\mathcal{G}_{\mathsf{att}}$, and can see if $\left(eid_1, g^b, \sigma_1\right)$ sent by $\mathcal{A}$ is different from the corresponding tuple it received from $\mathcal{G}_{\mathsf{att}}$.
   If the tuples differ and the signature verifies, output $\perp_{\mathcal{G}_{\mathsf{att}}}$ and exit.

5. Pick $k_0 \xleftarrow{\$} \{0,1\}^\lambda, \rho_0 \xleftarrow{\$} \{0,1\}^\lambda, r_0 \xleftarrow{\$} \{0,1\}^\lambda, t_0 \leftarrow \mathcal{L}^{\mathsf{Verify}}.\mathsf{GetCounter}()$ and initialize $\mathcal{G}_{\mathsf{att}}$ with these values, $(\mathsf{com}_0, \_) \leftarrow \mathcal{G}_{\mathsf{att}}.\texttt{resume}(eid_0, \text{“init”}, \rho_0, k_0, t_0, r_0)$.
   Simulator sees the values (ρ_1, k_1, t_1, r_1) that $\mathcal{A}$ sends to $\mathcal{G}_{\mathsf{att}}$.

6. $\mathcal{S}$ calls $(\mathsf{ct}_1, \_) \leftarrow \mathcal{G}_{\mathsf{att}}.\texttt{resume}(eid_0, \text{“send”})$, sends ct_1 to $\mathcal{A}$ and waits for ct_0.

   As before, the simulator observes if ct_0 sent by $\mathcal{A}$ is different from the value it received from $\mathcal{G}_{\mathsf{att}}$. If so, and $\mathcal{G}_{\mathsf{att}}$ doesn't throw an exception, output $\perp_{\mathsf{AE}_1}$ and exit.

7. Make a call to $\mathcal{G}_{\mathsf{att}}$ to get the parameters,

$$(T, y, \_) \leftarrow \mathcal{G}_{\mathsf{att}}.\texttt{resume}(eid_0, \text{“getParams”}, \perp)$$

8. Wait for $\mathcal{A}$ to send $(x_1, k_1', \mathsf{com}_0', \mathsf{com}_1', r_1')$ intended for $\mathcal{F}'$. If the commitment values are not the same as the ones received earlier, send abort to the ideal functionality and send $\perp$ to the adversary. If $k_1' \neq k_1$, i.e. the key shares sent at different points differ, and if $\mathsf{com}_1' = \mathsf{Com}(k_1'; r_1')$ output $\perp_{\mathsf{com}}$ and exit. Else, pick K' randomly and compute $\mathsf{ct}_{\mathsf{MPC}} \leftarrow \mathsf{AE.Enc}_{K'}(\mathcal{F}(0, x_1))$ to send to $\mathcal{A}$.

9. $\mathcal{S}$ obtains its token share from $\mathcal{G}_{\mathsf{att}}$,

$$(\rho_0, \_) \leftarrow \mathcal{G}_{\mathsf{att}}.\texttt{resume}(eid_0, \text{“getTokenShare”}, \perp)$$

10. If $\mathcal{A}$ aborts immediately after receiving the output from $\mathcal{F}'$ without the honest party getting it, send abort to the ideal functionality. But continue running $\mathcal{S}$. If the adversary sends to the bulletin board or enclave the correct pre-image of y, $\mathcal{S}$ outputs $\perp_f$ and exits.

11. If $\mathcal{A}$ has not aborted, send ρ_0 to $\mathcal{A}$. If the adversary does not send ρ_1, or post a valid pre-image during the interval T to $T + \Delta T$, but queries $\mathcal{G}_{\mathsf{att}}$ for the output on a valid authentication tag, then we output $\perp_{\mathsf{Ledger}}$ and exit.

12. Alternatively, we split the behavior of the simulator three cases:

    – If $\mathcal{A}$ responds with a valid ρ_1 (i.e. $f(\rho_0 \oplus \rho_1) = y$), then post to the bulletin board. Recollect that $\mathcal{S}$ has reached this point only if the key

shares sent by $\mathcal{A}$ were consistent. Send x_1 to the ideal functionality to receive out. If $\mathcal{A}$ makes the correct query to the enclave, i.e. the ciphertext sent is the same as the one from the MPC, $\mathcal{S}$ programs the output by returning $\mathcal{G}_{\mathsf{att}}.\texttt{resume}(eid_1, \text{“output”}, t_{\mathcal{L}}^{\mathsf{Verify}}, \sigma_{\mathcal{L}}^{\mathsf{Verify}}, \mathsf{out})$. If the ciphertext is different and authenticates under key K', then output $\perp_{\mathsf{AE}_2}$ and exit.

– If the adversary does not send ρ_1 but posts a pre-image of y during the interval T to $T + \Delta T$, $\mathcal{S}$ follows the same approach as the previous step.

– If $\mathcal{A}$ attempts to use the backdoor, forward the message to $\mathcal{G}_{\mathsf{att}}$ without modification.

We prove indistinguishability of the real and ideal worlds through a sequence of hybrids.

$\mathcal{H}_0$**:** Identical to the real execution.

$\mathcal{H}_1$**:** Identical to $\mathcal{H}_0$ except that we introduce the following check. Observe the messages between $\mathcal{A}$ and $\mathcal{G}_{\mathsf{att}}$, and can see if $\left(eid_1, g^b, \sigma_1\right)$ sent by $\mathcal{A}$ is different from the corresponding tuple it received from $\mathcal{G}_{\mathsf{att}}$. If the tuples differ and the signature verifies, output $\perp_{\mathcal{G}_{\mathsf{att}}}$ and exit.

Claim 4 *Assuming that the underlying signature scheme is secure, $\mathcal{H}_1$ is computationally indistinguishable from $\mathcal{H}_0$.*

Proof. [Proof of Claim 4] $\mathcal{H}_1$ exits with output $\perp_{\mathcal{G}_{\mathsf{att}}}$ with only negligible probability. If not, we can use $\mathcal{A}$ to construct an adversary that breaks the signature scheme.

$\mathcal{H}_2$**:** Identical to $\mathcal{H}_1$ except that we replace all occurrences of $sk = g^{ab}$ with a random key.

Claim 5 *Assuming that DDH holds, $\mathcal{H}_2$ is computationally indistinguishable from $\mathcal{H}_1$.*

Proof. [Proof of Claim 5] Follows directly from DDH security.

$\mathcal{H}_3$**:** Identical to $\mathcal{H}_2$ except that we add the following additional checks:

- observe if ct_0 sent by $\mathcal{A}$ is different from the value it received from $\mathcal{G}_{\mathsf{att}}$. If so, and $\mathcal{G}_{\mathsf{att}}$ doesn't throw an exception, output $\perp_{\mathsf{AE}_1}$ and exit.
- if $\mathcal{A}$ sends a different key share k_1' intended for $\mathcal{F}'_{\Delta t}$ and $\mathsf{com}'_1 = \mathsf{Com}(k_1'; r_1')$, output $\perp_{\mathsf{com}}$ and exit.
- if $\mathcal{A}$ aborts immediately after receiving the output from $\mathcal{F}'$ (without the honest party getting it), send abort to the ideal functionality. Additionally, wait to see if the adversary sends to the bulletin board or enclave the correct pre-image of y. If so, outputs $\perp_f$ and exit.
- if the adversary does not send ρ_1 and does not post a valid pre-image during the interval T to $T + \Delta T$ but queries $\mathcal{G}_{\mathsf{att}}$ on a valid authentication tag, output $\perp_{\mathsf{Ledger}}$ and exit.

Claim 6 *Assuming the security of one-way permutation, statistical binding of the commitment scheme INT-CTXT security of AE and unforgeability of the authentication scheme $\mathcal{H}_3$ is computationally indistinguishable from $\mathcal{H}_2$.*

Proof. [Proof of Claim 6] The only changes are in the checks performed. We argue that $\mathcal{H}_3$ will output a special abort with only negligible probability:

- $\perp_{\mathsf{AE}_1}$ is output with only negligible probability. Else, we can leverage the adversary to break the INT-CTXT security of the AE scheme.
- $\perp_{\mathsf{com}}$ is output with only negligible probability. Else, we can leverage the adversary to break the statistical binding property of the commitment scheme.
- $\perp_f$ is output with only negligible probability. Else we can break the security of the one way function. This follows from the fact that the simulator is see the queries that the adversary makes to the enclave and the bulletin board. Since we want to force the challenge value y^* onto the adversary, we use a backdoor in the function. This backdoor does not give the adversary any undue advantage as the value is not sent across to the other party.
- $\perp_{\mathsf{Ledger}}$ is output with negligible probability. Else, we can leverage the adversary to break the unforgeability of the authentication scheme for the bulletin board. This is because the adversary was able to produce a signature that has not been queried for before.

$\mathcal{H}_4$**:** Identical to $\mathcal{H}_3$ except that we intercept the ciphertext query for the output, and program the output using the trapdoor to be $\mathsf{AE.Dec}_K(\mathsf{ct})$ if the other conditions are satisfied. Here K is the key in the enclave.

Claim 7 *$\mathcal{H}_4$ is statistically indistinguishable from $\mathcal{H}_3$.*

Proof. [Proof of Claim 7] This follows from the fact that it was only a statistical change. This is because we moved the exact check to the outside of the enclave.

$\mathcal{H}_5$**:** Identical to $\mathcal{H}_4$ except that replace com_2 to be a commitment of a random value.

Claim 8 *If the commitment scheme is computationally hiding, $\mathcal{H}_5$ is computationally indistinguishable from $\mathcal{H}_4$.*

Proof. [Proof of Claim 8] If the two hybrids are distinguishable, we can leverage the adversary to break the computational hiding of the commitment scheme.

$\mathcal{H}_5$**:** Identical to $\mathcal{H}_4$ except that we pick K' randomly and use K' to encrypt the output. Now, the output is programmed in the last round with respect to the key K'.

Claim 9 *If the semantic security of the AE scheme holds, $\mathcal{H}_5$ is computationally indistinguishable from $\mathcal{H}_4$.*

Proof. [Proof of Claim 9] If the two hybrids are distinguishable, then we can build an adversary that breaks the semantic security of the AE scheme.

$\mathcal{H}_6$**:** Identical to $\mathcal{H}_5$ except that we add the following additional checks. If the ciphertext differs from the MPC output and it authenticates under key K', then output $\perp_{\mathsf{AE}_2}$ and exit.

Claim 10 *If INT-CTXT security of the AE scheme holds, $\mathcal{H}_6$ is computationally indistinguishable from $\mathcal{H}_5$.*

Proof. [Proof of Claim 10] Since the only changes are additional checks, it is enough to show that $\mathcal{S}$ outputs $\perp_{\mathsf{AE}_2}$ with only negligible probability. This follows directly from the INT-CTXT security of the AE scheme. Specifically, we can receive the challenge ciphertext to the be the encryption of the function value (either under the challenge key, or a random key). If the adversary is able to produce a verifying ciphertext different from the one it receives it constitutes a forgery, thus breaking the INT-CTXT security of the AE scheme.

$\mathcal{H}_7$: Identical to $\mathcal{H}_6$ except that if check did not result in a failure, send x_1 to the trusted party to obtain out. If the witness checks succeeds, program the output of the enclave to be out. Else program output to be $\perp$.

Claim 11 *$\mathcal{H}_7$ is statistically indistinguishable from $\mathcal{H}_6$.*

Proof. [Proof of Claim 11] The change is only statistical since the execution thread reaches the point only if all prior checks pass.

$\mathcal{H}_8$: Identical to $\mathcal{H}_7$ except that we replace the value inside the ciphertext to be $\mathcal{F}(0, x_1)$.

Claim 12 *If the semantic security of the encryption scheme holds, $\mathcal{H}_8$ is computationally indistinguishable from $\mathcal{H}_7$.*

Proof. [Proof of Claim 12] If the two hybrids are distinguishable, then we can build an adversary that breaks the semantic security of the encryption scheme.

$\mathcal{H}_9$: Identical to $\mathcal{H}_8$ except that we replace all occurrences of sk with g^{ab} again.

Claim 13 *Assume DDH is hard, $\mathcal{H}_9$ is computationally indistinguishable from $\mathcal{H}_8$.*

Proof. [Proof of Claim 13] Follows directly from DDH security.

$\mathcal{H}_9$ is the same as our simulator, and hence we're done.

4.5 Instantiating the Bulletin Board

Our proposed paradigm relies on a *verifiable* public bulletin board that makes three guarantees about entries posted to it:

- The entry's presence can be cryptographically verified using a public operation.
- Once posted, the entry is available to all parties.
- Entries are assigned a unique monotonically increasing sequence number.

We now consider several existing techniques that we can leverage to obtain such a bulletin board.

Certificate Transparency Logs. Certificate Transparency (CT) (*Certificate Transparency* 2017) is a public audit log operated by a coalition of browser vendors and certificate authorities. CT allows individual certificate authorities to post newly-issued certificates to a public log. These entries are then (1) signed by the log maintainer, and (2) added to a Merkle hash tree. The root of the hash tree is also signed by (one or more) log maintainers and published to the world.

A collection of users known as *monitors* can access the CT log to view the contents of certificates. While the CT log is itself not fundamentally tamper resistant – since the servers operating it can remove portions and/or be disabled by remote network attacks – any tampering is detectable due to the structure of the Merkle hash tree. The location of the entry within the Merkle hash tree also serves to act as a proxy for a monotonically increasing sequence number.

Under the assumption that the existing CT logs are reliable and trustworthy, we can use CT to build fairness systems by entombing the required public data into a component of an X.509 certificate signing request and requesting the certificate from a free certificate authority such as LetsEncrypt (*Let's Encrypt* 2017). Because LetsEncrypt submits all certificates to a public log[5] it is possible for any party to recover these certificates and verify a cryptographic proof that the entries have been published.

Public blockchains. Crypto-currencies such as Bitcoin or Etherium rely on a publicly available data structure called a *blockchain*. Block-chains are an append-only ledger that is maintained by an ad-hoc group of network peers. Blockchains come in two basic types. The first type use computational *proofs of work* to determine which peer should be allowed to add a new block of transactions to the blockchain. Clients accept the longest chain that contains well formed transactions; as a result the system is secure as long as a super-majority of the computational power in the network is controlled by honest peers. This approach is tolerant of churn, and thus we need not pick a set of

[5]See https://crt.sh/?id=15707024

	Initialization	Decrypt
mean	1.180 ± 0.112	0.039 ± 0.001
mean	0.002 ± 0.000	0.037 ± 0.001

Table 4.1: Performance of SGX enclave setup and decryption (not MPC). Average and standard deviation of 500 runs.

honest parties in advance.

An alternative approach uses *proof of stake* (Bentov, Gabizon, and Mizrahi, 2016). In these systems a quorum of peers is sampled from the network with probability proportional to the fraction of monetary holdings controlled by each peer. This quorum is responsible for producing the next block and selecting the next quorum by the same mechanism. The peers authenticate the resulting block by signing it using a secure digital signature scheme. The security assumption here assumes that the parties with the largest share of the cryptocurrency have a vested interest in keeping it running. Proof of stake systems are in their infancy both in terms of deployment and theory. However, they provide an interesting middle ground between the costs of a pure proof of work approach and the challenges with selecting a set of trusted parties a priori to maintain the bulletin board.

4.6 Implementation

In this section we present an implementation of the protocol given in Section 4.4, and show that the protocol is efficient. Our implementation consists of three major pieces: the bulletin board instantiated using Bitcoin, the MPC protocol instantiated using the SPDZ framework (Damgard et al., 2012; *Multiparty computation with SPDZ online phase and MASCOT offline phase* 2017),

and a "witness decryptor" instantiated using an Intel SGX secure enclave. We describe each component in more detail below.

Bitcoin as a bulletin board. For our prototype implementation we use the Bitcoin network, which supports a limited scripting system called Bitcoin Script. In Bitcoin each transaction contains a script that is evaluated to ensure the transaction is authorized. This scripting system supports an instruction named `OP_RETURN`, which allows the sender of a transaction to embed up to 40 bytes of arbitrary data into a transaction that is transmitted for inclusion in the Bitcoin blockchain. Each block of transactions in the blockchain contains a computational proof of work (PoW) that is computed by the network. This proof is bound cryptographically to all of the transactions within a block, as well as to the hash of the previous block. At current network difficulty, computing a proof of work for a single block requires an expected 2^{64} invocations of the SHA2 hash function on the standard Bitcoin network. To verify publication on the bulletin board, our implementation requires a fragment consisting of six consecutive blocks (where the transaction is located in the first block of the fragment). The cost of forging such a fragment scales linearly in the number of blocks required.

We note that the use of a computational proof of work bulletin board provides somewhat different fairness properties than a signature-based bulletin board, *e.g.*, Certificate Transparency or a proof-of-stake blockchain. Specifically, in this setting an attacker with sufficient time or computational power can always "forge" a satisfying chain of blocks, and use this private result as a witness to enable decryption. Such an attack would be economically

costly, since the corresponding effort – if applied to crypto-currency mining – would be worth a substantial sum of money.[6] However, we can further restrict this attacker by employing a trusted clock within the witness decryptor (*e.g.*, Intel SGX) [7]. This optimization requires the attacker to complete the forgery within a pre-defined time limit that approximates the expected time for the full Bitcoin network. Thus a successful attacker must possess most of the available hashpower of the Bitcoin network (which currently approximates the electrical consumption of Turkmenistan).

For our experiments, we use the public Bitcoin testnet. The Bitcoin testnet functions similarly to the main Bitcoin network, but uses a zero-value currency and a low difficulty setting for the proof of work. We selected testnet for our experiments mainly because blocks are mined extremely rapidly and transactions require no monetary expenditure for "transaction fees". However our code can use the production Bitcoin blockchain without any code changes.

MPC Protocol. Our protocol can be used to extend any MPC scheme that supports efficient symmetric encryption. We note that one could employ Intel SGX directly to perform a naive form of MPC. However, our goal in this work is to demonstrate that our approach works efficiently even when instantiated with a "cryptographic" MPC protocol.[8]

[6] At present rates as of July 2017, this opportunity cost is approximately $28,000 per block forged.

[7] Correctly accessing trusted time from within an enclave is part of the Intel SGX specification, but it is not yet supported as it relies on platform services which are not active. In our implementation, we include code to properly access trusted time, but do not include it in our measurements because of the lack of support.

[8] Additionally, we remark that if SGX is used to implement the MPC protocol itself, a security breach of the SGX system will result in the loss of all security properties provided by the MPC. On the other hand, if we employ a cryptographic MPC protocol, then a failure of Intel's SGX risks only the fairness property. We view this as a benefit of our approach.

Thus for our implementation we use the SPDZ-2 framework developed by the University of Bristol (*Multiparty computation with SPDZ online phase and MASCOT offline phase* 2017). SPDZ-2 is designed to tolerate dishonest majorities during computation. In SPDZ circuits are designed in python and then compiled down into a circuit structure. The computation is done in two phases: an offline phase that does not require the computation inputs and an online phase that performs the actual computation. In order to optimize the running time of the online phase, the pre-computation and compilation phases are relatively more time consuming.

The maintainers of SPDZ-2 have implemented the AES-128 cipher in order to benchmark its efficiency. We repurpose this code to build a simple authenticated encryption system for that uses 3 rounds of AES to encrypt and authenticate one 128-bit block of data output from the computation. The encryption scheme takes as input each party's private computation input x_i and keyshare k_i. It computes as output a ciphertext C encrypted under msk. We also use this AES-128 cipher to implement a commitment scheme. The randomness of the commitment scheme is used as the key to the cipher, with the commitment message as the plaintext.

We construct an MPC circuit for SPDZ-2 that takes in a private input x_i, a keyshare k_i, a randomness share r_i, and commitment to the master key $\mathsf{com}(msk; r)$. The first circuit computes the output of the desired MPC functionality $f(x_1, \ldots, x_N)$. Next it computes $r = \oplus_{1 \leq i \leq N} r_i$ and opens the commitment. It compares $\oplus_{1 \leq i \leq N} k_i$ with the msk from the commitment. If they do not match, sets $f(x_1, \ldots, x_N) = 0$. Finally, the circuit computes the

encryption of $f(x_1, \ldots, x_N)$ using *msk* and outputs the final ciphertext.

SGX as Witness Decryptor. Intel's SGX is a set of extensions to the x86 instruction set that allows for code to be executed in a protected enclave. SGX programs are segmented into two pieces: an untrusted *application* and a trusted *enclave*. The application consists of standard software running on a standard operating system and we assume that it may behave maliciously if the i^{th} player is corrupted. Code within an enclave is verified upon startup and isolated from inspection and tampering, even from an adversary that controls the system's operating system. The root of trust of an SGX enclave is the Intel processor, which enforces the enclave's isolation. It is worth noting that the code run within an enclave is not private; however secrets may be generated or retrieved after the enclave is initialized. Note that the enclave has no direct access to network communications, and must rely on the untrusted part of the application.[9]

We adapt an existing SGX-bitcoin client called Obscuro (*Obscuro* 2017) to perform the role of the Witness Decryptor. This enclave is instantiated by each of the N parties participating in the protocol. A single *master* instance of the enclave uses the `sgx_read_rand` function, supplied by the SGX environment, to generate an AES master key *msk* that will eventually encrypt the output of the MPC circuit. Additionally, the *master* enclave generates a random 320-bit release token t that must be verifiably posted to a bulletin board before the ciphertext can be decrypted. Next, the master applies a secret sharing scheme to derive secret shares $(k_1, \ldots, k_N)$ of *msk* and $(t_1, \ldots, t_N)$ of

[9]This enables the application to censor or tamper with communications between the application and the network.

t. Finally, the master computes a commitment $\mathsf{com}(msk; r)$ and secret shares the randomness into $r_1 \ldots r_N$. Now for $i = 1$ to N it distributes the tuple $(k_i, t_i, msk, \mathsf{t}, \mathsf{com}(msk; \mathsf{r}), \mathsf{r_i})$ to the i^{th} enclave via a secure channel.[10]

Once all secrets have been distributed by the master enclave, the channels are closed and each enclave outputs its key share k_i to the application. The users now invokes SPDZ to conduct the MPC protocol, using as its private inputs x_i, k_i, r_i, and $\mathsf{com}(msk; r)$. If the MPC protocol does not complete successfully, the application aborts and a full restart is required. Otherwise, the application obtains a ciphertext C output by the MPC protocol and provides this as input to the enclave. The enclave attempts to decrypt the ciphertext under msk and if and only if this decryption check completes successfully (and the result is the proper format and length), it releases t_i, which the application then transmits to all of the remaining parties.

To access the encrypted output of the MPC, at least one party must re-compute the release token as $\mathsf{t} = (\mathsf{t_1} \oplus \cdots \oplus \mathsf{t_N})$ and post this value to the Bitcoin network inside of a transaction. Each user's application monitors the Bitcoin network using RPC calls to a local Bitcoin client `bitcoincli` which is running on the user's machine. This userland code then feeds the resulting blockchain fragment (which consists of six consecutive blocks) back to the enclave, which confirms that the release token matches its stored value t, and also verifies the proofs of work on each block. While an adversarial user can block this response, they are unable to falsify or tamper with it due to the fact that such tampering would require an impractical amount of computation.

[10] SGX supports the creation of authenticated, secure channels using attestation and DHKE.

The application also supplies the enclave with the output of the MPC C.

If all verifications succeed, the enclave decrypts the ciphertext C using an authenticated encryption scheme under *msk*, and outputs the resulting plaintext.

Optimizations. In a bid to optimize the implementation, there are a few differences from the described protocol in Section 4.4. They do not make a difference to the security of the protocol, and are briefly described here:

- Instead of each party generating its key and token shares, a designated master enclave chooses them and distributes them to the other enclaves.
- Instead of a commitment for each share of the key, there is only a single "master commitment" of the key.

Sample computation and performance. For proof of concept, we implemented a search program that takes as input a search value x from one party and a list $(y_1, \ldots, y_n)$, from the other party. These circuits each calculate an integer output M and encrypt the result as $\mathsf{Enc}(\bigoplus_{i=1}^{i=n} k_i; M)$. Since these are two-party functions we tested with $N = 2$ and $n = (100, 500, 1000)$.

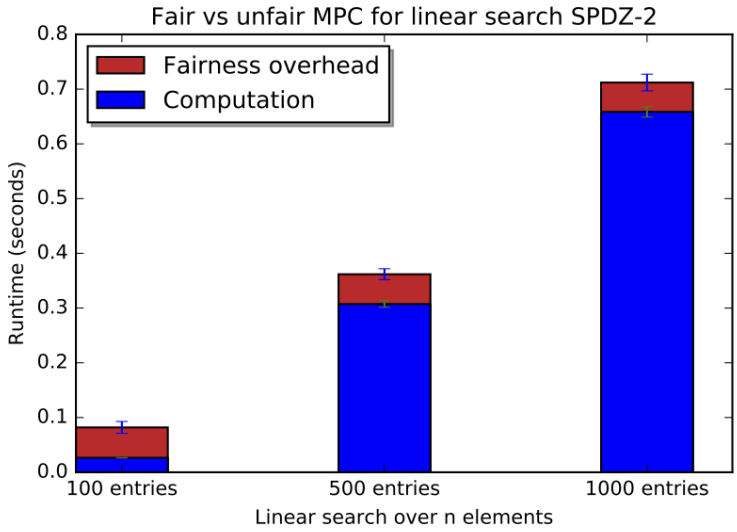

Figure 4.4: Mean runtimes for a linear search on n items using SPDZ taken over 50 iterations. Only the online portion of the MPC is shown. In blue, we show the cost of running the search without any provision for fairness. In red, the overhead from AES encryption needed for fairness.

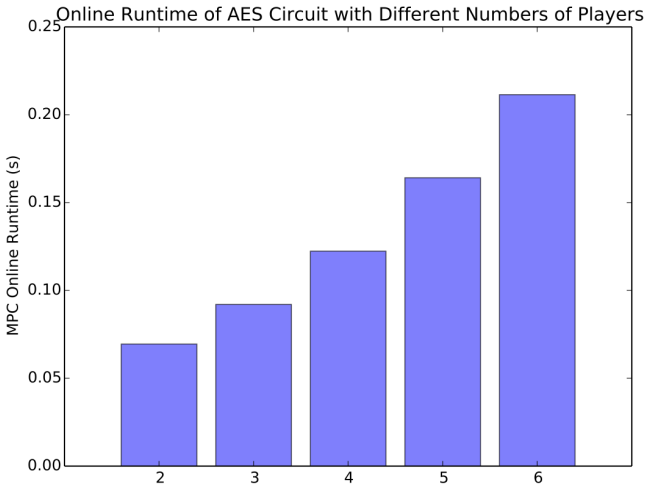

Figure 4.5: Mean runtimes for our AES circuit varied over the number of players participating. Only the online portion of the MPC is shown. This circuit is dominated by 3 AES operations.

Cost of fairness in the MPC. Our implementation demonstrates that our approach can be used add fairness to MPC schemes efficiently using current technology. We recall that fairness in MPC is particularly important when the output of the MPC is extremely valuable. While adding three rounds of AES to a simple MPC scheme represents a high cost, it adds a only a negligible cost when considering more time consuming computations. In Figure 4.4 we show the average runtime over 50 trials of a number of different circuits in SPDZ-2. The cost of encryption is clearly dwarfed by large search problems and set intersection.

While we ran the MPC experiments with N=2 players, SPDZ-2 allows computations with more players. In Figure 4.5 we consider *only the cost of running the encryption component* of the MPC protocol with higher numbers of players. Because each player contributes a key share, the cost of running the protocol increases with each player. While the runtime of the encryption operation does increase, we note that it is still adds only a fraction of one second of online computation time up to $N = 6$.

SGX Runtime. Intel SGX offers an extremely efficient method of trusted program execution. We benchmark our SGX Enclave over 500 trials of the two party protocol for some fixed parameters. We run our test on an Intel i5-6600K 3.5GHz processor with 16 GB of RAM running Ubuntu 14.04 and SGXSDK-1.7, running both the master and minion on the same hardware. For the purpose of benchmarking, we hardcode into the master enclave the master AES key and fix the release token to be the results of an OP_RETURN instruction in a known block of the Bitcoin Testnet. Additionally, we run

the MPC protocol once to generate a valid ciphertext. With the pre-fixed values, we can effectively check the running time of the various parts of the enclave's execution. All key exchange and interaction with the `bitcoincli` is still run as in the real protocol. In Table 4.1 we show the average running times of the various segments of the enclave, both for the master instance and minion instance. For the minion's execution time, we pause the timer while it is waiting for the minion to open a network connection. It is clear that the slowest piece of the program is the enclave initialization. This is because the enclave must provision all memory that it may require from the SGX driver during initialization. Our implementation allocates more memory than it will use to be conservative.

Achieving Fairness in MPC References

Yao, Andrew Chi-Chih (1982). "Protocols for Secure Computations (Extended Abstract)". In: *23rd FOCS*. IEEE Computer Society Press, pp. 160–164. DOI: 10.1109/SFCS.1982.38.

Goldreich, Oded, Silvio Micali, and Avi Wigderson (1987). "How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority". In: *19th ACM STOC*. Ed. by Alfred Aho. ACM Press, pp. 218–229. DOI: 10.1145/28395.28420.

Cleve, Richard (1986). "Limits on the Security of Coin Flips when Half the Processors Are Faulty (Extended Abstract)". In: *18th ACM STOC*. ACM Press, pp. 364–369. DOI: 10.1145/12130.12168.

Gordon, S. Dov, Carmit Hazay, Jonathan Katz, and Yehuda Lindell (2008). "Complete fairness in secure two-party computation". In: *40th ACM STOC*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, pp. 413–422. DOI: 10.1145/1374376.1374436.

Gordon, S. Dov and Jonathan Katz (2009). "Complete Fairness in Multi-party Computation without an Honest Majority". In: *TCC 2009*. Ed. by Omer Reingold. Vol. 5444. LNCS. Springer, Heidelberg, pp. 19–35. DOI: 10.1007/978-3-642-00457-5_2.

Asharov, Gilad, Yehuda Lindell, and Tal Rabin (2013). "A Full Characterization of Functions that Imply Fair Coin Tossing and Ramifications to Fairness". In: *TCC 2013*. Ed. by Amit Sahai. Vol. 7785. LNCS. Springer, Heidelberg, pp. 243–262. DOI: 10.1007/978-3-642-36594-2_14.

Asharov, Gilad (2014). "Towards Characterizing Complete Fairness in Secure Two-Party Computation". In: *TCC 2014*. Ed. by Yehuda Lindell. Vol. 8349. LNCS. Springer, Heidelberg, pp. 291–316. DOI: 10.1007/978-3-642-54242-8_13.

Asharov, Gilad, Amos Beimel, Nikolaos Makriyannis, and Eran Omri (2015). "Complete Characterization of Fairness in Secure Two-Party Computation of Boolean Functions". In: *TCC 2015, Part I*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9014. LNCS. Springer, Heidelberg, pp. 199–228. DOI: 10.1007/978-3-662-46494-6_10.

Cachin, Christian and Jan Camenisch (2000). "Optimistic Fair Secure Computation". In: *CRYPTO 2000*. Ed. by Mihir Bellare. Vol. 1880. LNCS. Springer, Heidelberg, pp. 93–111. DOI: 10.1007/3-540-44598-6_6.

Even, Shimon, Oded Goldreich, and Abraham Lempel (1982). "A Randomized Protocol for Signing Contracts". In: *CRYPTO'82*. Ed. by David Chaum, Ronald L. Rivest, and Alan T. Sherman. Plenum Press, New York, USA, pp. 205–210.

Beaver, Donald and Shafi Goldwasser (1989). "Multiparty Computation with Faulty Majority (Extended Announcement)". In: *30th FOCS*. IEEE Computer Society Press, pp. 468–473. DOI: 10.1109/SFCS.1989.63520.

Goldwasser, Shafi and Leonid A. Levin (1991). "Fair Computation of General Functions in Presence of Immoral Majority". In: *CRYPTO'90*. Ed. by Alfred J. Menezes and Scott A. Vanstone. Vol. 537. LNCS. Springer, Heidelberg, pp. 77–93. DOI: 10.1007/3-540-38424-3_6.

Pinkas, Benny (2003). "Fair Secure Two-Party Computation". In: *EUROCRYPT 2003*. Ed. by Eli Biham. Vol. 2656. LNCS. Springer, Heidelberg, pp. 87–105. DOI: 10.1007/3-540-39200-9_6.

Garay, Juan A., Philip D. MacKenzie, Manoj Prabhakaran, and Ke Yang (2006). "Resource Fairness and Composability of Cryptographic Protocols". In: *TCC 2006*. Ed. by Shai Halevi and Tal Rabin. Vol. 3876. LNCS. Springer, Heidelberg, pp. 404–428. DOI: 10.1007/11681878_21.

Pass, Rafael, Elaine Shi, and Florian Tramèr (2017). "Formal Abstractions for Attested Execution Secure Processors". In: *EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. LNCS. Springer, Heidelberg, pp. 260–289. DOI: 10.1007/978-3-319-56620-7_10.

Andrychowicz, Marcin, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek (2014). "Fair Two-Party Computations via Bitcoin Deposits". In: *FC 2014 Workshops*. Ed. by Rainer Böhme, Michael Brenner, Tyler Moore, and Matthew Smith. Vol. 8438. LNCS. Springer, Heidelberg, pp. 105–121. DOI: 10.1007/978-3-662-44774-1_8.

Bentov, Iddo and Ranjit Kumaresan (2014). "How to Use Bitcoin to Design Fair Protocols". In: *CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario

Gennaro. Vol. 8617. LNCS. Springer, Heidelberg, pp. 421–439. DOI: 10.1007/978-3-662-44381-1_24.

Kumaresan, Ranjit, Tal Moran, and Iddo Bentov (2015). "How to Use Bitcoin to Play Decentralized Poker". In: *ACM CCS 2015*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. ACM Press, pp. 195–206. DOI: 10.1145/2810103.2813712.

Kumaresan, Ranjit and Iddo Bentov (2016). "Amortizing Secure Computation with Penalties". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM Press, pp. 418–429. DOI: 10.1145/2976749.2978424.

Garg, Sanjam, Craig Gentry, Amit Sahai, and Brent Waters (2013). "Witness encryption and its applications". In: *45th ACM STOC*. Ed. by Dan Boneh, Tim Roughgarden, and Joan Feigenbaum. ACM Press, pp. 467–476. DOI: 10.1145/2488608.2488667.

Goldwasser, Shafi and Rafail Ostrovsky (1993). "Invariant Signatures and Non-Interactive Zero-Knowledge Proofs are Equivalent (Extended Abstract)". In: *CRYPTO'92*. Ed. by Ernest F. Brickell. Vol. 740. LNCS. Springer, Heidelberg, pp. 228–245. DOI: 10.1007/3-540-48071-4_16.

Lysyanskaya, Anna (2002). "Unique Signatures and Verifiable Random Functions from the DH-DDH Separation". In: *CRYPTO 2002*. Ed. by Moti Yung. Vol. 2442. LNCS. Springer, Heidelberg, pp. 597–612. DOI: 10.1007/3-540-45708-9_38.

Boyle, Elette, Kai-Min Chung, and Rafael Pass (2014). "On Extractability Obfuscation". In: *TCC 2014*. Ed. by Yehuda Lindell. Vol. 8349. LNCS. Springer, Heidelberg, pp. 52–73. DOI: 10.1007/978-3-642-54242-8_3.

Gentry, Craig, Allison B. Lewko, and Brent Waters (2014). "Witness Encryption from Instance Independent Assumptions". In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, pp. 426–443. DOI: 10.1007/978-3-662-44371-2_24.

Garg, Sanjam, Craig Gentry, and Shai Halevi (2013). "Candidate Multilinear Maps from Ideal Lattices". In: *EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, pp. 1–17. DOI: 10.1007/978-3-642-38348-9_1.

Coron, Jean-Sébastien, Tancrède Lepoint, and Mehdi Tibouchi (2013). "Practical Multilinear Maps over the Integers". In: *CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, pp. 476–493. DOI: 10.1007/978-3-642-40041-4_26.

Gentry, Craig, Sergey Gorbunov, and Shai Halevi (2015). "Graph-Induced Multilinear Maps from Lattices". In: *TCC 2015, Part II*. Ed. by Yevgeniy Dodis and Jesper Buus Nielsen. Vol. 9015. LNCS. Springer, Heidelberg, pp. 498–527. DOI: 10.1007/978-3-662-46497-7_20.

Coron, Jean-Sébastien, Tancrède Lepoint, and Mehdi Tibouchi (2015). "New Multilinear Maps Over the Integers". In: *CRYPTO 2015, Part I*. Ed. by Rosario Gennaro and Matthew J. B. Robshaw. Vol. 9215. LNCS. Springer, Heidelberg, pp. 267–286. DOI: 10.1007/978-3-662-47989-6_13.

Cramer, Ronald and Victor Shoup (1998). "A Practical Public Key Cryptosystem Provably Secure Against Adaptive Chosen Ciphertext Attack". In: *CRYPTO'98*. Ed. by Hugo Krawczyk. Vol. 1462. LNCS. Springer, Heidelberg, pp. 13–25. DOI: 10.1007/BFb0055717.

Gordon, S. Dov, Yuval Ishai, Tal Moran, Rafail Ostrovsky, and Amit Sahai (2010). "On Complete Primitives for Fairness". In: *TCC 2010*. Ed. by Daniele Micciancio. Vol. 5978. LNCS. Springer, Heidelberg, pp. 91–108. DOI: 10.1007/978-3-642-11799-2_7.

Goldreich, Oded and Ariel Kahan (1996a). "How to Construct Constant-Round Zero-Knowledge Proof Systems for NP". In: *Journal of Cryptology* 9.3, pp. 167–190.

Gordon, S. Dov (2010). *On Fairness in Secure Computation*. Ph.D. Dissertation.

Canetti, Ran, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish (2007). "Universally Composable Security with Global Setup". In: *TCC 2007*. Ed. by Salil P. Vadhan. Vol. 4392. LNCS. Springer, Heidelberg, pp. 61–85. DOI: 10.1007/978-3-540-70936-7_4.

Canetti, Ran, Abhishek Jain, and Alessandra Scafuro (2014). "Practical UC security with a Global Random Oracle". In: *ACM CCS 2014*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. ACM Press, pp. 597–608. DOI: 10.1145/2660267.2660374.

Canetti, Ran, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai (2002). "Universally composable two-party and multi-party secure computation". In: *34th ACM STOC*. ACM Press, pp. 494–503. DOI: 10.1145/509907.509980.

Certificate Transparency (2017). Available at https://www.certificate-transparency.org/.

Kiayias, Aggelos, Alexander Russell, Bernardo David, and Roman Oliynykov (2017). "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *CRYPTO 2017, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. LNCS. Springer, Heidelberg, pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12.

Rabin, Tal and Michael Ben-Or (1989). "Verifiable Secret Sharing and Multiparty Protocols with Honest Majority (Extended Abstract)". In: *STOC*, pp. 73–85.

Gordon, S. Dov and Jonathan Katz (2010). "Partial Fairness in Secure Two-Party Computation". In: *EUROCRYPT*, pp. 157–176.

Beimel, Amos, Yehuda Lindell, Eran Omri, and Ilan Orlov (2011). "$1/p$-Secure Multiparty Computation without Honest Majority and the Best of Both Worlds". In: *CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. LNCS. Springer, Heidelberg, pp. 277–296. DOI: 10.1007/978-3-642-22792-9_16.

Alon, Bar and Eran Omri (2016). "Almost-Optimally Fair Multiparty Coin-Tossing with Nearly Three-Quarters Malicious". In: *TCC, Part I*, pp. 307–335.

Ben-Or, Michael, Oded Goldreich, Silvio Micali, and Ronald L. Rivest (1985). "A Fair Protocol for Signing Contracts (Extended Abstract)". In: *ICALP*, pp. 43–52.

Asokan, N., Matthias Schunter, and Michael Waidner (1997). "Optimistic Protocols for Fair Exchange". In: *ACM CCS 97*. Ed. by Richard Graveman, Philippe A. Janson, Clifford Neuman, and Li Gong. ACM Press, pp. 7–17. DOI: 10.1145/266420.266426.

Asokan, N., Victor Shoup, and Michael Waidner (1998). "Optimistic Fair Exchange of Digital Signatures (Extended Abstract)". In: *EUROCRYPT'98*. Ed. by Kaisa Nyberg. Vol. 1403. LNCS. Springer, Heidelberg, pp. 591–606. DOI: 10.1007/BFb0054156.

Garay, Juan A., Markus Jakobsson, and Philip D. MacKenzie (1999). "Abuse-Free Optimistic Contract Signing". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Heidelberg, pp. 449–466. DOI: 10.1007/3-540-48405-1_29.

Micali, Silvio (2003). "Simple and fast optimistic protocols for fair electronic exchange". In: *22nd ACM PODC*. Ed. by Elizabeth Borowsky and Sergio Rajsbaum. ACM, pp. 12–19. DOI: 10.1145/872035.872038.

Dodis, Yevgeniy, Pil Joong Lee, and Dae Hyun Yum (2007). "Optimistic Fair Exchange in a Multi-user Setting". In: *PKC 2007*. Ed. by Tatsuaki Okamoto and Xiaoyun Wang. Vol. 4450. LNCS. Springer, Heidelberg, pp. 118–133. DOI: 10.1007/978-3-540-71677-8_9.

Küpçü, Alptekin and Anna Lysyanskaya (2010). "Usable Optimistic Fair Exchange". In: *CT-RSA 2010*. Ed. by Josef Pieprzyk. Vol. 5985. LNCS. Springer, Heidelberg, pp. 252–267. DOI: 10.1007/978-3-642-11925-5_18.

Kilinç, Handan and Alptekin Küpçü (2016). "Efficiently Making Secure Two-Party Computation Fair". In: *FC 2016*. Ed. by Jens Grossklags and Bart Preneel. Vol. 9603. LNCS. Springer, Heidelberg, pp. 188–207.

Boneh, Dan and Moni Naor (2000). "Timed Commitments". In: *CRYPTO 2000*. Ed. by Mihir Bellare. Vol. 1880. LNCS. Springer, Heidelberg, pp. 236–254. DOI: 10.1007/3-540-44598-6_15.

Garay, Juan A. and Markus Jakobsson (2003). "Timed Release of Standard Digital Signatures". In: *FC 2002*. Ed. by Matt Blaze. Vol. 2357. LNCS. Springer, Heidelberg, pp. 168–182.

Garay, Juan A. and Carl Pomerance (2003). "Timed Fair Exchange of Standard Signatures: [Extended Abstract]". In: *FC 2003*. Ed. by Rebecca Wright. Vol. 2742. LNCS. Springer, Heidelberg, pp. 190–207.

Beaver, Donald and Shafi Goldwasser (1990). "Multiparty Computation with Faulty Majority". In: *CRYPTO'89*. Ed. by Gilles Brassard. Vol. 435. LNCS. Springer, Heidelberg, pp. 589–590. DOI: 10.1007/0-387-34805-0_51.

Asharov, Gilad, Yehuda Lindell, and Hila Zarosim (2013). "Fair and Efficient Secure Multiparty Computation with Reputation Systems". In: *ASIACRYPT 2013, Part II*. Ed. by Kazue Sako and Palash Sarkar. Vol. 8270. LNCS. Springer, Heidelberg, pp. 201–220. DOI: 10.1007/978-3-642-42045-0_11.

Chen, Liqun, Caroline Kudla, and Kenneth G. Paterson (2004). "Concurrent Signatures". In: *EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, Heidelberg, pp. 287–305. DOI: 10.1007/978-3-540-24676-3_18.

Lindell, Yehuda (2009). "Legally Enforceable Fairness in Secure Two-Party Communication". In: *Chicago J. Theor. Comput. Sci.* 2009.

Goldreich, Oded and Ariel Kahan (1996b). "How to Construct Constant-Round Zero-Knowledge Proof Systems for NP". In: *J. Cryptology* 9.3, pp. 167–190.

Let's Encrypt (2017). Available at https://letsencrypt.org/.

Bentov, Iddo, Ariel Gabizon, and Alex Mizrahi (2016). "Cryptocurrencies without proof of work". In: *International Conference on Financial Cryptography and Data Security*. Springer, pp. 142–157.

Damgard, Ivan, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart (2012). *Practical Covertly Secure MPC for Dishonest Majority – or: Breaking the SPDZ Limits*. Cryptology ePrint Archive, Report 2012/642.

Multiparty computation with SPDZ online phase and MASCOT offline phase (2017). Github.
Obscuro (2017). Github.

Chapter 5

Abuse Resistant Law Enforcement Access Systems

5.1 Introduction

Communication systems have increasingly begun to deploy end-to-end (E2E) encryption as a means to secure physical device storage and communications traffic. End-to-end encryption systems differ from traditional link encryption mechanisms such as TLS in that keys are held by endpoints — typically end-user devices such as phones or computers — rather than by service providers. This approach ensures that plaintext data cannot be accessed by service providers or manufacturers, or by attackers who may compromise a provider. Widely-deployed examples include messaging protocols (*Signal Secure Messaging System*; WhatsApp, 2017; *iMessage*), telephony (*FaceTime*), and device encryption (*iCloud security overview*; *Encrypt your data - Pixel Phone Help*), with many systems deployed to billions of users.

The increased adoption of E2E encryption in commercial services has provoked a backlash from the law enforcement and national security communities

around the world. This is based on concerns that encryption will hamper agencies' investigative and surveillance capabilities (Watt, Mason, and Traynor, 2015; Franceschi-Bicchierai, 2014; Barr, 2019). In the United States, the U.S. Federal Bureau of Investigation has mounted a high-profile policy campaign around these issues (*Going Dark*). Similar efforts have resulted in legislative pushes in the United States (Poplin, 2016) and adopted legislation in Australia that requires providers to guarantee access to plaintext in commercial communication systems (Tarabay, 2018).

The proposals made by law enforcement have ignited a debate between technologists and policymakers. Some technical experts have expressed concerns that these proposals, if implemented, will undermine the security offered by encryption systems (Abelson et al., 2015; National Academies of Sciences, Engineering, and Medicine, 2016; Sing, 2020), either by requiring unsafe changes or prohibiting the use of E2E encryption altogether. Law enforcement officials have, in turn, exhorted researchers to develop new solutions that resolve these challenges (Barr, 2019). However, even the basic technical requirements of such a system remain unspecified, complicating both the technical and public debates.

Existing Proposals for Law Enforcement Access. A number of recent and historical technical proposals have been advanced to resolve the technical questions raised by the encryption policy debate (Denning, 1994; Savage, 2018; Bellovin et al., 2018; Wright and Varia, 2018; Tait, 2016; Levy and Robinson, 2018; Bellare and Rivest, 1999). With some exceptions, the bulk of these proposals are variations on the classical *key escrow* (Denning and Branstad,

1996) paradigm. In key escrow systems, one or more trusted authorities retain key material that can be used to decrypt targeted communications or devices.

Technologists and policymakers have criticized key escrow systems (Abelson et al., 2015; Encryption Working Group, 2019; National Academies of Sciences, Engineering, and Medicine, 2018), citing concerns that, without additional protection measures, these systems could be abused to covertly conduct mass surveillance of citizens. Such abuses could result from a misbehaving operator or a compromised escrow keystore. Two recent policy working group reports (Encryption Working Group, 2019; National Academies of Sciences, Engineering, and Medicine, 2018) indicate an expert consensus that extant key escrow proposals provide insufficient protections against abuse, except possibly for the special case of device encryption: a setting where physical possession of a device poses a natural barrier to mass surveillance. Reflecting this consensus, some recent high-profile technical proposals have been proposed for the device encryption case; these include physical countermeasures to mitigate surveillance risks (*e.g.*, tamper-resistant hardware) (Savage, 2018; Bellovin et al., 2018). Expanding similar precautions to messaging or telephony is difficult, as there is no secure hardware equivalent for data in motion.

Mitigating Illicit Surveillance Risk. Escrow-based access proposals suffer from three primary limitations. First, key escrow requires that highly-trusted parties maintain and protect valuable key material, possibly against sophisticated, nation-state supported attackers. Second, if key material is surreptitiously exfiltrated from a keystore, it may be difficult or impossible to detect

its subsequent misuse. This is because escrow systems designed to allow lawful access to encrypted data typically store *decryption keys*, which can be misused without producing any detectable artifact.[1] Finally, these access systems require someone to interface between the digital escrow technology and the non-digital legal system, which raises the possibility of misbehavior by operators. These concerns are not theoretical: wiretapping and surveillance systems have proven to be targets for both nation-state attacks and operator abuse (Bryan-Low, 2006; Nakashima, 2013; Gorman, 2013).

Overcoming these challenges is further complicated by law enforcement's desire to access data that was encrypted before an investigation initiated.[2] In this work we will divide access systems into two categories: *prospective* and *retrospective*. When using a *prospective* system, law enforcement may only access information encrypted to or from suspects after they have been explicitly identified for surveillance: this is analogous to "placing an alligator clip on a wire" in an analog wiretap. A *retrospective* access system allows investigators to decrypt past communications, even those from suspects who were not the target of surveillance when encryption took place. Retrospective access clearly offers legitimate investigators more capabilities, but may also present a greater risk of abuse.

Towards Abuse-Resistant Law Enforcement Access. Some past work has investigated the problem of securing law enforcement access systems against

[1] This contrasts with the theft of *e.g.*, certificates or signing keys, where abuse may produce artifacts such as fraudulent certificates (Nightingale, 2011) that can be detected through Internet-wide surveillance.

[2] For example, several recent investigations requested the unlocking of phones following a crime or terrorist attack (Lichtblau and Goldstein, 2016).

abuse, *e.g.*, (Blaze, 1996; Bellare and Rivest, 1999; Wright and Varia, 2018). Bellare and Rivest (Bellare and Rivest, 1999) proposed a mechanism to build *probabilistic* law enforcement access, in order to mitigate the risk of mass surveillance. Wright and Varia (Wright and Varia, 2018) proposed cryptographic puzzles as a means to increase the financial cost of abuse. These techniques have practical limitations: law enforcement are unlikely to tolerate prohibitive costs or arbitrary barriers to legitimate investigations. Moreover, these proposals do little to enable detection of key theft or prevent more subtle forms of misuse.

In this work we investigate a different approach to this problem. Our goal is to achieve a balance: enforce protections that limit abuse, without placing impractical restrictions on law enforcement capabilities. We develop access systems that weave accountability features into the basic definitions of the system. More concretely, our goal is to construct systems that achieve the following three main features:

Accountability. To prohibit abuse by authorized parties, access systems must enforce specific and *fine-grained* global policies that restrict the types of surveillance may take place. These policies could, for example, encompass limitations on the number of messages decrypted, the total number of targets, and the types of data accessed. They can be agreed upon in advance and made publicly available. This approach ensures that global limits can be developed that meet law enforcement needs, while also protecting the population against surveillance.

Detectability. We require that any unauthorized use of escrow key material

can be detected, either by the public or by authorized auditing parties. Achieving this goal ensures that even fully-adversarial use of escrow key material (*e.g.,* following a key exfiltration) can be detected, and the system's security can be renewed through rekeying.

Operability. At the same time, escrow systems must remain operable, in the sense that honest law enforcement parties should be able to access messages sent through a compliant system, even in settings where dishonest users seek to evade the escrow properties.

The most critical aspect of our work is that we seek to enforce these policies *through the use of cryptography*, rather than relying on correct implementation of key escrow hardware or software, or proper behavior by authorities. Achieving this goal in the challenging setting of retrospective key escrow, where encryption may take place *prior* to any use of escrow decryption keys, is one of the most technically challenging aspects of this work.

Our contributions. More concretely, in this work we make the following contributions.

- **Formalizing security notions for abuse resistant access systems.** We first provide a high-level discussion of the properties required to prevent abuse in a key escrow system, with a primary focus on the general data-in-motion setting: *i.e.,* we do not assume that targets possess trusted hardware. Based on this discussion, we formalize the roles and protocol interface of an Abuse-Resistant Law Enforcement Access System (ARLEAS): a message transmission framework that possesses law

enforcement access capability with strong accountability guarantees. Finally, we provide an ideal functionality $\mathcal{F}_{\text{ARLEAS}}$ in Canetti's Universal Composability framework (Canetti, 2001).

- **A prospective ARLEAS construction from lossy encryption.** As a warmup to our main result, we show how to realize ARLEAS that is restricted to the case of *prospective* access: this restricts the use of ARLEAS such that law enforcement must identify surveillance parameters before a target communication occurs. We show how this construction can be built efficiently from lossy encryption and efficient simulation-sound NIZK. For our example construction, we rely primarily on a variant of a lossy tag-based scheme proposed by Hofheinz (Hofheinz, 2012).

- **A retrospective ARLEAS construction from proof-of-publication ledgers and extractable witness encryption.** As the primary technical contribution of this work, we show how to realize ARLEAS that admits *retrospective* access, while still maintaining the auditability and detectability requirements of the system. The novel idea behind our construction is to use secure *proof-of-publication ledgers* to condition cryptographic escrow operations. This paradigm of proof-of-publication ledgers has recently been explored in several works (Choudhuri et al., 2017; Goyal and Goyal, 2017; Kaptchuk, Green, and Miers, 2019; Scafuro, 2019). Such ledgers may be realized using recent advances in consensus networking, a subject that has been the subject of a significant amount of research.

- **Evaluating minimal assumptions for retrospective systems.** Finally, we investigate the *minimal* assumptions for realizing retrospective access

in accountable law enforcement access system. As a concrete result, we present a lower-bound proof that any protocol realizing the ARLEAS functionality implies the existence of extractable Witness Encryption scheme for some language $\mathcal{L}$ which is related to the ledger functionality and policy functions of the system. While this proof does not imply that all ARLEAS realizations require extractable Witness Encryption (*i.e.,* it may be possible to construct languages that have trivial EWE realizations), it serves as a guidepost to illustrate the barriers that researchers may face in seeking to build accountable law enforcement access systems.

5.1.1 Towards Abuse Resistance

In this work we consider the problem of constructing secure message transmission (SMT) protocols, as formalized in the UC framework by Canetti (Canetti, 2001; Canetti, Krawczyk, and Nielsen, 2003). To these protocols, we add new interfaces that allow law enforcement to access messages under specific conditions. Prior to introducing formal definitions for accountable access, we first discuss several of the security properties required of such a system.

Abuse-Resistant Law Enforcement Access System. At setup time an ARLEAS system is parameterized by two functions, which we refer to as the *global policy function* $p(\cdot)$ and *warrant transparency function* $t(\cdot)$. The system is comprised of three types of party:

1. **Users:** Users employ a secure message transmission protocol to exchange messages with other users. From the perspective of these users,

this system acts like a normal messaging service, with the ability to view public audit log information about the use of warrants on information send through the system.

2. **Law Enforcement:** Law enforcement parties are responsible for initiating surveillance and accessing decrypted messages. This involves determining the scope of a surveillance request, obtaining a digital warrant, and then accessing the resulting data.

3. **Judiciary:** The final class of parties act as a check on law enforcement, determining whether a surveillance request meets the necessary legal requirements cause. In our system, any surveillance request must be approved by a judge before it is activated on the system. In our model we assume a single judge per system, though in practice this functionality can be distributed.

In addition to these parties, our proposals assume the existence of a public broadcast channel, such as an append-only ledger. While this ledger may be operated by a centralized party, in practice we expect that such systems will be highly-distributed, *e.g.* using blockchain or consensus network techniques.

Operation. To initiate a surveillance request, law enforcement must first identify a specific class of messages (*e.g.* by metadata or sender); it then requests a surveillance *warrant* from a judge. The judge reviews the request and authorizes or rejects the request. If the judge produces an authorized warrant, law enforcement must take a final step to *activate* the warrant in order to initiate surveillance. This activation process is a novel element of

an abuse-resistant access scheme, and it is what allows for the detection of misbehavior. To enforce this, we require that activation of a warrant $\hat{w}$ results in the publication of some transparency data that is viewable by all parties in the system: the amount and nature of the data to be published is determined by a global transparency function $t(\hat{w})$.

For a system to be considered *abuse-resistant*, it must satisfy the following intuitive properties:

Warrant Security. A system must provide strong cryptographic security against attackers who are not authorized to receive messages (*i.e.* everyone except the legitimate sender and receiver). Moreover, if a judge has not issued a warrant that allows a message to be decrypted, attackers should be unable to learn any information about the content of messages, excluding public metadata and length.

Accountability. Surveillance requests must always obey a set of global limits defined by the public *policy function* which was chosen during system setup. Only surveillance requests that satisfy the limits defined by this function can be approved within the system, even in cases where all escrow authorities (*i.e.* law enforcement and judges) collude. This flexible mechanism allows operators to prevent certain forms of surveillance even in cases where all key material becomes compromised.

Detectability. To enable detection of abuse, or theft of key material, we require that every warrant activation must produce a public artifact that can be detected by the public or by authorized auditing parties. The

simplest form of detection would simply require law enforcement to publish the full contents of each warrant request; however, doing so might imperil the secrecy of ongoing investigations. As a compromise between these outcomes, we allow system operators to define a public *warrant leakage function* $L(\cdot)$ at system setup. To activate a warrant W, law enforcement must publish $L(W)$ to all parties in the system. This publication must occur even in cases where all escrow parties collude.

Target anonymity. To preserve the integrity of investigations, users should learn no information about the contents of a warrant beyond what is revealed by the leakage function.

Escrow verifiability. Escrow authorities must be assured that the access system will work properly. Because senders can always behave dishonestly (*e.g.* using an alternative encryption mechanism to transmit messages), this guarantee cannot be enforced for all possible sender behavior. Instead, we mandate a weaker property that we call *escrow verifiability*: this ensures that recipients and/or service providers can publicly detect and filter non-compliant messages. This ensures that any compliant message will be accessible by escrow authorities under appropriate circumstances.

In Section 5.3 we formalize this intuition and present concrete security definitions for the ARLEAS concept.

5.1.2 Intuition

We now present an overview of the key technical contributions of this work. We will consider this in the context of secure message transmission systems, which can be generalized to the setting of encrypted storage.

Accountability From Ledgers. For an abuse-resistant access system the most difficult properties to satisfy are accountability and detectability. Existing solutions attempt to achieve this property by combining auditors and key escrow custodians; in order to retrieve key material that facilitates decryption, law enforcement must engage with an auditor. This solution, however, does not account for dishonest authorities, and is therefore vulnerable to covert key exfiltration and collusion. In our construction, we turn to public ledgers — a primitive that can be realized using highly-decentralized and auditable systems — as a way to reduce these trust assumptions.

Ledgers have the property that any party can access their content. Importantly, they also have the property that any parties can be convinced that other parties have access to these contents. Thus, if auditing information is posted on a ledger, all parties are convinced that that information is truly public. We note that using ledgers in this way is fundamentally different than prior work; our ledger is a public functionality that does not need to have any escrow secrets. As such, if it is corrupted, there is no private state that can be exploited by an attacker.

Warm up: Prospective ARLEAS. To motivate our main construction, we first consider the simpler problem of constructing a *prospective* access system, one

that is capable of accessing messages that are sent subsequent to a warrant becoming active. For our practical construction, we make a further simplifying assumption that law enforcement will target *specific parties* for surveillance, rather than targeting messages by a more powerful and complex predicate — a subject we discuss in later sections.

A key aspect of this construction is that we consider a relatively flexible setting where parties have network access, and can receive periodic communications from escrow system operators prior to transmitting messages. We employ a public ledger for transmission of these messages, which provides an immutable record as well as a consistent view of these communications. The goal in our approach is to ensure that escrow updates embed information about the specific of surveillance warrants that are active, while ensuring that even corrupted escrow parties cannot abuse the system to misuse the surveillance apparatus.

Escrow lossy encryption. The basic intuition of our approach is to construct a "dual-trapdoor" public-key encryption system (Bresson, Catalano, and Pointcheval, 2003) that senders can use to encrypt messages to specific parties. This scheme is designed to ensure that ciphertexts can be decrypted by the intended recipient using a normal secret key, while allowing decryption by the escrow authorities only if the recipient is under active surveillance. A feature of this scheme is that for all recipient not so targeted, it should leak no additional information about the plaintext.

We can realize this construction using an efficient tag-based variant of lossy encryption (Peikert and Waters, 2008; Bellare, Hofheinz, and Yilek,

2009; Hemenway et al., 2011; Hofheinz, 2012) that we call *none-but-N* lossy-tag-based encryption, or LTE. In this scheme, key generation creates public parameters R with respect to the set of user identifiers (tags) $\mathcal{T}$ that are under active surveillance, along with a secret decryption key. When encrypting a message, a sender encrypts under both the recipient's public key and tag, along with R.

Our scheme must satisfy three main security properties. First, if the parameter generation process is run honestly with some set of user identifiers $\mathcal{T}$ and any (even biased) random coins r, then no adversarial escrow agent should be able to retrieve the message M if the receiver is not in $\mathcal{T}$. Second, to ensure that escrow decryption is possible, we require that adversarial encryptors cannot produce a ciphertext that appears correctly formatted but does not admit decryption. Finally, to provide anonymity for the escrow authorities, we require that R must at least computationally hide the set of users that are being target for surveillance, i.e. no efficient adversary given R should be able to recover any information about $\mathcal{T}$ beyond for the size of its description. In Section 2.2.4.1 we discuss candidate constructions for LTE schemes, and discuss generalizing this construct to allow the tag set to be replaced by a general predicate.

Building prospective ARLEAS from LTE. Given an appropriate lossy tag-based encryption scheme, the remainder of the ARLEAS construction proceeds as follows. The global parameters of the scheme are created at setup: these include a public verification key for the judge(s) presiding over the system, as well as a pair of global leakage and transparency functions L, P agreed on by

system participants.

When a law enforcement agency wishes to create a warrant w that embeds a specific set of users $\mathcal{T}$ to be surveilled, it first runs the LTE parameter generation algorithm to obtain R on $\mathcal{T}$ and stores the corresponding secret key. It now calculates transparency information $T \leftarrow L(w)$. Law enforcement next contacts the judge(s) to obtain a signature over the warrant, and proceeds to generate a NIZK $\hat{\pi}$ that the following statements hold: (1) the prover possesses a signature on w from the judge(s), (2) the parameter R was correctly generated with respect to the user-set $\mathcal{T}$ specified in the warrant, (3) the warrant w does not violate the global policy function P, and (4) the transparency information T was calculated correctly. Finally, it transmits $(R, T, \hat{\pi})$ to a global ledger. Each participant in the system must ensure that this message was correctly published, and verify the proof $\hat{\pi}$. If this proof verifies correctly, the participants will accept the new parameter R and employ this value for all subsequent messages they encrypt.

A critical security property of this system is that, even if law enforcement and judges collude (*e.g.,* if both parties become catastrophically compromised), users retain the assurance that no warrant can be issued in violation of the global policy P. Moreover, even in this event, the publication of a transparency record T ensures that every warrant activated in the system produces a detectable artifact that can be used to identify abuse.[3]

[3]The flexible nature of the transparency function t ensures that these records can contain both publicly-visible records (*e.g.,* a quantized description of the user set size, as well as private information that can be encrypted to auditors.

From Prospective to Retrospective. The major limitation of the ARLEAS construction above is that it is fundamentally restricted to the case of *prospective* access. Abuse-resistance derives from the fact that "activation" of a warrant results in a distribution of fresh key material to participants, and each new escrow parameter targets renders only a subset of communications accessible. A second drawback of the prospective protocol is that it requires routine communication between escrow authorities and the users of the system, which may not be possible in all settings.

Updating these ideas to provide *retrospective* access provides a stark illustration of the challenges that occur in this setting. In the retrospective setting, the space of targeted communications is unrestricted at the time that encryption takes place. By the time this information is known, both sender and recipient may have completed their interaction and gone offline. Using some traditional, key based solution to this problem implies the existence of powerful master decryption keys that can access *every* ciphertext sent by users of the system. Unfortunately, granting such power to any party (or set of parties) in our system is untenable if we assume that this key material may be compromised. The technical challenge in the retrospective setting is to find an alternative means to enable decryption, such that decryption is only possible on the conditions that (1) a relevant warrant has been issued that is compliant with the global policy function, (2) a detectable artifact has been made public. This mechanism must remain secure even when encryption occurs significantly before the warrant is contemplated.

Ledgers as a cryptographic primitive. A number of recent works (Choudhuri

et al., 2017; Goyal and Goyal, 2017; Kaptchuk, Green, and Miers, 2019; Choudhuri, Goyal, and Jain, 2019; Scafuro, 2019) have proposed to use public ledgers as a means to *condition* cryptographic operations on published events. This paradigm was initially used by Choudhuri *et al.* (Choudhuri et al., 2017) to achieve fairness in MPC computations, while a variant was proposed by Goyal and Goyal (Goyal and Goyal, 2017) to construct one-time programs without the need for trusted hardware. Conceptually, these functionalities all allow decryption or program execution to occur only *after* certain information has been made public. This model assumes the existence of a secure global ledger $\mathcal{L}^{\mathsf{Verify}}$ that is capable of producing a publicly-verifiable proof π that a value has been made public on the ledger. In principle, this ledger represents an alternative form of "trusted party" that participates in the system. However, unlike the trusted parties proposed in past escrow proposals (Denning, 1994), ledgers do not store any decryption secrets. Moreover, recent advances in consensus protocols, and particularly the deployment of proof-of-work and proof-of-stake cryptocurrency systems. *e.g.,* (Kiayias et al., 2017; David et al., 2018; Gazi, Kiayias, and Zindros, 2019; Pietrzak, 2019; Boneh et al., 2018), provide evidence that these ledgers can be operated safely at large scale.

Following the approach outlined by Choudhuri *et al.* (Choudhuri et al., 2017), we make use of the ledger to *conditionally encrypt* messages such that decryption is only possible following the verifiable publication of the leakage function evaluated over a warrant on the global ledger. For some forms of general purpose ledgers that we seek to use in our system, this can be accomplished using extractable witness encryption (EWE) (Boyle, Chung,

and Pass, 2014).[4] EWE schemes allow a sender to encrypt under a statement such that decryption is possible only if the decryptor knows of a witness ω that proves that the statement is in some language $\mathcal{L}$, where $\mathcal{L}$ parameterizes the scheme. While candidate schemes for witness encryption are known for specific languages (like hash proof systems (Cramer and Shoup, 2002; Garg et al., 2012)), extractable witness encryption for general languages is unlikely to exist (Garg et al., 2014).

Building Retrospective ARLEAS from EWE. Our prospective ARLEAS construction assumes the existence of a global ledger that produces verification proofs π that a warrant has been published to a ledger. As mentioned before, we aim to condition law enforcement access on the issuance of a valid warrant and the publication of a detectable artifact. Thus, in this construction, a sender encrypts each message under a statement with a witness that shows evidence that these conditions have been met. This language reasons over (1) the warrant transparency function, (2) a function determining the relevance of the warrant to ciphertext, (3) the global policy function, (4) the judge's warrant approval mechanism, and (5) the ledger's proof of publication function. Unlike the prospective solution described above, there is no limitation on the structure of warrants; the relevance of a warrant to a particular message may be structured as an arbitrary predicate included in the EWE language.

On the Requirement of Extractable Witness Encryption: We justify the use of extractable witness encryption in our construction by showing that the existence of a secure protocol realizing retrospective ARLEAS implies the

[4] Using the weaker witness encryption primitive may be possible if the ledger produces *unique* proofs of publication.

existence of a secure extractable witness encryption scheme for a related language that is deeply linked to the ARLEAS protocol. Intuitively, the witness for this language should serve as proof that the protocol has been correctly executed; law enforcement should be able to learn information about a message if and only if the accountability and detectability mechanisms have been run. For the concrete instantiation of retrospective we give in Section 5.5, this would include getting a valid proof of publication from the ledger. If the protocol is realized with a different accountability mechanism, the witness encryption language will reason over that functionality. No matter the details of the accountability mechanism, we note that it should be difficult for law enforcement to locally simulate the mechanism. If it were computationally feasible, then law enforcement would be able to circumvent the accountability mechanism with ease.

Importantly, this proof does not mean that extractable witness encryption for *general languages* is required to realize retrospective ARLEAS. Instead, we require extractable witness encryption that can reason over the accountability mechanisms. Currently, constructions of extractable witness encryption for non-trivial functionalities are not known, and there are implausibility results regarding the existence of the primitive for general languages (Garg et al., 2014). This indicates that realizing ARLEAS in the retrospective setting will be extremely difficult in practice.

5.2 Related work

The past decade has seen the start of academic work investigating the notion of accountability for government searches. Bates et al. (Bates et al., 2012) focus specifically on CALEA wiretaps and ensuring that auditors can ensure law enforcement compliance with court orders. In the direct aftermath of the Snowden leaks, Segal et al. (Segal, Ford, and Feigenbaum, 2014) explored how governments could accountably execute searches without resorting to dragnet surveillance. Liu et al. (Liu, Ryan, and Chen, 2014) focus on making the number of searches more transparent, allow democratic processes to balance social welfare and individual privacy. Kroll et al. (Kroll, Felten, and Boneh, 2014; Kroll et al., 2014) investigate different accountability mechanisms for key escrow systems, but stop short of addressing end-to-end encryption systems and the collusion problems we address in this work. Backes considered anonymous accountable access control (Backes, Camenisch, and Sommer, 2005). Goldwasser and Park (Goldwasser and Park, 2017) investigate similar notions with the limitation that policies themselves may be secret, due to national security concerns. Frankle *et al.* (Frankle et al., 2018) make use of ledgers to get accountability for search procedures, but their solution cannot be extended to the end-to-end encryption setting. Wright and Varia (Wright and Varia, 2018) give a construction that uses cryptographic puzzles to imposes a high cost for law enforcement to decrypt messages. Servan-Schreiber and Wheeler (Servan-Schreiber and Wheeler, 2019) give a construction for accountability that randomly selects custodians that law enforcement must access to decrypt a message. Panwar *et al.* (Panwar et al., 2019) attempt to

integrate the accountability systems closely with ledgers, but do not use the ledgers to address access to encryption systems. Finally, Scafuro (Scafuro, 2019) proposes a closely related concept of "break-glass encryption" and give a construction that relies on trusted hardware.

5.3 Definitions

In this section we present definitions for abuse-resistant law enforcement access systems.

5.3.1 Abuse-Resistant Law Enforcement Access Systems

We now formally define the notion of an Abuse-Resitant Law Enforcement Access System (ARLEAS). This is a form of message transmission system that supports accountable law enforcement access. We base our definitions on the Secure Message Transmission ($\mathcal{F}_{\mathsf{SMT}}$) notion originally introduced by Canetti (Canetti, 2001). Indeed, our systems can be viewed as an extension of a multi-message SMT functionality (Canetti, Krawczyk, and Nielsen, 2003), with added escrow capability.

System parameters. An ARLEAS system is parameterized by three functions, which are selected during setup:

- $t(w)$: the *transparency function* takes as input a warrant w and outputs specific information about the warrant and the use of the backdoor that can be published to the general public. One property that we require from the leakage functionality is that it is binding for w.

- $p(w)$: the *global policy* function takes as input a warrant w and outputs TRUE if this warrant is allowed by the system.
- $i(w, \mathsf{aux})$: takes as input a warrant w and per-message metadata aux. It outputs *true* if aux is in scope of w for surveillance.

Parties. An ARLEAS is an interactive protocol run between several parties. We describe these as:

- User P_i: This is a user of the end-to-end encrypted service or application. They interact with the system by simply sending and receiving messages.
- Judge P_J: The judge is responsible for determining the validity of a search and issuing search warrants to law enforcement. The judge interacts with the system by receiving warrant requests and choosing to deny or approve the request.
- LawEnforcement P_{LE}: Law enforcement is responsible for conducting searches, pursuant to a valid warrant authorized by a judge. Law enforcement interacts with the system by requesting warrants from the judge and collecting the plaintext messages relevant to their investigations.

A concrete ARLEAS function is initiated through a trusted setup phase, in which the parties agree on three functions: a global policy function p, a transparency function t and a targeting function i, whose purposes are described above.

A UC ideal functionality. We present a formal definition of the ideal functionality $\mathcal{F}_{\text{ARLEAS}}$ in Figure 5.1. This functionality is parameterized by the three functions p, t and i as well as a mode parameter that determines whether the system supports retrospective or prospective surveillance, $\mathsf{mode} \in \{\mathsf{pro}, \mathsf{ret}\}$.

Ideal World. For any ideal-world adversary $\mathcal{S}$ with auxiliary input $z \in \{0,1\}^*$, input vector x, and security parameter λ, we denote the output of the ideal world experiment by $\mathbf{Ideal}_{\mathcal{S},\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{mode}}}(1^\lambda, x, z)$.

Real World. The real world protocol starts with the adversary $\mathcal{A}$ selecting a subset of the parties to compromise $\mathcal{P}^{\mathcal{A}} \subset \mathcal{P}$, where $\mathcal{P}^{\mathcal{A}} \subset \{\{P_i\}, \{P_{\text{LE}}\}, \{P_{\text{LE}}, P_J\}\}$. We limit the subsets of parties that can be compromised to these cases, because any other combination doesn't make much sense in this setting. In our model we don't worry about the other combinations, because if both P_i and P_j would be corrupted, there is nothing stopping them from not using the system. Moreover, we also don't consider the case where P_J is the only corrupted party, as it could only block the whole protocol, but they don't really have an output. Finally, we consider the view of the system participants to always be compromised, because all other parties are also part of the general public. All parties engage in a real protocol execution Π, the adversary $\mathcal{A}$ sends all messages on behalf of the corrupted parties and can choose any polynomial time strategy.

In a real world protocol we assume that communication between P_i and P_j happens over a transparent channel, meaning all other parties are able to receive all communication. We make this choice to simplify the protocol and security proofs, in the real world, this can be modeled with a service provider

Functionality $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{mode}}$

The ideal functionality is parameterized by a metadata generation function $\ell : \{0,1\}^* \to \{0,1\}^*$, the transparency function t, the global policy function p, and the target functionality i. The three latter functions are as defined above. We have several parties:

- $P_1, \ldots, P_n$: participants in the system
- P_J: the generator of a warrant
- P_{LE}: Law enforcement that can read the message given a valid warrant

Send Message: Upon receiving a message $(\mathsf{SendMessage}, \mathsf{sid}, P_j, m, b)$ where $b \in \{0,1\}$ from party P_i, the trusted party computes the metadata for that as message as $\mathsf{aux} \leftarrow \ell(P_j \| m)$ and sends $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux})$ to the adversary. If (sid, c) is received from the adversary,

- If $b = 0$, send $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ to P_i and send $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, 0)$ to P_{LE}.
- If $b = 1$ and there is no entry w in the active warrant table W_{active} send $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ to P_i and P_j, and send $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c)$ to P_{LE}.
- If $b = 1$ and there is an entry w in the active warrant table W_{active} send $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ to P_i, P_j, and P_{LE}.

Finally, store $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ in the message table M.

Request Warrant: Upon receiving a message $(\mathsf{RequestWarrant}, \mathsf{sid}, w)$ from P_{LE}, the trusted party first checks if $p(w) = 1$,responding with $\perp$ and aborting if not. Otherwise, the trusted party sends $(\mathsf{ApproveWarrant}, w)$ to P_J. If P_J responds with $(\mathsf{Disapprove})$, the trusted functionality sends $\perp$ to P_{LE}. If P_J responds with $(\mathsf{Approve})$, the trusted functionality sends $(\mathsf{Approve})$ to P_{LE}, and stores the entry w in the issued warrant table W_{issued}.

Activate Warrant: Upon receiving a message $(\mathsf{ActivateWarrant}, \mathsf{sid}, w)$ from P_{LE}, the trusted party checks to see if w is in W_{issued}, responding with $\perp$ and aborting if not. If w is in W_{issued}, the trusted functionality adds the entry w to the active warrant table W_{active}, computes $t(w)$, and sends $(\mathsf{NotifyWarrant}, t(w))$ to all parties and the adversary.

Verify Warrant Status: Upon receiving message $(\mathsf{VerifyWarrantStatus}, \mathsf{sid}, c, \mathsf{aux}, w)$ from P_{LE}, if $\mathsf{mode} = \mathsf{pro}$, it responds with $\perp$ and aborts. Otherwise, if $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ is in M and w is in W_{active} such that $i(w, \mathsf{aux}) = 1$, the trusted party returns 1. Otherwise, it returns 0.

Access message: Upon receiving message $(\mathsf{AccessData}, \mathsf{sid}, c, \mathsf{aux}, w)$ from P_{LE}, if if $\mathsf{mode} = \mathsf{pro}$, it responds with $\perp$ and aborts. Otherwise, if $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ is in M and w is in W_{active} such that $i(w, \mathsf{aux}) = 1$, the trusted party returns m. Otherwise, it returns $\perp$.

Figure 5.1: Ideal functionality for an Abuse Resistant Law Enforcement Access System.

relaying messages between P_i and P_j, this service provider will always comply with law enforcement and hand over encrypted messages when there is a valid warrant. Moreover, this service provider could also validate if messages are well-formed to make sure P_i and P_j follow the real protocol.

For any adversary $\mathcal{A}$ with auxiliary input $z \in \{0,1\}^*$, input vector x, and security parameter λ, we denote the output of Π by $\mathbf{Real}_{\mathcal{A},\Pi}(1^\lambda, x, z)$.

Definition 16 *A protocol Π is said to be a secure ARLEAS protocol computing $\mathcal{F}_{ARLEAS}^{\ell,t,p,i,\mathsf{mode}}$ if for every PPT real-world adversary $\mathcal{A}$, there exists an ideal-world PPT adversary $\mathcal{S}$ corrupting the same parties such that for every input x and auxiliary input z it holds that*

$$\mathbf{Ideal}_{\mathcal{S},\mathcal{F}_{ARLEAS}^{\ell,t,p,i,\mathsf{mode}}}(1^\lambda, x, z) \overset{c}{\approx} \mathbf{Real}_{\mathcal{A},\Pi}(1^\lambda, x, z)$$

Protocol $\mathbf{Real}_{\mathcal{A},\Pi}(1^\lambda, x, z)$

$\mathbf{Real}_{\mathcal{A},\Pi}(1^\lambda, x, z)$ is parameterized by the protocol $\Pi = (\mathsf{Setup}, \mathsf{SendMessage}, \mathsf{RequestWarrant}, \mathsf{ActivateWarrant}, \mathsf{VerifyWarrantStatus}, \mathsf{AccessMessage})$ and a variable $\mathsf{mode} \in \{\mathsf{pro}, \mathsf{ret}\}$.

1. When $\mathbf{Real}_{\mathcal{A},\Pi}(1^\lambda, x, z)$ is initialized, the all parties engage in the interactive protocol $\Pi.\mathsf{Setup}$
2. When P_i is activated with $(\mathsf{SendMessage}, \mathsf{sid}, P_j, m, 1)$, parties P_i, P_j, and P_{LE} engage in the interactive protocol $\Pi.\mathsf{SendMessage}$. P_{LE} should learn some aux about the message.
3. When P_i is activated with $(\mathsf{SendMessage}, \mathsf{sid}, P_j, m, 0)$, parties P_i, and P_{LE} engage in the interactive protocol $\Pi.\mathsf{SendMessage}$ (with P_j not getting output). P_{LE} should learn some aux about the message.
4. When P_{LE} is activated with $(\mathsf{RequestWarrant}, \mathsf{sid}, \hat{w})$, parties P_{LE} and P_J engage in the interactive protocol $\Pi.\mathsf{RequestWarrant}$.
5. When P_{LE} is activated with $(\mathsf{ActivateWarrant}, \mathsf{sid}, w)$, all parties engage in the interactive protocol $\Pi.\mathsf{ActivateWarrant}$.
6. When P_{LE} is activated with $(\mathsf{VerifyWarrantStatus}, \mathsf{sid}, c, \mathsf{aux}, w)$, if $\mathsf{mode} = \mathsf{pro}$, P_{LE} returns $\perp$. Otherwise, P_{LE} calls the non-interactive functionality $\Pi.\mathsf{VerifyWarrantStatus}(c, \mathsf{aux}, w)$
7. When P_{LE} is activated with $(\mathsf{AccessData}, \mathsf{sid}, c, \mathsf{aux}, w)$, if $\mathsf{mode} = \mathsf{pro}$, P_{LE} returns $\perp$. Otherwise, P_{LE} calls the non-interactive functionality $\Pi.\mathsf{AccessMessage}(c, \mathsf{aux}, w)$

Figure 5.2: The real world experiment for a protocol implementing $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{mode}}$

5.4 Prospective Solution

In this section we describe an efficient prospective ARLEAS scheme. Recall that in the prospective case, warrants must be activated *before* targets perform encryption. The implication of this setting is that new key material can be sampled and distributed to clients each time law enforcement updates the set of active warrants.

The need for accountability restricts us from using many other cryptographic tools. For example, Identity Based Encryption (IBE) systems provide a natural form of key escrow. Unfortunately, in a standard IBE scheme this key escrow is absolute: the master authority can decrypt any ciphertext in the system. To enable limited surveillance, we require a system in which only a subset of communications will be targeted at any time epoch, and no additional information about *non-targeted* plaintexts will be revealed to the authorities.

Rather than using a fully-escrowed scheme such as IBE, we instead make use of the Π_{LTE} scheme presented in Section 2.2.4.1. Depending on the inputs to the key generation algorithm, encrypting to certain recipients will result in a ciphertext that can be decrypted by law enforcement, while encrypting to other recipients will result in a ciphertext that is statistically independent of the message. Importantly, the scheme we present in Section 2.2.4.1 supports efficient proofs that the public parameters were correctly constructed. We make use of this feature in our construction.

5.4.1 UC-Realizing Prospective Abuse-Resistant Law Enforcement Access Systems

Our construction makes use of a CCA secure encryption system Π_{Enc}, a SUF-CMA secure signature scheme Π_{Sign}, a lossy-tag encryption scheme Π_{LTE} (presented in Section 2.2.4.1). We now present a protocol $\pi^{t,p,i}_{\text{PRO}}$ in the $\mathcal{L}^{\mathsf{Verify}}$, $\mathcal{F}_{CRS}^{\Pi_{\text{NIZK}}.\mathsf{ZKSetup}}$, $\mathcal{F}_{AUTH}$ hybrid model. Our scheme consists of the following interactive protocols:

$\pi^{t,p,i}_{\mathbf{PRO}}.\mathsf{Setup}$**:**

- All users send (CRS) to $\mathcal{F}_{CRS}^{\Pi_{\text{NIZK}}.\mathsf{ZKSetup}}$ to retrieve the common reference string for the NIZK scheme.
- P_j computes $(pk_j, sk_j) \leftarrow \Pi_{\text{Enc}}.\mathsf{KeyGen}(1^\lambda)$ and selects a unique tag id_j and sends (pk_j, id_j) to P_{LE} and to each P_i via $\mathcal{F}_{AUTH}$.
- P_J computes $(pk_{\text{sign}}, sk_{\text{sign}}) \leftarrow \Pi_{\text{Sign}}.\mathsf{KeyGen}(1^\lambda)$ and send pk_{sign} to all other users via $\mathcal{F}_{AUTH}$.
- P_{LE} runs $\pi^{t,p,i}_{\text{PRO}}.\mathsf{ActivateWarrant}$ with as input an empty set $\emptyset$ as the valid warrants.

$\pi^{t,p,i}_{\mathbf{PRO}}.\mathsf{SendMessage}$:

- The sender P_i computes the ciphertext $(c_1, c_2, \pi, \mathsf{aux})$ as follows, and sends it to P_j and P_{LE} via $\mathcal{F}_{AUTH}$:
  - Send $(\mathsf{GetCounter})$ to $\mathcal{L}^{\mathsf{Verify}}$ and receive the current counter t. Then query $\mathcal{L}^{\mathsf{Verify}}$ on (GetVal, t) to receive the latest posting $(t, x, \pi_{\mathsf{publish}})$.

Parse x as $(params, \pi, \mathbb{L})$. If $\Pi_{\text{NIZK}}.\mathsf{ZKVerify}(params, \mathbb{L}, \pi) = 0$ or $\mathcal{L}^{\mathsf{Verify}}.\mathsf{Verify}(t \| (params, \pi, \mathbb{L}), \pi_{\mathsf{publish}}) = 0$ return $\perp$ and halt.

- $c_1 \leftarrow \Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, m; r_1)$, where $r_1 \xleftarrow{\$} \{0,1\}^\lambda$
- $c_2 \leftarrow \Pi_{\text{LTE}}.\mathsf{Enc}(params, id_j, m; r_2)$ where $r_2 \xleftarrow{\$} \{0,1\}^\lambda$
- Use $\Pi_{\text{NIZK}}.\mathsf{ZKProve}$ to compute π such that

$$\pi \leftarrow NIZK \left\{ (m, r_1, r_2) : \begin{array}{r} c_1 = \Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, m; r_1) \wedge \\ \\ c_2 = \Pi_{\text{LTE}}.\mathsf{Enc}(params, id_j, m; r_2) \end{array} \right\}$$

- Create $\mathsf{aux} \leftarrow id_j$

- Upon receiving c from P_i, P_j calls $\pi^{t,p,i}_{\text{PRO}}.\mathsf{VerifyMessage}$ on c. If the output is 1, then recover the message as $m \leftarrow \Pi_{\text{Enc}}.\mathsf{Dec}(sk_j, c_2)$

$\pi^{t,p,i}_{\mathbf{PRO}}.\mathsf{VerifyMessage}$:

- Any party parses $(c_1, c_2, \pi, \mathsf{aux}) \leftarrow c$ and verifies that π and aux are correct, aborting if the output is 0. Otherwise, output 1.

$\pi^{t,p,i}_{\mathbf{PRO}}.\mathsf{RequestWarrant}$**:**

- P_{LE} sends $(\mathsf{RequestWarrant}, id)$ to P_J. P_J then either decides to send $(\mathsf{Disapprove})$ to P_{LE} and halt or executes the following:
  - Verify that $p(id) = 1$. If not send $(\mathsf{Disapprove})$ to P_{LE} and abort.
  - $\pi \leftarrow \Pi_{\text{Sign}}.\mathsf{Sign}(sk_{\text{sign}}, id)$
  - Send the signed warrant (id, π) to P_{LE}

$\pi^{t,p,i}_{\mathbf{PRO}}$.ActivateWarrant**:**

- P_{LE} adds the new warrant w to the set of valid warrants $\mathcal{W}$. It then extracts the set of tags $\mathcal{T} \leftarrow \{id | (id, \sigma) \in \mathcal{W}\}$
- Compute $(params, msk) \leftarrow \Pi_{\text{LTE}}.\mathsf{KeyGen}(1^\lambda, \mathcal{T}; r)$ for $r \xleftarrow{\$} \{0,1\}^\lambda$
- Compute $\mathbb{L} \leftarrow \{t(w) | w \in \mathcal{W}\}$
- Use $\Pi_{\text{NIZK}}.\mathsf{ZKProve}$ to compute π such that

$$\pi \leftarrow NIZK\{(\mathcal{W}, \mathcal{T}, msk, r) : \mathbb{L} = \{t(w) | w \in \mathcal{W}\} \wedge \mathcal{T} \leftarrow \{id | (id, \sigma) \in \mathcal{W}\} \wedge$$

$$(params, msk) \in \Pi_{\text{LTE}}.\mathsf{KeyGen}(1^\lambda, \mathcal{T}; r) \wedge$$

$$\forall (id, \pi) \in \mathcal{W}, \Pi_{\text{Sign}}.\mathsf{Verify}(pk_{\text{sign}}, id, \pi) = p(id) = 1\}$$

- Send $(\mathsf{Post}, (params, \pi, \mathbb{L}))$ to $\mathcal{L}^{\mathsf{Verify}}$ and receiver $(t, x, \pi_{\mathsf{publish}})$.

Theorem 5 *Assuming a CCA secure public key encryption scheme Π_{Enc}, a SUF-CMA secure signature scheme Π_{Sign}, a NIZK scheme Π_{NIZK}, and a lossy-tag encryption scheme Π_{LTE}, $\pi^{t,p,i}_{PRO}$ UC-realizes $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{ARLEAS}$ initialized in prospective mode in the $\mathcal{L}^{\mathsf{Verify}}$, $\mathcal{F}^{\Pi_{NIZK}.\mathsf{ZKSetup}}_{CRS}$, $\mathcal{F}_{AUTH}-$hybrid model.*

5.4.1.1 Proof of Security

Proof. We prove that the above construction securely realizes $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{\text{ARLEAS}}$ by showing that there does not exist a distinguisher $\mathcal{Z}$ that can distinguish between an interaction with the ideal functionality and a simulator $\mathcal{S}$ and the

real protocol $\pi_{\text{PRO}}^{t,p,i}$. We define the interaction with the real protocol as follows: The experiment is initialized with N users $P_1, \ldots P_N$, law enforcement P_{LE} and a judge P_J. The adversary $\mathcal{A}$ chooses a subset of these users to corrupt. Then, users run $\pi_{\text{PRO}}^{t,p,i}.\mathsf{Setup}$, with $\mathcal{A}$ controlling the actions of the corrupted parties. Honest users then, according to their arbitrary strategy, run $\pi_{\text{PRO}}^{t,p,i}.\mathsf{SendMessage}$ to exchange messages with other users. Law enforcement interacts with the judge to get warrants via $\pi_{\text{PRO}}^{t,p,i}.\mathsf{RequestWarrant}$ and uses $\pi_{\text{PRO}}^{t,p,i}.\mathsf{ActivateWarrant}$ to start surveilling a user. Honest parties follow an arbitrary strategy, but follow the protocol and corrupted parties are controlled by the adversary.

We start our proof by first considering the case where a single user P_i is corrupted.

P_i is corrupted.

We begin by showing that $\pi_{\text{PRO}}^{t,p,i}$ UC-realizes the $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{pro}}$ when a user P_i is compromised. We construct the simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ generates the common reference string for the NIZK scheme directly, and stores the trapdoor τ. $\mathcal{S}$ runs $(pk_{\text{sign}}, sk_{\text{sign}}) \leftarrow \Pi_{\text{Sign}}.\mathsf{KeyGen}(1^\lambda)$ as the judge would in the real protocol, and outputs pk_{sign} to the adversary $\mathcal{A}$. $\mathcal{S}$ initializes an instance of the ideal functionality in prospective mode. Finally, $\mathcal{S}$ runs $\pi_{\text{PRO}}^{t,p,i}.\mathsf{ActivateWarrant}$ with an empty active warrants set.

2. $\mathcal{S}$ computes $(pk_j, sk_j) \leftarrow \Pi_{\text{Enc}}.\mathsf{KeyGen}(1^\lambda)$ and samples a unique $id_j \leftarrow \mathbb{Z}_p$ for all honest P_j and sends (pk_j, id_j) to $\mathcal{A}$. Additionally, $\mathcal{S}$ waits to receive pk_i, id_i from P_i.

3. When $\mathcal{S}$ receives $(c_1, c_2, \pi, \mathsf{aux})$ from $\mathcal{A}$ intended for uncorrupted P_j, $\mathcal{S}$

begins by verifying π and checking that aux is correct. If these checks pass, set $b \leftarrow 1$, and $b \leftarrow 0$ otherwise. $\mathcal{S}$ computes $m \leftarrow \Pi_{\text{Enc}}.\text{Dec}(sk_j, c_1)$. $\mathcal{S}$ then sends $(\text{SendMessage}, P_j, m, b)$ to the ideal functionality.

4. Upon receiving $(\text{Sent}, \text{aux}, c, m)$ from the ideal functionality, $\mathcal{S}$ computes $(c_1, c_2, \pi, \text{aux})$ using $\pi_{\text{PRO}}^{t,p,i}.\text{SendMessage}$ and forwards it to P_i.

5. Upon receiving $(\text{NotifyWarrant}, t(w))$ from $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\text{pro}}$, $\mathcal{S}$ aggregates $t(w)$ from all $(\text{NotifyWarrant}, \cdot)$ messages seen so far into $\mathbb{L}$. $\mathcal{S}$ then randomly chooses a set of tags $\mathcal{T}$ such that $|\mathcal{T}| = |\mathbb{L}|$ and computes $(params, msk) \leftarrow \Pi_{\text{LTE}}.\text{KeyGen}(1^\lambda, \mathcal{T})$. $\mathcal{S}$ then simulates the proof π and sends $(\text{Post}, (params, \pi, \mathbb{L}))$ to $\mathcal{L}^{\text{Verify}}$.

We proceed with a hybrid argument. Let $\mathcal{H}_0$ denote the distribution of the view of $\mathcal{A}$ in the real world interaction.

$\mathcal{H}_1$: Let $\mathcal{H}_1$ be the same as $\mathcal{H}_0$, but instead of having the common reference string generated by the $\mathcal{F}_{CRS}^{\Pi_{\text{NIZK}}.\text{ZKSetup}}$, the common reference string is generated using $(\text{crs}, \tau) \leftarrow \Pi_{\text{NIZK}}.\text{ZKSetup}(1^\lambda)$. Note that the common reference string is selected from exactly the same distribution, so the difference in the distribution of the view of $\mathcal{A}$ between $\mathcal{H}_0$ and $\mathcal{H}_1$ is 0.

$\mathcal{H}_2$: Let $\mathcal{H}_2$ be the same as $\mathcal{H}_1$, but when an honest P_{LE} sends $(\text{Post}, (params, \pi, \mathbb{L}))$ to $\mathcal{L}^{\text{Verify}}$, the proof π is instead simulated with τ. Because of the perfect zero-knowledge property of Π_{NIZK}, the adversary's view in $\mathcal{H}_2$ and $\mathcal{H}_1$ is statistically close.

$\mathcal{H}_3$: Let $\mathcal{H}_3$ be the same as $\mathcal{H}_2$, but when an honest P_{LE} sends $(\text{Post}, (params, \pi, \mathbb{L}))$ to $\mathcal{L}^{\text{Verify}}$, let *params* be generated with an random set of tags

with the correct size. Because of the indistinguishability of tag sets property of Π_{LTE}, the difference in the distribution in the view of the $\mathcal{A}$ is negligible.

Because the view of $\mathcal{A}$ in $\mathcal{H}_3$ is distributed the same as in the ideal world with simulator $\mathcal{S}$, the proof is done. Notice that this proof extends directly to multiple corrupt users, as the views of the users are independent, except when they send messages to each other. However, such messages do not require simulation. Thus it suffices to simulate each one independently.

P_{LE} is corrupted. We now extend the previous proof to include corrupted P_{LE}. We extend $\mathcal{S}$ to simulate the view of P_{LE}. Note that step 5 described in $\mathcal{S}$ above is no longer applicable for corrupted users, as the notification mechanism from the actual protocol will look correct.

1. $\mathcal{S}$ generates the common reference string for the NIZK scheme directly, and stores the trapdoor τ. Then, $\mathcal{S}$ sets ValidParameters to true. $\mathcal{S}$ then runs $\Pi_{\text{Sign}}.\mathsf{KeyGen}(1^\lambda) \rightarrow (pk_{\text{sign}}, sk_{\text{sign}})$ as the judge would in the read protocol, and outputs pk_{sign} to the adversary $\mathcal{A}$. $\mathcal{S}$ initializes an instance of the ideal functionality in prospective mode. $\mathcal{S}$ computes $(pk_j, sk_j) \leftarrow \Pi_{\text{Enc}}.\mathsf{KeyGen}(1^\lambda)$ and samples a unique $id_j \leftarrow \mathbb{Z}_p$ for all honest P_j and sends (pk_j, id_j) to $\mathcal{A}$. When $\mathcal{S}$ detects $(params, \pi, \mathbb{L})$ posted on $\mathcal{L}^{\mathsf{Verify}}$, it verifies the proof π, and sets ValidParameters to false if it does not verify or $|\mathbb{L}| \neq 0$. $\mathcal{S}$ then initializes an empty warrant table W.

2. We split the case of receiving $(\mathsf{Sent}, \mathsf{aux}, c)$ from $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{pro}}$ intended for P_{LE} into three cases. All begin by $\mathcal{S}$ extracting the recipient P_j from aux:

   (a) If ValidParameters is false, $\mathcal{S}$ silently drops the message.

(b) If ValidParameters is true and P_j is not corrupted, $\mathcal{S}$ and chooses a message m_0 such that $\ell(m_0) = \mathsf{aux}$, and computes the ciphertext $(c_1, c_2, \pi, \mathsf{aux})$ by encrypting m_0 using $\pi^{t,p,i}_{\mathrm{PRO}}.\mathsf{SendMessage}(pk_j, id_j, m_0)$.

(c) If ValidParameters is true and P_j is corrupted, $\mathcal{S}$ will also receive $(\mathsf{Sent}, \mathsf{aux}, c, m)$ from the ideal functionality, intended for P_j. $\mathcal{S}$ then encrypts m using $\pi^{t,p,i}_{\mathrm{PRO}}.\mathsf{SendMessage}(pk_j, id_j, m)$

Finally, $\mathcal{S}$ sends the resulting ciphertext to $\mathcal{A}$.

3. Upon receiving $(\mathsf{Sent}, \mathsf{aux}, c, 0)$ from $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{\mathrm{ARLEAS}}$ intended for P_{LE}, $\mathcal{S}$ samples a random message and creates a ciphertext with $\pi^{t,p,i}_{\mathrm{PRO}}.\mathsf{SendMessage}$. Then, it generates a false proof instead of the real proof.

4. Upon receiving $(\mathsf{Sent}, \mathsf{aux}, c, m)$ from $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{\mathrm{ARLEAS}}$ due to an active warrant, if ValidParameters is true, $\mathcal{S}$ calls the send message algorithm of the real protocol on $\pi^{t,p,i}_{\mathrm{PRO}}.\mathsf{SendMessage}(pk_j, id_j, m)$ and sends the output to $\mathcal{A}$.

5. Upon receiving $(\mathsf{RequestWarrant}, id_j)$ from the adversary, intended for P_J, $\mathcal{S}$ generates a warrant w for the ideal functionality that only targets P_j and sends $(\mathsf{RequestWarrant}, w)$ to $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{\mathrm{ARLEAS}}$. If $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{\mathrm{ARLEAS}}$ answers with $(\mathsf{Approve})$, $\mathcal{S}$ uses the sk_{sign} for P_J to form and sign (id_j, σ) as in the real protocol and adds (w, False) to W. If $\mathcal{F}^{\ell,t,p,i,\mathsf{pro}}_{\mathrm{ARLEAS}}$ responds with $(\mathsf{Disapprove})$, $\mathcal{S}$ send $\perp$ to the adversary.

6. $\mathcal{S}$ monitors $\mathcal{L}^{\mathsf{Verify}}$. Upon seeing a new post on the ledger, $\mathcal{S}$ performs the following steps

(a) Retrieve the post by sending $(\mathsf{GetCounter})$ to $\mathcal{L}^{\mathsf{Verify}}$ and receive the

current counter t. Then query $\mathcal{L}^{\mathsf{Verify}}$ on (GetVal, t) to receive the latest posting $(t, (params, \pi, \mathbb{L}), \pi_{\mathsf{publish}})$

(b) Verify $\Pi_{\mathrm{NIZK}}.\mathsf{ZKVerify}(params, \mathbb{L}, \pi) = 1$. If the proof does not verify, the $\mathcal{S}$ sets ValidParameters to false and halts.

(c) Set ValidParameters to true and runs $(\mathcal{W}, \mathcal{T}, msk, r) \leftarrow \mathsf{Extract}(crs, \tau, x, \pi)$. If extraction fails, the simulator halts with an error.

(d) If there is an entry (w, False) in W for $w \in \mathcal{W}$ $\mathcal{S}$ sends $(\mathsf{ActivateWarrant}, w)$ to the ideal functionality and sets the entry to be (w, True).

We proceed with a hybrid argument. Let $\mathcal{H}_0$ denote the distribution of the view of $\mathcal{A}$ in the real world interaction.

$\mathcal{H}_\mathbf{1}$: Let $\mathcal{H}_1$ be the same as $\mathcal{H}_0$, but instead of having the common reference string generated by the $\mathcal{F}_{CRS}^{\Pi_{\mathrm{NIZK}}.\mathsf{ZKSetup}}$, the common reference string is generated using $(\mathsf{crs}, \tau) \leftarrow \Pi_{\mathrm{NIZK}}.\mathsf{ZKSetup}(1^\lambda)$. Note that the common reference string is selected from exactly the same distribution, so the the distribution in the view of $\mathcal{A}$ between $\mathcal{H}_0$ and $\mathcal{H}_1$ is statistically close.

$\mathcal{H}_\mathbf{2}$: Let $\mathcal{H}_2$ be the same as $\mathcal{H}_1$, except the proof of ciphertexts consistency in a ciphertext $(c_1, c_2, \pi, \mathsf{aux})$ bound for an honest user for which there is no active warrant is simulated. Due to the zero-knowledge property of Π_{NIZK}, the difference in the view of the adversary in $\mathcal{H}_2$ and $\mathcal{H}_1$ is negligible.

$\mathcal{H}_\mathbf{3}$: Let $\mathcal{H}_3$ be the same as $\mathcal{H}_2$, when $\mathcal{S}$ receives a ciphertext $(c_1, c_2, \pi, \mathsf{aux})$ bound for an honest user for which there is no active warrant, $\mathcal{S}$ samples a message m_0 that would result in the same metadata and sets

$$c_1 \leftarrow (\Pi_{\mathrm{Enc}}.\mathsf{Enc}(pk_j, m_0; r_1)$$

. By the CCA security of Π_{Enc} the advantage of $\mathcal{A}$ in distinguishing between $\mathcal{H}_3$ and $\mathcal{H}_2$ is negligible.

$\mathcal{H}_4$: Let $\mathcal{H}_4$ be the same as $\mathcal{H}_3$, except the second ciphertext element c_2 in a ciphertext $(c_1, c_2, \pi, \mathsf{aux})$ bound for an honest user for which there is no active warrant is computed as $\Pi_{\text{LTE}}.\mathsf{Enc}(params, id_j, m_0; r_2)$. By the lossy property of Π_{LTE}, $\mathcal{H}_4$ and $\mathcal{H}_3$ are statistically indistinguishable.

$\mathcal{H}_5$: Let $\mathcal{H}_5$ be the same as $\mathcal{H}_4$, except the proof of ciphertexts consistency in a ciphertext $(c_1, c_2, \pi, \mathsf{aux})$ bound for an honest user for which there is no active warrant is computed honestly with respect to the plaintext message m_0. Again, by the zero-knowledge property of Π_{NIZK}, the difference in the view of the adversary in $\mathcal{H}_5$ and $\mathcal{H}_4$ is negligible.

$\mathcal{H}_6$: Let $\mathcal{H}_6$ be the same as $\mathcal{H}_5$, except when $\mathcal{A}$ detects $(params, \pi, \mathbb{L})$ being posted on the ledger, we attempt to run the extractor $\Pi_{\text{NIZK}}.\mathsf{Extract}$ and abort the experiment if it fails. However, because the extractor only fails with negligible probability, the difference in the view of the adversary between $\mathcal{H}_6$ and $\mathcal{H}_5$ is negligible.

$\mathcal{H}_6$ has the same distribution as $\mathcal{S}$, concluding the proof. In the real world, P_{LE} would be denied warrants at the same rate as in the ideal world, as an honest P_J handles warrant requests in the same way. One note is that law enforcement can "deactivate" warrants in way not possible in the ideal functionality. However, when warrants are deactivated, honestly encrypting the message still hides the plaintext from the adversary.

P_J and P_{LE} are corrupted. We now focus on the case when P_J and P_{LE} are both corrupted. Note that step 4 of the above simulator description is no longer

relevant, as the warrant request is handled internally by $\mathcal{A}$. Our simulator requires on minor changes to steps 1 and 5, which we show below.

1. Do the setup as before, but waiting to receive pk_{sign} from $\mathcal{A}$

⋮

5. $\mathcal{S}$ monitors $\mathcal{L}^{\mathsf{Verify}}$. Upon seeing a new post on the ledger, $\mathcal{S}$ performs the following steps

   (a) Retrieve the post by sending $(\mathsf{GetCounter})$ to $\mathcal{L}^{\mathsf{Verify}}$ and receive the current counter t. Then query $\mathcal{L}^{\mathsf{Verify}}$ on (GetVal, t) to receive the latest posting $(t, (params, \pi, \mathbb{L}), \pi_{\mathsf{publish}})$

   (b) Verify $\Pi_{\text{NIZK}}.\mathsf{ZKVerify}(params, \mathbb{L}, \pi) = 1$. If the proof does not verify, the $\mathcal{S}$ sets ValidParameters to false and halts.

   (c) Set ValidParameters to true and runs

   $$(\mathcal{W}, \mathcal{T}, msk, r) \leftarrow \mathsf{Extract}(crs, \tau, x, \pi).$$

   If extraction fails, the simulator halts with an error.

   (d) for each warrant $w \in \mathcal{W}$ for which there does not exist an entry (w, True) in W, the $\mathcal{S}$ executes the following steps

      i. $\mathcal{S}$ generates a warrant w for the ideal functionality that only targets P_j and sends $(\mathsf{RequestWarrant}, w)$ to $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{pro}}$.

      ii. When $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{pro}}$ sends $(\mathsf{ApproveWarrant}, w)$ to the $\mathcal{S}$ intended to P_J, $\mathcal{S}$ responds $(\mathsf{Approve})$ on behalf of P_J

      iii. $\mathcal{S}$ sends $(\mathsf{ActivateWarrant}, w)$ to the ideal functionality

iv. $\mathcal{S}$ adds (w, True) to W

In fact, the hybrid argument above holds directly in this case as well. First notice that there are no additional messages that require simulation. By giving the adversary control of P_J, we give it the ability to create valid warrants independently. However, there is no change in behavior expected until those warrants are activated. As such, those warrants can be requested from the ideal functionality right as they are being activated.

Prospective Solution with Arbitrary Predicates. The prospective solution we have presented is limited in the flexibility of warrants; unlike the ideal functionality, warrants were limited to specifying a single individual. It is clear that it would be better to support warrants whose applicability to some aux could be checked with *arbitrary* functions i. While we do not specify a full solution, we quickly give the intuition for how such a system could work.

First, we note that directly extending the existing solution's intuitions is insufficient, as creating a one-to-one correspondence between tags and aux would clearly result in exponentially sized public parameters. Instead, we require a mechanism that evaluates the in-scope predicate $i(w, \mathsf{aux})$ and outputs the message only if the result is 1. Clearly this can be accomplished using extractable witness encryption, and indeed we will use extractable witness encryption to accomplish a similar goal in Section 5.5. However, we are able to leverage the *prospective* nature of this case to realize prospective ARLEAS from garbled circuits.

As before, the sender send a ciphertext c_1, c_2, π. c_1 is a normal public key

ciphertext for the recipient, c_2 is a ciphertext that can be decrypted by law enforcement under the proper conditions, and π is a consistency proof between them. To allow for the arbitrary predicate function $i(\cdot, \cdot)$, c_2 will be computed as a garbled circuit of i with the message and metadata hardcoded. This garbled circuit will be evaluated by law enforcement, and output the message if and only if the existing warrants satisfy $i(\cdot, \cdot)$. When law enforcement calls ActivateWarrant (before this message is encrypted), law enforcement also posts the first message of the oblivious transfer protocol for each bit in the warrant. When the sender is encrypting, it finishes the oblivious transfer protocol to send the proper labels law enforcement, and includes these oblivious transfer message in c_2. To attempt to recover the message, law enforcement recovers the input labels corresponding to its warrant from the oblivious transfer messages and evaluates the circuit. Note that the sender includes a proof that the garbled circuit is correctly formatted and that the oblivious transfer was completed correctly.

We now describe this protocol more formally. For simplicity, let all warrants w have some fixed length $|w|$ (which can be accomplished by choosing some maximum length and padding). Let $C_k^{m,\mathsf{aux}}$ be a circuit representation of the functionality $I_k(m, \mathsf{aux}, w_1, w_2, \ldots, w_k)$ with m and aux hardcoded, such that

$$I_k(m, \mathsf{aux}, w_1, w_2, \ldots, w_k) = m \wedge (i(\mathsf{aux}, w_1) \vee \ldots \vee i(\mathsf{aux}, w_k)).$$

This function evaluates the in-scope functionality on the metadata over k

different warrants. If any of the evaluate to true, the message is output. Otherwise, I_k outputs 0. When publishing a set of warrants, law enforcement runs the receiver side of the OT scheme with the bits of the warrants as input and proves that these messages are formed correctly. When sending a message, the sender computes $c_2 \leftarrow \tilde{C}^{m,\mathsf{aux}}_{|\mathbb{L}|} \leftarrow \Pi_{\mathrm{GC}}.\mathsf{Garble}(C^{m,\mathsf{aux}}_{|\mathbb{L}|}, (\{\mathsf{label}^{i,b}\}_{i\in\mathsf{input},b\in\{0,1\}}))$ and then appends the sender messages for the OT scheme. The proof included in the ciphertext not only proves that the garbled circuit is correctly formed, but also that the OT sender messages are well formed. Note that the number of warrants is an explicit parameter of the function and its circuit representation.

5.5 Retrospective Solution

In the previous section we proposed a protocol to realize ARLEAS under the restriction that access would be prospective only. That protocol requires that law enforcement must activate a warrant and post the resulting parameters on the ledger before any targeted communication occurs. In this section we address the retrospective case. The key difference in this protocol is that law enforcement may activate a warrant at any stage of the protocol, even after a target communication has occurred.

Our construction makes use of an extractable witness encryption scheme Π_{EWE} as defined in Definition 9. This scheme is parameterized by a language L_{WE} that is defined with respect to the transparency function t, the policy function p, the targeting function i, the warrant signing key pk_{sign}, and the ledger verification function Verify as follows:

$$L_{\text{WE}} = \left\{ \mathsf{aux} \;\middle|\; \exists w, (t, x, \pi_{\mathsf{publish}}) \text{ s.t. } \begin{array}{l} w = (\hat{w}, \sigma) \\ x = t(w); \\ \mathsf{Verify}((t \| x), \pi_{\mathsf{publish}}) = 1; \\ p(\hat{w}) = 1; \\ \Pi_{\text{Sign}}.\mathsf{Verify}(pk_{\text{sign}}, \hat{w}, \sigma) = 1; \\ i(\hat{w}, \mathsf{aux}) = 1 \end{array} \right\}$$

Our construction also makes use of a simulation-extractable NIZK scheme Π_{NIZK} satisfying Definition 11. We will prove statements in the following languages:

$$L^1_{\text{NIZK}} = \left\{ (c_1, c_2, pk, \mathsf{aux}) \;\middle|\; \exists (r, r_1, r_2) \text{ s.t. } \begin{array}{l} c_1 = \Pi_{\text{Enc}}.\mathsf{Enc}(pk, r; r_1) \wedge \\ c_2 = \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r; r_2) \end{array} \right\}$$

$$L^2_{\text{NIZK}} = \left\{ (x, pk_{\text{sign}}) \;\middle|\; \exists (\hat{w}, \pi) \text{ s.t. } \begin{array}{l} \Pi_{\text{Sign}}.\mathsf{Verify}(pk_{\text{sign}}, \hat{w}, \sigma) = 1 \wedge \\ x \leftarrow t(w) \end{array} \right\}$$

We will describe our protocol in a hybrid model that makes use of several functionalities. These include $\mathcal{L}^{\mathsf{Verify}}$, $\mathcal{F}^{\mathcal{D}}_{CRS}$, $\mathcal{G}_{\mathsf{pRO}}$ and $\mathcal{F}_{AUTH}$.

5.5.1 UC-Realizing Retrospective Abuse-Resistant Law Enforcement Access Systems

We now provide a description of the retrospective ARLEAS protocol $\pi^{t,p,i}_{\text{RET}}$.

$\pi^{t,p,i}_{\text{RET}}$.Setup:

- All users send (CRS) to $\mathcal{F}^{\Pi_{\text{NIZK}}.\mathsf{ZKSetup}}_{CRS}$ to retrieve the common reference string for the NIZK scheme.
- P_j computes $(pk_j, sk_j) \leftarrow \Pi_{\text{Enc}}.\mathsf{KeyGen}(1^\lambda)$ and selects a unique tag id_j and sends (pk_j, id_j) to P_{LE} and to each P_i via $\mathcal{F}_{AUTH}$.

- P_J computes $(pk_{\text{sign}}, sk_{\text{sign}}) \leftarrow \Pi_{\text{Sign}}.\mathsf{KeyGen}(1^\lambda)$ and sends pk_{sign} to all other users via $\mathcal{F}_{AUTH}$.

$\pi^{t,p,i}_{\mathbf{RET}}.\mathsf{SendMessage}$**:**

- P_i performs the following steps, on input a metadata string aux and a plaintext M:
  - Sample $r \leftarrow \{0,1\}^\lambda$
  - Query the random oracle to obtain the hashes:

    $$(\mathsf{HashConfirm}, r_1) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“ENC”} \| r \| m)),$$

    $$(\mathsf{HashConfirm}, r_2) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“WE”} \| r \| m)), \text{ and}$$

    $$(\mathsf{HashConfirm}, r_3) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“RP”} \| r))$$

  - $c_1 \leftarrow \Pi_{\text{Enc}}.\mathsf{Enc}(pk, r; r_1)$, $c_2 \leftarrow \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r; r_2)$, and $c_3 \leftarrow m \oplus r_3$
  - Use $\Pi_{\text{NIZK}}.\mathsf{ZKProve}$ for L^1_{NIZK} to compute

    $$\pi \leftarrow NIZK\{(r, r_1, r_2) : c_1 = \Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, r; r_1) \wedge c_2 = \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r; r_2)\}$$

  - Send $(\mathsf{send}, c) = (c_1, c_2, c_3, \pi, \mathsf{aux})$ to P_j and P_{LE}.
- Upon receiving (send, c), P_j performs the following steps:
  - Call $\pi^{t,p,i}_{\text{PRO}}.\mathsf{VerifyMessage}$ on c, aborting if the output is 0;
  - Compute $r' \leftarrow \Pi_{\text{Enc}}.\mathsf{Dec}(sk_j, c_1)$;
  - $(\mathsf{HashConfirm}, r_3) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“RP”} \| r'))$;
  - Compute $m' \leftarrow c_3 \oplus r_3$;

- $(\mathsf{HashConfirm}, r_1) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“ENC”} \| r' \| m'))$;
- $(\mathsf{HashConfirm}, r_2) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“WE”} \| r' \| m'))$;
- Then to verify that the message has not been mauled, P_j recomputes $c_1' \leftarrow \Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, r'; r_1)$ and $c_2' \leftarrow \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r'; r_2)$. If $c_1 \neq c_1'$ or $c_2 \neq c_2'$, return $\perp$. Otherwise, return m'.

$\pi^{t,p,i}_{\mathbf{PRO}}.\mathsf{VerifyMessage}$:

- Any party parses $(c_1, c_2, c_3, \pi, \mathsf{aux}) \leftarrow c$ and verifies that π and aux are correct, aborting if the output is 0. Otherwise, output 1.

$\pi^{t,p,i}_{\mathbf{RET}}.\mathsf{RequestWarrant}$**:**

- P_{LE} formulates a warrant request $\hat{w}$ and sends $(\mathsf{RequestWarrant}, \hat{w})$ to P_J via $\mathcal{F}_{AUTH}$
- Upon receiving $(\mathsf{RequestWarrant}, \hat{w})$ from P_{LE}, P_J either outputs $\perp$ or computes $\pi \leftarrow \Pi_{\text{Sign}}.\mathsf{Sign}(wsk, \hat{w})$ and sends $w = (\hat{w}, \pi)$ to P_{LE} via $\mathcal{F}_{AUTH}$.

$\pi^{t,p,i}_{\mathbf{RET}}.\mathsf{ActivateWarrant}$**:**

- P_{LE} computes $x \leftarrow t(w)$; uses $\Pi_{\text{NIZK}}.\mathsf{ZKProve}$ for L^2_{NIZK} to compute $\pi \leftarrow NIZK\{(w) : w = (\hat{w}, \sigma), \Pi_{\text{Sign}}.\mathsf{Verify}(pk_{\text{sign}}, \hat{w}, \sigma) = 1 \wedge x \leftarrow t(w)\}$; and sends $(\mathsf{Post}, (x, \pi))$ to $\mathcal{L}^{\mathsf{Verify}}$. It receives and returns $(t, x, \pi_{\mathsf{publish}})$.

$\pi^{t,p,i}_{\mathbf{RET}}.\mathsf{VerifyWarrantStatus}$**:**

- P_{LE} calls $\Pi_{\text{EWE}}.\mathsf{Dec}(c_2, \mathsf{aux}, (\hat{w}, \sigma), (t, x, \pi_{\mathsf{publish}}))$. If the output is $\perp$, return 0. Otherwise, return 1.

$\pi^{t,p,i}_{\mathbf{RET}}.\mathsf{AccessMessage}$**:**

- P_{LE} computes $r' \leftarrow \Pi_{\text{EWE}}.\mathsf{Dec}(c_2, \mathsf{aux}, (\hat{w}, \sigma), (t, x, \pi_{\mathsf{publish}}))$.
- $(\mathsf{HashConfirm}, r_3) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“RP”} \| r'))$;
- Recovers $m' \leftarrow c_3 \oplus r_3$.
- $(\mathsf{HashConfirm}, r_1) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“ENC”} \| r' \| m'))$;
- $(\mathsf{HashConfirm}, r_2) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“WE”} \| r' \| m'))$;
- Recomputes $c'_1 \leftarrow \Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, r'; r_1)$ and $c'_2 \leftarrow \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r'; r_2)$. If $c'_1 = c_1$ and $c'_2 = c_2$, P_{LE} returns m' and $\perp$ otherwise.

Theorem 6 *Assuming a CCA-secure public key encryption scheme Π_{Enc}, an extractable witness encryption scheme for L_{WE}, a SUF-CMA secure signature scheme Π_{Sign}, and a simulation-extractable NIZK scheme Π_{NIZK}, $\pi^{t,p,i}_{RET}$ UC-realizes $\mathcal{F}^{\ell,t,p,i,ret}_{ARLEAS}$ in the $\mathcal{L}^{\mathsf{Verify}}, \mathcal{F}^{\Pi_{NIZK}.\mathsf{ZKSetup}}_{CRS}, \mathcal{G}_{\mathsf{pRO}}$−hybrid model.*

5.5.1.1 Proof of Security

As in the prospective case in Section 5.4, we show that $\pi^{t,p,i}_{\text{RET}}$ UC-realizes $\mathcal{F}^{\ell,t,p,i,\mathsf{ret}}_{\text{ARLEAS}}$ in a series of steps. First, we prove this for a single corrupt user, then extend that argument to multiple users. We then expand our analysis to consider corrupted law enforcement and corrupted judges.

P_i **is corrupted.** We begin with the simple case of a single corrupted user P_i controlled by an adversary $\mathcal{A}$. We construct a simulator $\mathcal{S}$ to mediate $\mathcal{A}$'s interaction with the ideal functionality as follows.

1. $\mathcal{S}$ begins by generating $(pk_{\text{sign}}, sk_{\text{sign}})$ as P_J would do. $\mathcal{S}$ then performs key generation for each honest party P_j. $\mathcal{S}$ then outputs all the public information to $\mathcal{A}$. Finally, $\mathcal{S}$ receives pk_i from $\mathcal{A}$.

2. When receiving a (CRS) request from $\mathcal{A}$, $\mathcal{S}$ generates $(crs, \tau) \leftarrow \Pi_{\text{NIZK}}.\mathsf{ZKSetup}(1^\lambda)$ and returns crs to $\mathcal{A}$ and keeps τ.

3. Whenever $\mathcal{S}$ receives (send, c) from P_i intended for P_j, $\mathcal{S}$ parses $c = (c_1, c_2, c_3, \pi, \mathsf{aux})$ and verifies π. If it does not verify, set $b \leftarrow 1$, and set $b \leftarrow 0$ otherwise. Next, $(r, r_1, r_2) \leftarrow \Pi_{\text{NIZK}}.\mathsf{Extract}(crs, \tau, x, \pi)$, queries the random oracle $(\mathsf{HashConfirm}, r_3) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“RP”} \| r))$. It then computes $m \leftarrow c_3 \oplus r_3$ and verifies the structure of c_1, c_2 by recomputing the ciphertexts c_1, c_2 as in the real protocol. If some passes do not check, set $b \leftarrow 0$. Finally, it sends $(\mathsf{SendMessage}, \mathsf{sid}, P_j, m, b)$ to the ideal functionality.

4. When $\mathcal{S}$ receives $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ from the ideal functionality destined for P_i, it identifies the public key for P_i and computes the ciphertext $c = (c_1, c_2, c_3, \pi, \mathsf{aux})$ as in the real protocol. It sends the resulting ciphertext to P_i.

5. When $\mathcal{S}$ receives $(\mathsf{NotifyWarrant}, t(w))$ from the ideal functionality, it simulates the proof π, and sends $(\mathsf{Post}, (t(w), \pi))$ to $\mathcal{L}^{\mathsf{Verify}}$.

We prove that $\mathcal{A}$'s interaction with the real protocol and the ideal functionality, mediated by $\mathcal{S}$ are computationally indistinguishable by using the following hybrids.

Let $\mathcal{H}_0$ denote the distribution of the view of $\mathcal{A}$ in the real world interaction.

$\mathcal{H}_1$: Let $\mathcal{H}_1$ be the same as $\mathcal{H}_0$, but instead of having the common reference string generated by the $\mathcal{F}_{CRS}^{\Pi_{\text{NIZK}}.\mathsf{ZKSetup}}$, the common reference string is generated using $(\mathsf{crs}, \tau) \leftarrow \Pi_{\text{NIZK}}.\mathsf{ZKSetup}(1^\lambda)$. Note that the common reference string is selected from exactly the same distribution, therefore, the distribution of the view of $\mathcal{A}$ between $\mathcal{H}_0$ and $\mathcal{H}_1$ is the same.

$\mathcal{H}_2$: In this hybrid we will use the extractor $\Pi_{\text{NIZK}}.\mathsf{Extract}$ to get the randomness and create the ciphertext as in step 3. $\mathcal{H}_1$ and $\mathcal{H}_2$ are computationally indistinguishable as $\mathsf{Extract}$ will only fail with negligible probability.

$\mathcal{H}_3$: At this point we will simulate the proof π when publishing on the ledger. $\mathcal{H}_2$ and $\mathcal{H}_3$ are indistinguishable because of the zero knowledge property of Π_{NIZK}.

The view of the adversary in $\mathcal{H}_3$ is the same as its view when talking with $\mathcal{S}$ in the ideal world, which concludes the hybrid argument.

This argument extends to multiple parties as all ciphertexts can be properly simulated by either extracting from the NIZK or actually receiving the message in case the receiver is corrupted.

Subset of users and P_{LE} are corrupted. We now extend $\mathcal{S}$ from above to handle a corrupt P_{LE}. Note that step 5 of the above description is not longer relevant, as notifications will come directly from the ledger for the corrupt parties.

1. $\mathcal{S}$ runs the same setup as above. Additionally, $\mathcal{S}$ initializes an empty message equivocation table T.

2. When receiving a (CRS) request from $\mathcal{A}$, $\mathcal{S}$ generates $(crs, \tau) \leftarrow \Pi_{\text{NIZK}}.\mathsf{ZKSetup}(1^\lambda)$ and returns crs to $\mathcal{A}$ and keeps τ.

3. We split the case of receiving $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux})$ from $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{ret}}$ intended for P_{LE} into two cases:

   (a) If P_j is not corrupted, $\mathcal{S}$ samples $r, r_1, r_2, r_3 \leftarrow \{0,1\}^{\text{poly}(\lambda)}$ (where $\text{poly}(\cdot)$ is a polynomial function that upper-bounds the longest random string necessary) and computes the ciphertext components as follows

   - $c_1 \leftarrow \Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, r; r_1)$
   - $c_2 \leftarrow \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r; r_2)$
   - $c_3 \leftarrow r_3$
   - Use $\Pi_{\text{NIZK}}.\mathsf{ZKProve}$ to compute

   $$\pi \leftarrow NIZK\left\{(r, r_1, r_2) \;\middle|\; \begin{array}{l} c_1 = \Pi_{\text{Enc}}.\mathsf{Enc}(pk, r; r_1) \wedge \\ c_2 = \Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r; r_2) \end{array}\right\}$$

   $\mathcal{S}$ adds an entry $(r, r_1, r_2, r_3, c_1, c_2, c_3, \pi, \mathsf{aux})$ to T.

   (b) If P_i or P_j is corrupted, $\mathcal{S}$ will also receive $(\mathsf{Sent}, \mathsf{sid}, \mathsf{aux}, c, m)$ from the ideal functionality, intended for P_j. $\mathcal{S}$ then encrypts m using $\pi_{\text{RET}}^{t,p,i}.\mathsf{SendMessage}$, programming the random oracle honestly as needed.

   $\mathcal{S}$ then sends the resulting ciphertext to both the ideal functionality and $\mathcal{A}$.

4. Upon receiving $(\mathsf{Sent}, \mathsf{aux}, c, 0)$ from $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{ret}}$ intended for P_{LE}, $\mathcal{S}$ samples a random message and creates a ciphertext with $\pi_{\text{RET}}^{t,p,i}.\mathsf{SendMessage}$. Then, it generates a false proof instead of the real proof.

5. When $\mathcal{S}$ receives $(\mathsf{RequestWarrant}, \hat{w})$ from $\mathcal{A}$, intended for P_J, $\mathcal{S}$ sends $(\mathsf{RequestWarrant}, \hat{w})$ to the trusted party. If the trusted party responds with $\perp$, $\mathcal{S}$ sends $(\mathsf{Disapprove})$ to $\mathcal{A}$. Otherwise, $\mathcal{S}$ generates a warrant $\hat{w}$ and signs it with $\sigma = \Pi_{\text{Sign}}.\mathsf{Sign}(sk_{\text{sign}}, \hat{w})$ and sends $w = (\hat{w}, \sigma)$ to $\mathcal{A}$.

6. When $\mathcal{S}$ notices new posts on $\mathcal{L}^{\mathsf{Verify}}$ it retrieves that information by sending $t \leftarrow (\mathsf{GetCounter})$ and $(t(w), \pi)) \leftarrow (\mathsf{GetVal}, t)$ to $\mathcal{L}^{\mathsf{Verify}}$. $\mathcal{S}$ uses τ to extract w from π. Without loss of generality, let w be for user P_i. $\mathcal{S}$ sends $(\mathsf{ActivateWarrant}, w, P_j)$ to the ideal functionality. Next, $\mathcal{S}$ checks to see if there is an entry $(r, r_1, r_2, r_3, c_1, c_2, c_3, \pi, \mathsf{aux})$ for which $i(\hat{w}, \mathsf{aux}) = 1$ in T and sends $(\mathsf{AccessData}, (c_1, c_2, c_3, \pi), w)$ to the ideal functionality for all such records. If the ideal functionality responds with $\perp$, abort. Otherwise, the ideal functionality will return a message m. $\mathcal{S}$ then programs the random oracle by sending $(\mathsf{ProgramRO}, r, (r_3 \oplus m))$, $(\mathsf{ProgramRO}, (\text{“WE”} r \| m), r_2)$ and $(\mathsf{ProgramRO}, (\text{“ENC”} \| r \| m), r_1)$, and responds to the initial query.

To show that the simulation above is computationally indistinguishable from the real experiment in the view of $\mathcal{A}$, we proceed with a hybrid argument. Let $\mathcal{H}_0$ denote the distribution of the view of $\mathcal{A}$ in the real world interaction.

$\mathcal{H}_1$: Let $\mathcal{H}_1$ be the same as $\mathcal{H}_0$, but instead of having the common reference

string generated by the $\mathcal{F}_{CRS}^{\Pi_{\text{NIZK}}.\mathsf{ZKSetup}}$, the common reference string is generated using $(\mathsf{crs}, \tau) \leftarrow \Pi_{\text{NIZK}}.\mathsf{ZKSetup}(1^\lambda)$. Note that the common reference string is selected from exactly the same distribution, therefore, the distribution of the view of $\mathcal{A}$ between $\mathcal{H}_0$ and $\mathcal{H}_1$ is the same.

$\mathcal{H}_2$: In this hybrid we change the NIZK proof for L^1_{NIZK} to be a simulation. Because of the zero knowledge property of Π_{NIZK}, $\mathcal{H}_2$ is statistically close to $\mathcal{H}_3$.

$\mathcal{H}_3$: Choose r_1, r_2, and r_3 uniformly at random in $\{0,1\}^\lambda$ to replace the randomness used to compute c_1, c_2, and c_3 respectively. Also, send $(\mathsf{ProgramRO}, (\text{“ENC”}\|r\|m), r_1)$, $(\mathsf{ProgramRO}, (\text{“WE”}\|r\|m), r_2)$, and $(\mathsf{ProgramRO}, r, r_3)$ to $\mathcal{G}_{\mathsf{pRO}}$. Clearly the only way the view between $\mathcal{H}_1$ and $\mathcal{H}_2$ can differ is when $\mathcal{G}_{\mathsf{pRO}}$ was queried on these inputs before $\mathcal{S}$ programs them, in which case the protocol aborts, but this only happens with negligible probability which we prove in Lemma 5.

$\mathcal{H}_4$: Change the simulated NIZK proof back to a real proof. Again, by the zero-knowledge property $\mathcal{H}_3$ and $\mathcal{H}_4$ are indistinguishable.

$\mathcal{H}_5$: Now, do everything as described in step 5 to recover the message m. Next, we change the ciphertext c_3 in step 3 to be r_3 and change the programming of $\mathcal{G}_{\mathsf{pRO}}$ to be $(\mathsf{ProgramRO}, r, m \oplus r_3)$ in step 5. Note that by security of the one-time pad $\mathcal{H}_4$ and $\mathcal{H}_5$ are indistinguishable.

Now, the view of the adversary in $\mathcal{H}_5$ is exactly the same as its view when talking with $\mathcal{S}$ in the ideal world, which concludes the hybrid argument.

Lemma 5 *For any adversary $\mathcal{A}$ against $\mathcal{H}_4$ in the security proof of $\pi_{RET}^{t,p,i}$ where P_{LE} is compromised, the probability that $\mathcal{A}$ queries or programs $\mathcal{G}_{\mathsf{pRO}}$ on r, “ENC”$\|r\|m$, or*

"WE"$\|r\|m$ without sending $(\mathsf{Post}, (t(w), \pi))$ *to* $\mathcal{L}^{\mathsf{Verify}}$ *is negligible, for* $r \xleftarrow{\$} \{0,1\}^\lambda$ *and* m *uniformly at random from the message space, both used inside* $\mathcal{H}_4$*.*

Proof. Assume such adversary $\mathcal{A}$ exists, then if $\mathcal{A}$ has queried or programmed $\mathcal{G}_{\mathsf{pRO}}$ on one of the inputs it must have had r. Either $\mathcal{A}$ has selected r by accident which can only happen with probability $2^{-\lambda}$, or it has extracted it from the ciphertext, which we will now show with a series of hybrids can only happen with negligible probability.

$\mathcal{H}'_0$: This looks exactly the same as $\mathcal{H}_4$. The encryption is created in the following way:

- sample $r \leftarrow \{0,1\}^\lambda$
- $(\mathsf{HashConfirm}, r_1) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“ENC”}\|r\|m))$,
- $(\mathsf{HashConfirm}, r_2) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“WE”}\|r\|m))$, and
- $(\mathsf{HashConfirm}, r_3) \leftarrow \mathcal{G}_{\mathsf{pRO}}(\mathsf{HashQuery}, (\text{“RP”}\|r))$
- Create the components or the encryption in the following way:
  - $c_1 \leftarrow \Pi_{\mathrm{Enc}}.\mathsf{Enc}(pk_j, r; r_1)$
  - $c_2 \leftarrow \Pi_{\mathrm{EWE}}.\mathsf{Enc}(\mathsf{aux}, r; r_2)$
  - $c_3 \leftarrow m \oplus r_3$
  - Use $\Pi_{\mathrm{NIZK}}.\mathsf{ZKSimulate}$ to simulate the proof

$\mathcal{H}'_1$: Replace all three calls to $\mathcal{G}_{\mathsf{pRO}}$ with uniform random values r_1, r_2, r_3. By the properties of $\mathcal{G}_{\mathsf{pRO}}$ the view of $\mathcal{A}$ doesn't change between $\mathcal{H}'_0$ and $\mathcal{H}'_1$.

Note that we only care about the view of $\mathcal{A}$ up until it calls $\mathcal{G}_{\mathsf{pRO}}$ on one of the three values, so we don't have to program $\mathcal{G}_{\mathsf{pRO}}$ accordingly.

$\mathcal{H}'_{\mathbf{2}}$: Now we replace c_1 by $\Pi_{\text{Enc}}.\mathsf{Enc}(pk_j, r'; r_1)$, which is possible because the proof is simulated. By the security of Π_{Enc} the view of the adversary doesn't change.

$\mathcal{H}'_{\mathbf{3}}$: Next, we replace c_2 with $\Pi_{\text{EWE}}.\mathsf{Enc}(\mathsf{aux}, r'; r_2)$. If there would be a distinguisher for $\mathcal{H}'_2$ and $\mathcal{H}'_3$ we can build a distinguisher for Π_{EWE}, by the extractable security of Π_{EWE} we can now extract a witness, but this is in contradiction with the ideal functionality $\mathcal{G}_{\mathsf{ledger}}$, as we assumed it was not called. Therefore, $\mathcal{H}'_2$ and $\mathcal{H}'_3$ must be indistinguishable.

We see that $\mathcal{H}'_3$ does not contain any reference to r, and $\mathcal{A}$ would not have changed its strategy as the difference between its view in $\mathcal{H}'_0$ and $\mathcal{H}'_3$ is computationally indistinguishable.

Subset of users, P_J, and $P_{\mathbf{LE}}$ are corrupted. We further extend our analysis to cover both corrupt law enforcement and corrupt judges. Notice that step 4 of the previous simulator description is no longer relevant, as these requests are handled inside $\mathcal{A}$. The proof is exactly the same is in the previous case.

5.6 On the Need for Extractable Witness Encryption

The retrospective solution we present in Section 5.5 relies on extractable witness encryption. Intuitively, this strong assumption is required in our construction because a user must encrypt in a way that decryption is only possible

under certain circumstances. Because the description of these circumstances can be phrased as an NP relation, witness encryption represents a "natural" primitive for realizing it. However, thus far we have not shown that the use of extractable witness encryption is *strictly* necessary. Given the strength (and implausibility (Garg et al., 2014)) of the primitive, it is important to justify its use. We do this by showing that any protocol Π_{A} that UC-realizes $\mathcal{F}_{\mathrm{ARLEAS}}^{\ell,t,p,i,\mathsf{ret}}$ implies the existence of extractable witness encryption for a related language.

Before proceeding to formally define this related language, we give some intuition about its form. We wish to argue that a protocol Π_{A} *acts like* an extractable witness encryption scheme in the specific case where an adversary has corrupted the escrow authorities P_{LE} and P_J (along with an arbitrary number of unrelated users). Recall that in order to learn any information about a message sent in Π_{A}, the following conditions must be met: specifically, law enforcement must correctly run the protocol for $\Pi_{\mathrm{A}}.\mathsf{RequestWarrant}$ and $\Pi_{\mathrm{A}}.\mathsf{ActivateWarrant}$ such that if $\Pi_{\mathrm{A}}.\mathsf{VerifyWarrantStatus}$ were to be called, it would output 1.[5] For the protocol we presented in Section 5.5, this corresponds to obtaining a correct proof of publication from the ledger. Importantly, it must be impossible for law enforcement and judges to generate this information independently; if it were possible, it would be easy for these parties to circumvent the accountability mechanism.

We give a formal definition of this language $\mathcal{L}$ below. We denote the view of a user P_i as $\mathcal{V}_{P_i}$, where this view is a collection of the views of running

[5] As specified in the ideal functionality, during verification it will be checked that a warrant was properly requested and activated.

all algorithms that appear. We abuse notation slightly and denote the protocol transcript resulting from a sender P_S sending a message m to P_R as $\Pi_{\mathrm{A}}.\mathsf{SendMessage}(\cdot, P_S, P_R, m)$

$$\mathcal{L} = \left\{ (\mathsf{aux}, \mathsf{sid}) \;\middle|\; \exists \left(w, c, \left\{ \begin{array}{c} \mathcal{V}_{P_{\mathrm{LE}}}, \mathcal{V}_{P_J}, \\ \\ \mathcal{V}_{P_0}, \ldots, \mathcal{V}_{P_n} \end{array} \right\} \right) \text{ s.t.} \right.$$

$$\left. \begin{array}{r} c, \mathsf{aux} \leftarrow \Pi_{\mathrm{A}}.\mathsf{SendMessage}(\mathsf{sid}, P_S, P_R, m), \\ (\mathsf{Approve}) \leftarrow \Pi_{\mathrm{A}}.\mathsf{RequestWarrant}(\mathsf{sid}, w), \\ (\mathsf{NotifyWarrant}, t(w)) \leftarrow \Pi_{\mathrm{A}}.\mathsf{ActivateWarrant}(\mathsf{sid}, w), \\ 1 \leftarrow \Pi_{\mathrm{A}}.\mathsf{VerifyWarrantStatus}(\mathsf{sid}, w, \mathsf{aux}, c) \end{array} \right\}$$

In this language, the statement comprises some specified metadata and a valid instance of the protocol Π_{A} from the perspectives of the parties P_{LE}, P_J, and the users P_i without the sender and receiver. This setup specifies all the relevant components of the protocol (including the ledger functionality, in the case of the protocol presented in Section 5.5). The witness is a valid transcript starting with that setup, that includes the sending party sending a message with the appropriate metadata and concludes with a call to $\Pi_{\mathrm{A}}.\mathsf{VerifyWarrantStatus}$ that returns 1. Note that if $\mathsf{VerifyWarrantStatus}$ returns 1, then in the real protocol, $\mathsf{AccessMessage}$ would return the relevant plaintext. Unlike other common witness encryption languages, we note that all correctly sampled statements are trivially in the language and have multiple witnesses. Therefore, we need the strong notion of extractable witness encryption. As we will discuss, finding a witness for the statement remains a difficult task.

Consider the implications if it were computationally feasible for an adversary to generate a witness for an honestly sampled statement for $\mathcal{L}$. This would imply that an adversary corrupting P_{LE} and P_J interacting with the real protocol has a correct witness, which includes a call to ActivateWarrant, this implies our accountability property. Such a protocol could never succeed in meeting our original goals; law enforcement would always be able to simulate the steps required for proper accountability. An accountability mechanism that can be locally simulated cannot guarantee that all parties can monitor the mechanism, undermining the purpose of the protocol.

To formalize this intuition, we begin by describing an extractable witness encryption scheme Π_{EWE} for language $\mathcal{L}$ given access to an ARLEAS protocol Π_{A}.

- $\mathsf{Enc}(x, m)$ parses $(\mathsf{aux}, \mathsf{sid})$ from x and calls $\Pi_{\text{A}}.\mathsf{SendMessage}(\mathsf{sid}, m, P_S, P_R)$ such that it outputs aux, c. It then returns the views $\{\mathcal{V}_{P_{\text{LE}}}, \mathcal{V}_{P_J}, \mathcal{V}_{P_0}, \ldots, \mathcal{V}_{P_n}\}$ resulting from that run, excluding the private information associated with sending the message.

- $\mathsf{Dec}(c, \omega)$ parses $c, w, \mathsf{aux}, \mathsf{sid}$ from c and ω, calls $m \leftarrow \Pi_{\text{A}}.\mathsf{AccessMessage}(\mathsf{sid}, w, \mathsf{aux}, c)$ and returns the result.

It is easy to see that this construction satisfies the correctness property of extractable witness encryption. Notice that a valid witness needs to contain inputs to VerifyWarrantStatus such that it outputs 1. Because VerifyWarrantStatus is defined to return 1 exactly when AccessMessage will return a message, the above decryption algorithm will return a message only with a valid witness.

We introduce the metadata in the statement in order to fix a witness to a particular statement. Note that our protocol generates an encryption as running part of the protocol, actually generating part of the witness. If metadata is not included in the statement, then *any* witness for a particular setup can be used to decrypt *any* ciphertext generated by the encryption oracle under the same statement. While this is not inherently problematic for extractable witness encryption, it no longer corresponds neatly to ARLEAS. Recall that warrants in ARLEAS specify the metadata for which they are relevant through the in-scope funtionality i and this property must be enforced in the language.

We now proceed to our main result, showing that the above scheme Π_{EWE} satisfies extractable security if Π_{A} UC-realizes $\mathcal{F}_{\text{ARLEAS}}^{\ell,t,p,i,\mathsf{ret}}$.

Theorem 7 *Given a protocol Π_A that UC-realizes $\mathcal{F}_{ARLEAS}^{\ell,t,p,i,\mathsf{ret}}$, Π_{EWE} is a secure extractable witness encryption scheme for the language $\mathcal{L}$.*

Security Proof. Given an adversary $\mathcal{A}$ that has non-negligible advantage in the extractable witness encryption game for language $\mathcal{L}$, either

1. We construct an extractor $\mathsf{Ext}_{\mathcal{A}}(1^{\lambda}, x, aux)$ by verifying if the adversary $\mathcal{A}$ ran the following parts of Π_{A}:

   - $\Pi_{\text{A}}.\mathsf{RequestWarrant}(\mathsf{sid}, w)$
   - $\Pi_{\text{A}}.\mathsf{ActivateWarrant}(\mathsf{sid}, w)$

   such that $\Pi_{\text{A}}.\mathsf{VerifyWarrantStatus}(\mathsf{sid}, w, \mathsf{aux}, c) = 1$. If this was the case, the extractor would have all information to form a witness that it can output;

2. else, if such extractor does not exist, we construct a distinguisher $\mathcal{Z}$ that distinguishes between Π_{A} and ARLEAS ideal functionality. $\mathcal{Z}$ proceeds as follows

   (a) When $\mathcal{A}$ asks to sample a statement, $\mathcal{Z}$ instantiates Π_{A} with parties $\{P_{\text{LE}}, P_J, P_0, \ldots P_n, P_S, P_R\}$ on honest random coins. $\mathcal{Z}$ then generates some arbitrary metadata aux associated with a message that P_S could send in the future. and returns $\mathsf{aux}, \mathsf{sid}$ to $\mathcal{A}$.

   (b) When $\mathcal{A}$ sends the challenge plaintexts m_0, m_1 (such that $|m_0| = |m_1|$) on statement x, $\mathcal{Z}$ then flips a coin $b \xleftarrow{\$} \{0,1\}$, $\mathcal{Z}$ has P_S call $\Pi_{\text{A}}.\mathsf{SendMessage}(\mathsf{sid}, m_b, P_S, P_R)$ such that it outputs c, aux. $\mathcal{Z}$ then returns the updated views of P_{LE}, P_J and the N other users to $\mathcal{A}$.

   (c) When $\mathcal{A}$ outputs the guess b' and halts, $\mathcal{Z}$ outputs $b' == b$, where 1 indicates the real world and 0 indicates the ideal world.

   Note that in the ideal functionality, the joint views of law enforcement and the judge contain no information about the plaintext, because the ciphertext is chosen by the ideal world adversary without access to the plaintext. As such, if the adversary is able to distinguish between messages with non-negligible probability, $\mathcal{Z}$ must be interacting with the real world protocol.

Abuse Resistant Law Enforcement Access Systems References

Signal. *Signal Secure Messaging System*. URL: https://signal.org/.

WhatsApp (2017). *WhatsApp Encryption Overview*. Available at https://scontent.whatsapp.net/v/t61/68135620_760356657751682_6212997528851833559_n.pdf/WhatsApp-Security-Whitepaper.pdf?_nc_ohc=KLWdfyGXHtkAX_XDL-n&_nc_ht=scontent.whatsapp.net&oh=8436499c2708501873e5bae88ce96c6e&oe=5E450BE5.

Apple. *iMessage*. Available at https://support.apple.com/explore/messages. URL: https://support.apple.com/explore/messages.

Apple. *FaceTime*. Available at https://apps.apple.com/us/app/facetime/id1110145091. URL: https://apps.apple.com/us/app/facetime/id1110145091.

Apple. *iCloud security overview*. Available at https://support.apple.com/en-us/HT202303. URL: https://support.apple.com/en-us/HT202303.

Google. *Encrypt your data - Pixel Phone Help*. Available at https://support.google.com/pixelphone/answer/2844831?hl=en. URL: https://support.google.com/pixelphone/answer/2844831?hl=en.

Watt, Nicholas, Rowena Mason, and Ian Traynor (2015). "David Cameron pledges anti-terror law for internet after Paris attacks". In: *The Guardian*. URL: \url{https://www.theguardian.com/uk-news/2015/jan/12/david-cameron-pledges-anti-terror-law-internet-paris-attacks-nick-clegg}.

Franceschi-Bicchierai, Lorenzo (2014). "FBI Director: Encryption Will Lead to a 'Very Dark Place'". In: *Mashable*. URL: \url{https://mashable.com/2014/10/16/fbi-director-encryption-going-dark-speech/}.

Barr, William (2019). *Attorney General William P. Barr Delivers Keynote Address at the International Conference on Cyber Security*. Available at https://

www.justice.gov/opa/speech/attorney-general-william-p-barr-delivers-keynote-address-international-conference-cyber.

Federal Bureau of Investigation. *Going Dark*. Available at https://www.fbi.gov/services/operational-technology/going-dark.

Poplin, Cody M. (2016). "Burr-Feinstein Encryption Legislation Officially Released". In: *Lawfare*. URL: \url{https://www.lawfareblog.com/burr-feinstein-encryption-legislation-officially-released}.

Tarabay, Jamie (2018). "Australian Government Passes Contentious Encryption Law". In: *The New York Times*. URL: \url{https://www.nytimes.com/2018/12/06/world/australia/encryption-bill-nauru.html}.

Abelson, Harold, Ross Anderson, Steven M. Bellovin, Josh Benaloh, Matt Blaze, Whitfield Diffie, John Gilmore, Matthew Green, Susan Landau, Peter G. Neumann, Ronald L. Rivest, Jeffrey I. Schiller, Bruce Schneier, Michael A. Specter, and Daniel J. Weitzner (2015). "Keys under doormats: mandating insecurity by requiring government access to all data and communications". In: *Journal of Cybersecurity* 1.1, pp. 69–79. ISSN: 2057-2085. DOI: 10.1093/cybsec/tyv009. eprint: https://academic.oup.com/cybersecurity/article-pdf/1/1/69/7002861/tyv009.pdf. URL: https://doi.org/10.1093/cybsec/tyv009.

National Academies of Sciences, Engineering, and Medicine (2016). *Exploring Encryption and Potential Mechanisms for Authorized Government Access to Plaintext*. The National Academies Press.

Sing, Manish (2020). "Over two dozen encryption experts call on India to rethink changes to its intermediary liability rules". In: *TechCrunch*. URL: \url{https://techcrunch.com/2020/01/09/over-two-dozen-encryption-experts-call-on-india-to-rethink-changes-to-its-intermediary-liability-rules/}.

Denning, Dorothy E. (1994). "The US key escrow encryption technology". In: *Computer Communications* 17.7, pp. 453–457. DOI: 10.1016/0140-3664(94)90099-X. URL: https://doi.org/10.1016/0140-3664(94)90099-X.

Savage, Stefan (2018). "Lawful Device Access without Mass Surveillance Risk: A Technical Design Discussion". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, pp. 1761–1774. ISBN: 9781450356930. DOI: 10.1145/3243734.3243758. URL: https://doi.org/10.1145/3243734.3243758.

Bellovin, Steven M., Matt Blaze, Dan Boneh, Susan Landau, and Ronald R. Rivest (2018). *Analysis of the CLEAR Protocol per the National Academies'*

Framework. Tech. rep. CUCS-003-18. Columbia University. URL: \url{https://mice.cs.columbia.edu/getTechreport.php?techreportID=1637}.

Wright, Charles and Mayank Varia (2018). "Crypto Crumple Zones: Enabling Limited Access without Mass Surveillance". In: *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pp. 288–306. DOI: 10.1109/EuroSP.2018.00028.

Tait, Matt (2016). "An Approach to James Comey's Technical Challenge". In: *Lawfare*. URL: \url{https://www.lawfareblog.com/approach-james-comeys-technical-challenge}.

Levy, Ian and Crispin Robinson (2018). "Principles for a More Informed Exceptional Access Debate". In: *Lawfare*. URL: \url{https://www.lawfareblog.com/principles-more-informed-exceptional-access-debate}.

Bellare, Mihir and Ronald L. Rivest (1999). "Translucent Cryptography - An Alternative to Key Escrow, and Its Implementation via Fractional Oblivious Transfer". In: *Journal of Cryptology* 12.2, pp. 117–139. DOI: 10.1007/PL00003819.

Denning, Dorothy E and Dennis K Branstad (1996). "A taxonomy for key escrow encryption systems". In: *Communications of the ACM* 39.3, pp. 34–40.

Encryption Working Group (2019). *Moving the Encryption Policy Conversation Forward*. Tech. rep. Carnegie Endowment for International Peace. URL: \url{https://carnegieendowment.org/files/EWG__Encryption_Policy.pdf}.

National Academies of Sciences, Engineering, and Medicine (2018). *Decrypting the Encryption Debate: A Framework for Decision Makers*. Washington, DC: The National Academies Press. URL: \url{https://www.nap.edu/catalog/25010/decrypting-the-encryption-debate-a-framework-for-decision-makers}.

Nightingale, Johnathan (2011). *Fraudulent *.google.com Certificate*. Mozilla Security Blog, https://blog.mozilla.org/security/2011/08/29/fraudulent-google-com-certificate/.

Bryan-Low, Cassell (2006). "Vodafone, Ericsson Get Hung Up In Greece's Phone-Tap Scandal". In: *The Wall Street Journal*. URL: \url{https://www.wsj.com/articles/SB115085571895085969}.

Nakashima, Ellen (2013). "Chinese hackers who hacked Google gained access to sensitive data, U.S. officials say". In: *The Washington Post*. URL: \url{https://www.washingtonpost.com/world/national-security/chinese-hackers-who-breached-google-gained-access-to-sensitive-}

data-us-officials-say/2013/05/20/51330428-be34-11e2-89c9-3be8095fe767_story.html}.

Gorman, Siobhan (2013). "NSA Officers Spy on Love Interests". In: *The Wall Street Journal*. URL: \url{https://blogs.wsj.com/washwire/2013/08/23/nsa-officers-sometimes-spy-on-love-interests/}.

Lichtblau, Eric and Joseph Goldstein (2016). "Apple Faces U.S. Demand to Unlock 9 More iPhones". In: *The New York Times*. URL: \url{https://www.nytimes.com/2016/02/24/technology/justice-department-wants-apple-to-unlock-nine-more-iphones.html}.

Blaze, Matt (1996). "Oblivious key escrow". In: *Information Hiding*. Ed. by Ross Anderson. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 335–343. ISBN: 978-3-540-49589-5.

Canetti, Ran (2001). "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

Hofheinz, Dennis (2012). "All-But-Many Lossy Trapdoor Functions". In: *EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. LNCS. Springer, Heidelberg, pp. 209–227. DOI: 10.1007/978-3-642-29011-4_14.

Choudhuri, Arka Rai, Matthew Green, Abhishek Jain, Gabriel Kaptchuk, and Ian Miers (2017). "Fairness in an Unfair World: Fair Multiparty Computation from Public Bulletin Boards". In: *ACM CCS 2017*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. ACM Press, pp. 719–728. DOI: 10.1145/3133956.3134092.

Goyal, Rishab and Vipul Goyal (2017). "Overcoming Cryptographic Impossibility Results Using Blockchains". In: *TCC 2017, Part I*. Ed. by Yael Kalai and Leonid Reyzin. Vol. 10677. LNCS. Springer, Heidelberg, pp. 529–561. DOI: 10.1007/978-3-319-70500-2_18.

Kaptchuk, Gabriel, Matthew Green, and Ian Miers (2019). "Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers". In: *NDSS 2019*. The Internet Society.

Scafuro, Alessandra (2019). "Break-glass Encryption". In: *PKC 2019, Part II*. Ed. by Dongdai Lin and Kazue Sako. Vol. 11443. LNCS. Springer, Heidelberg, pp. 34–62. DOI: 10.1007/978-3-030-17259-6_2.

Canetti, Ran, Hugo Krawczyk, and Jesper Buus Nielsen (2003). "Relaxing Chosen-Ciphertext Security". In: *CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. LNCS. Springer, Heidelberg, pp. 565–582. DOI: 10.1007/978-3-540-45146-4_33.

Bresson, Emmanuel, Dario Catalano, and David Pointcheval (2003). "A Simple Public-Key Cryptosystem with a Double Trapdoor Decryption Mechanism and Its Applications". In: *ASIACRYPT 2003*. Ed. by Chi-Sung Laih. Vol. 2894. LNCS. Springer, Heidelberg, pp. 37–54. DOI: 10.1007/978-3-540-40061-5_3.

Peikert, Chris and Brent Waters (2008). "Lossy trapdoor functions and their applications". In: *40th ACM STOC*. Ed. by Richard E. Ladner and Cynthia Dwork. ACM Press, pp. 187–196. DOI: 10.1145/1374376.1374406.

Bellare, Mihir, Dennis Hofheinz, and Scott Yilek (2009). "Possibility and Impossibility Results for Encryption and Commitment Secure under Selective Opening". In: *EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. LNCS. Springer, Heidelberg, pp. 1–35. DOI: 10.1007/978-3-642-01001-9_1.

Hemenway, Brett, Benoît Libert, Rafail Ostrovsky, and Damien Vergnaud (2011). "Lossy Encryption: Constructions from General Assumptions and Efficient Selective Opening Chosen Ciphertext Security". In: *ASIACRYPT 2011*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. LNCS. Springer, Heidelberg, pp. 70–88. DOI: 10.1007/978-3-642-25385-0_4.

Choudhuri, Arka Rai, Vipul Goyal, and Abhishek Jain (2019). "Founding Secure Computation on Blockchains". In: *EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. LNCS. Springer, Heidelberg, pp. 351–380. DOI: 10.1007/978-3-030-17656-3_13.

Kiayias, Aggelos, Alexander Russell, Bernardo David, and Roman Oliynykov (2017). "Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol". In: *CRYPTO 2017, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. LNCS. Springer, Heidelberg, pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12.

David, Bernardo, Peter Gazi, Aggelos Kiayias, and Alexander Russell (2018). "Ouroboros Praos: An Adaptively-Secure, Semi-synchronous Proof-of-Stake Blockchain". In: *EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. LNCS. Springer, Heidelberg, pp. 66–98. DOI: 10.1007/978-3-319-78375-8_3.

Gazi, Peter, Aggelos Kiayias, and Dionysis Zindros (2019). "Proof-of-Stake Sidechains". In: *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, pp. 139–156. DOI: 10.1109/SP.2019.00040.

Pietrzak, Krzysztof (2019). "Simple Verifiable Delay Functions". In: *ITCS 2019*. Ed. by Avrim Blum. Vol. 124. LIPIcs, 60:1–60:15. DOI: 10.4230/LIPIcs.ITCS.2019.60.

Boneh, Dan, Joseph Bonneau, Benedikt Bünz, and Ben Fisch (2018). "Verifiable Delay Functions". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, pp. 757–788. DOI: 10.1007/978-3-319-96884-1_25.

Boyle, Elette, Kai-Min Chung, and Rafael Pass (2014). "On Extractability Obfuscation". In: *TCC 2014*. Ed. by Yehuda Lindell. Vol. 8349. LNCS. Springer, Heidelberg, pp. 52–73. DOI: 10.1007/978-3-642-54242-8_3.

Cramer, Ronald and Victor Shoup (2002). "Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption". In: *EUROCRYPT 2002*. Ed. by Lars R. Knudsen. Vol. 2332. LNCS. Springer, Heidelberg, pp. 45–64. DOI: 10.1007/3-540-46035-7_4.

Garg, Sanjam, Rafail Ostrovsky, Ivan Visconti, and Akshay Wadia (2012). "Resettable Statistical Zero Knowledge". In: *TCC 2012*. Ed. by Ronald Cramer. Vol. 7194. LNCS. Springer, Heidelberg, pp. 494–511. DOI: 10.1007/978-3-642-28914-9_28.

Garg, Sanjam, Craig Gentry, Shai Halevi, and Daniel Wichs (2014). "On the Implausibility of Differing-Inputs Obfuscation and Extractable Witness Encryption with Auxiliary Input". In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, pp. 518–535. DOI: 10.1007/978-3-662-44371-2_29.

Bates, Adam M., Kevin R. B. Butler, Micah Sherr, Clay Shields, Patrick Traynor, and Dan S. Wallach (2012). "Accountable Wiretapping -or- I know they can hear you now". In: *NDSS 2012*. The Internet Society.

Segal, Aaron, Bryan Ford, and Joan Feigenbaum (2014). "Catching Bandits and Only Bandits: Privacy-Preserving Intersection Warrants for Lawful Surveillance". In: *4th USENIX Workshop on Free and Open Communications on the Internet (FOCI 14)*. San Diego, CA: USENIX Association. URL: https://www.usenix.org/conference/foci14/workshop-program/presentation/segal.

Liu, J., M. D. Ryan, and L. Chen (2014). "Balancing Societal Security and Individual Privacy: Accountable Escrow System". In: *2014 IEEE 27th Computer Security Foundations Symposium*, pp. 427–440. DOI: 10.1109/CSF.2014.37.

Kroll, J, E Felten, and Dan Boneh (2014). "Secure protocols for accountable warrant execution". In:

Kroll, Joshua A, Joe Zimmerman, David J Wu, Valeria Nikolaenko, Edward W Felten, and Dan Boneh. "Accountable Cryptographic Access Control". In:

Backes, Michael, Jan Camenisch, and Dieter Sommer (2005). "Anonymous yet Accountable Access Control". In: *Proceedings of the 2005 ACM Workshop on*

Privacy in the Electronic Society. WPES '05. Alexandria, VA, USA: Association for Computing Machinery, pp. 40–46. ISBN: 1595932283. DOI: 10.1145/1102199.1102208. URL: https://doi.org/10.1145/1102199.1102208.

Goldwasser, Shafi and Sunoo Park (2017). "Public Accountability vs. Secret Laws: Can They Coexist? A Cryptographic Proposal". In: *Proceedings of the 2017 on Workshop on Privacy in the Electronic Society*. WPES '17. Dallas, Texas, USA: Association for Computing Machinery, pp. 99–110. ISBN: 9781450351751. DOI: 10.1145/3139550.3139565. URL: https://doi.org/10.1145/3139550.3139565.

Frankle, Jonathan, Sunoo Park, Daniel Shaar, Shafi Goldwasser, and Daniel J. Weitzner (2018). "Practical Accountability of Secret Processes". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, pp. 657–674.

Servan-Schreiber, Sacha and Archer Wheeler (2019). *Judge, Jury & Encryptioner: Exceptional Access with a Fixed Social Cost*. arXiv: 1912.05620 [cs.CR].

Panwar, Gaurav, Roopa Vishwanathan, Satyajayant Misra, and Austin Bos (2019). "SAMPL: Scalable Auditability of Monitoring Processes using Public Ledgers". In: *ACM CCS 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, pp. 2249–2266. DOI: 10.1145/3319535.3354219.

Chapter 6

Conclusion

In this work, we have explored new applications of public ledgers. In Chapter 3, we presented a new computational paradigm called enclave ledger interaction (ELI). ELI defines a protocol between a trusted execution environment (*e.g.* TEEs or cryptographic obfuscation) and a public ledger, mediated by a host machine. The first application of ELI is to provide enclaves protection against rollback attacks, in which a malicious host attempts to re-execute the enclave on old state with new inputs. More importantly, ELI allows for general purpose, conditional execution over secret data. Put another way, the program embedded in the enclave can execute in response to information posted on the public ledger. This can be used to realize an array of applications, including smart contracts executing on secret data, limited password guessing, and autonomous ransomware.

In Chapters 4, we explored an additional application of conditional execution: achieving fairness in MPC. Since the 1980's, it has been well known that achieving fairness in the presence of a dishonest majority of protocol

participants is impossible. This means that fair multiparty computation is impossible in the dishonest majority setting. We have showed that it is possible to overcome this impossibility if the protocol participants have access to a public ledger. Importantly, the public ledger is innately fair — once a piece of information is posted on the ledge, all players can retrieve that information. The protocol participants can use a conditional execution primitive to make recovering the output of the MPC protocol contingent on posting information on a public ledger. Using this approach, we show that fairness can be achieved, either assuming the existence of witness encryption or TEEs.

Finally, in Chapter 5, we showed that conditional execution can be used to build an abuse resistant law enforcement access system. The system we described ensures that law enforcement must post transparency information before decrypting a ciphertext. Transparency of this kind protects against government abuse of power, as well as key material theft by a external malicious actors. We gave a workable construction of such a system, providing law enforcement only needs to access messages encrypted after a warrant was issued and activated. If law enforcement requires the ability to perform retrospective surveillance, the existence of such a system implies the existence of extractable witness encryption for some non-trivial languages.

These applications are all different embodiments of conditioning execution on public events. More specifically, fairness in MPC and abuse resistant law enforcement access systems require only conditional decryption. We believe there are many more applications of conditional execution that have yet to be explored, particularly when considering issues of transparency. The

technology underpinning public ledgers will continue to evolve and become more efficient. If public ledgers are only used to enable cryptocurrencies, the potential impact of public ledgers will be wasted. To harness the true power of public ledgers, we must continue to find novel applications, like those discussed in this work.