CacheCash: A Cryptocurrency-based Decentralized Content Delivery Network

Ghada Almashaqbeh

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2019

ABSTRACT

CacheCash: A Cryptocurrency-based Decentralized Content Delivery Network

Ghada Almashaqbeh

Online content delivery has witnessed dramatic growth recently with traffic consuming over half of today's Internet bandwidth. This escalating demand has motivated content publishers to move outside the traditional solutions of infrastructure-based content delivery networks (CDNs). Instead, many are employing peer-to-peer data transfers to reduce the service cost and avoid bandwidth over-provision to handle peak demands. Unfortunately, the open access work model of this paradigm, which allows anyone to join, introduces several design challenges related to security, efficiency, and peer availability.

In this dissertation, we introduce CacheCash, a cryptocurrency-based decentralized content distribution network designed to address these challenges. CacheCash bypasses the centralized approach of CDN companies for one in which end users organically set up new caches in exchange for cryptocurrency tokens. Thus, it enables publishers to hire caches on an as-needed basis, without constraining these parties with long-term business commitments.

To address the challenges encountered as the system evolved, we propose a number of protocols and techniques that represent basic building blocks of CacheCash's design. First, motivated by the observation that conventional security assessment tools do not suit cryptocurrency-based systems, we propose *ABC*, a threat modeling framework capable of identifying attacker collusion and the new threat vectors that cryptocurrencies introduce. Second, we propose *CAPnet*, a defense mechanism against cache accounting attacks (i.e., a client pretends to be served allowing a colluding cache to collect rewards without doing any work). CAPnet features a bandwidth expenditure puzzle that clients must solve over the content before caches are given credit, which bounds the effectiveness of this collusion case. Third, to make it feasible to reward caches per data chunk served, we introduce *MicroCash*, a decentral-

ized probabilistic micropayment scheme that reduces the overhead of processing these small payments. MicroCash implements several novel ideas that make micropayments more suitable for delay-sensitive applications, such as online content delivery.

CacheCash combines the previous techniques to produce a novel service-payment exchange protocol that secures the content distribution process. This protocol utilizes gradual content disclosure and partial payment collection to encourage the honest collaborative work between participants. We present a detailed game theoretic analysis showing how to exploit rational financial incentives to address several security threats. This is in addition to various performance optimization mechanisms that promote system efficiency and scalability. Lastly, we evaluate system performance and show that modest machines can serve/retrieve content at a high bitrate with minimal overhead.

Contents

List of Figures

List of Tables

Acknowledgements

Along the path, many people have enriched my experience in graduate school: family, collaborators, and friends.

My deepest gratitude goes to my advisers Allison Bishop, Tal Malkin, and Justin Cappos. Thanks Allison for being such an amazing mentor and a dear friend, and for the invaluable research skills I have learned from you. Tal, thanks for teaching me how to enjoy research, how to see the big picture of my work, and what a great teacher should be. And thanks Justin for the priceless guidance that shaped my way of thinking about research problems and for the continued support.

Also, I would like to thank Marina Blanton, my former adviser at University of Notre Dame, for her great help and advice.

I would like to express my appreciation to all my collaborators on this, and other, work: Fabrice Benhamouda, Allison Bishop, Marina Blanton, Justin Cappos, Yaniv Erlich, Jonny Gershoni, Kevin Kelley, Itsik Pe'er, Tal Rabin, Eran Tromer, and Yihua Zhang. In addition, I would like to thank Suman Jana for the helpful feedback and interesting discussions while serving as a committee member on my candidacy exam, my thesis proposal, and my dissertation defense.

This dissertation would not be in its current shape without the expertise of Lois Anne DeLong. The technical writing skills I've gained while working with her will accompany me for the rest of my life.

I am extremely thankful for all the friends I have met at Columbia, NYU, and Notre Dame, and those whom I have met outside school. I will never forget the wonderful time we had together over the past years.

No words can express how grateful I am for my parents and my siblings. I would have needed longer than a lifetime to describe their unconditional love and support.

Thanks for encouraging me to chase my dreams. Thanks for all the joy you bring to my life. Thanks for being who you are!

To Ali, my dearest brother

Chapter 1

Introduction

Online content distribution has seen dramatic growth over the last decade. Video streaming, in particular occupies more than 60% of today's Internet traffic and is projected to exceed 80% by 2022 [22]. To meet this huge demand, content publishers typically distribute the load among geographically dispersed caches, an action that has fostered the development of infrastructure-based content delivery networks (CDNs) [3, 23]. These centrally-organized networks have proven effective in handling this task. However, they require publishers to over-provision the needed bandwidth to handle peak demands [93], and to establish expensive and complex business relationships with CDN providers [68, 90]. As demand for these services grows, the need for more efficient and cheaper solutions have lead to radically different approaches [59, 132].

One such approach is the use of peer-to-peer (P2P) content distribution models, which can reduce costs [89, 120], while also enabling access to lower latency caches. Several studies have shown that up to $\sim$ 88% of the traffic can be offloaded from infrastructure-based CDN nodes to peers during peak demand hours [59,64,89,92]. In addition, these P2P-based paradigms can provide wide network coverage, scale easily with demand [128], and, when managed carefully, offer a good quality of service to end users [60, 132].

However, a major stumbling block for such models is the limited availability of peers ready to donate time and bandwidth. It is not unusual for participants to join the system just to receive the service and then exit without serving others [119]. Exchanging the correct service for monetary incentives has proven effective in mitigating this problem [62]. That is, by forming a bandwidth market, selfish peers are

willing to serve others to collect more payments. This creates robust and flexible systems from which content publishers can build dynamic CDNs tailored to their service specifications.

Nonetheless, previous systems relied either on centralized payment services, or placed trust in content publishers to handle this task. Cryptocurrencies and blockchain technologies offer a fully distributed, and trustless, way to reward peers for the expended resources. Since Bitcoin was introduced in 2009 [107], the number of these "digital currencies" has grown into the thousands by 2019, with a total market capital exceeding $175 billion [24]. Although early systems focused only on providing a virtual currency exchange medium [40,107], nowadays there is increasing interest in providing other types of distributed services on top of this medium, such as computation outsourcing [27] and file storage [49, 130]. These newer applications lay down a potential framework for building service-payment exchange systems using P2P networks.

In this dissertation, we propose and implement CacheCash, a fully distributed, cryptocurrency-based, bandwidth market. CacheCash introduces a number of modules and protocols needed to secure the content delivery process and optimize its costs. These include ABC, a threat modeling framework that helps in investigating the full threat space of cryptocurrency-based distributed services. CAPnet, a defense mechanism that protects against colluding caches and clients who try to collect rewards without performing the promised work. MicroCash, a probabilistic micropayment scheme that allows rewarding caches at fine-grained scale. This is in addition to various financially-based defense mechanisms, and efficiency optimization techniques, to encourage the honest behavior while reducing the overall overhead.

1.1 Design Challenges of Monetary-Incentivised Distributed CDNs

Despite the many advantages, the open access work model of P2P content distribution introduces multiple security and performance challenges that have hindered its

adoption in practice. In this work, we focus on the following issues.

Impossibility of fair-exchange. Lacking a trusted party to ensure that each entity performs as expected is one such issue. It means that the system must deal with the impossibility of fair-exchange between mutually-untrusted parties [76, 108]. In the context of content delivery, this means that a cache may not provide a service if paid in advance, or alternatively, a publisher may not pay after having its clients served. Moreover, there is the risk of service noncompliance, where a cache may deliver corrupted content while collecting full payments, or may not commit to the service amount and quality a publisher is anticipating. Therefore, there is a need for secure service-payment exchange protocols that reduce the risks resulted from this inevitable type of behavior.

Cache accounting attacks. Another security challenge is that peer-assisted work models are vulnerable to cache accounting attacks [51,95], in which a cache and client collude to defraud the content publisher by claiming to have transferred data (and claiming payment) when no actual work has been done. This is a particular problem in content distribution applications that do not require subscription fees from clients, such as ad-funded video streaming [48], or services that allow a client to play content on multiple devices under one subscription [36], or even in situations where a client is not interested in the content, e.g., a cache running as a client. This has been confirmed through several empirical studies. For example, Lian et al. [95] showed that such a collusion case allowed malicious parties in the Maze file sharing system to collect incentive scores for free. In addition, Aditya et al. [51] documented how a small number of colluding peers can cause significant log inaccuracies in the Akamai Netsession interface.

Although some defenses against these attacks have been proposed [51, 113], they do not work in typical peer-to-peer scenarios. This is because they either rely on service activity logs from the participants themselves [51], which could be fabricated. Or they require a knowledge of the peer computation power and link latency [113], information that could be hard to obtain or estimate. Supporting a a P2P-based content distribution service requires an effective defense mechanism that lets untrusted,

anonymous nodes serve as caches.

Micropayments. Given that peer-assisted models rely on the collaborative work of mutually-untrusted parties, micropayments (or payments in pennies) are usually used to compensate for the service. This payment paradigm provides a great deal of flexibility for clients who may switch servers or stop a service at anytime. It also reduces the financial risks of the service-payment exchange process. That is, a server loses only a small amount of money if a client does not pay for the service, and vice versa.

However, processing these small payments individually is problematic. It can be expensive due to high transaction fees that may exceed the few pennies received. For example, the average base cost of a debit or credit card transaction in the US is around 21 to 24 cents, and 23 to 42 cents, respectively [19,20]. In cryptocurrencies such a fee could be even higher, e.g., around $1 in Bitcoin as of mid April 2018 [16]. Beside this financial drawback, handling micropayments individually can impose a huge workload on the system, and may explode the payment log needed for accountability purposes.

Probabilistic micropayment schemes have emerged as a potential solution to address these problems [99, 114, 115, 129]. In these models, payments take the form of lottery tickets that are exchanged locally between parties, and only winning tickets can be redeemed for currency. This reduces the number of processed transactions in the system. Unfortunately, these older solutions rely on a trusted party to audit the lottery and manage payments. Such a centralized approach may increase the deployment cost and limit the use of the payment service to systems of fully authenticated participants [70].

Newer, fully distributed, solutions emerged by utilizing cryptocurrencies [70,111], where they replaced the trusted party with the miners, and exploited the blockchain to provide public verifiability of system operation. Yet, these decentralized approaches force a payer to issue micropayments sequentially using the same escrow, which means that a new ticket cannot be issued until it is confirmed that the previous one did not win. In addition, these schemes rely on computationally-heavy cryptographic primitives [70,111], and several rounds of communication to exchange payments [70].

Consequently, there is still a need for a decentralized scheme that supports concurrent micropayments at lower computational and bandwidth overhead. This is particularly important in online content delivery services that may not tolerate high delays.

Handling these, and many other issues, in an efficient way is a pre-requisite for any practical deployment of a fully decentralized CDN service.

1.2 Limitations of Existing Solutions

Several systems have exploited the idea of exchanging bandwidth for rewards to encourage peer availability and the honest work, e.g., KARMA [126], Floodgate [106], Dandelion [120], and Hincent [93]. Unfortunately, these systems have several drawbacks that have hindered their adoption in practice. Floodgate [106] and Dandelion [120] assume that content publishers are trusted entities to mediate service-payment exchanges between peers and track payments. Hincent [93] is a centralized system that relies on a trusted CDN node to register all peers, track payments, and solve disputes. While KARMA [126] is fully distributed, it requires the exchange of large number of messages to confirm a payment or content delivery. As a result, it incurs large delays and bandwidth overhead. Furthermore, KARMA does not allow publishers to sponsor content retrieval for their clients, where the clients themselves have to pay for the service. In other words, it does not support the general work paradigm of CDNs where publishers pay caches for the distribution service, and clients need only to deal with the publisher for any financial obligations (e.g., subscription fees), if any.

The evolution of cryptocurrencies [107, 131] offers a potential solution for practical, decentralized, and trustless services. As mentioned previously, this is done by providing a distributed service on top of the currency exchange medium that cryptocurrencies manage. Systems related to our work include:

- Decentralized data and file storage services, such as Storj [130] and Filecoin [49]. Although these systems allow customers to retrieve the stored files, they do not allow publishers to sponsor the retrieval service for their clients.

- Generalized bandwidth exchange systems, such as Torcoin [79] and Mysterium [35]. The former rewards end users for providing their bandwidth to serve as relays in the Tor network, while the latter rewards the users who provide a virtual private network (VPN) service for others. These more general solutions do not specifically target content distribution. As a result, they do not provide tailored mechanisms for publishers to form CDNs or sponsor the content retrieval process for their clients.
- Distributed CDN services, which to the best of our knowledge, currently include only two systems, Gringotts [81] and NOIA [38]. Though these schemes come the closest to the system we propose, neither is fully decentralized. Gringotts still requires the intervention of a trusted traditional CDN node to defend against cache accounting attacks, and the current implementation of NOIA relies on a centralized cloud controller to authorize content managing entities. In addition, NOIA rewards caches based on activity reports from caches and clients without any defense against cache accounting attacks.

CacheCash attempts to address the aforementioned limitations by, first, being a fully decentralized and trustless system. Publishers manage their caches without any help from any trusted entity, and a distributed cryptocurrency system is used to handle payments. Second, CacheCash defends against cache accounting attacks in a fully decentralized way. Third, it supports a single goal of providing a CDN service. Hence, CacheCash allows publishers to sponsor the content delivery service for their clients, and its tools are customized based on the security and performance issues in CDNs.

The differences between CacheCash and the systems listed above are summarized in Table 1.1. Note that all systems that assume trusted publishers, or these that do not allow sponsoring the retrieval service, do not suffer from cache accounting attacks (for this, they have "N/A" in the table). Furthermore, some systems provide bandwidth incentives, where the credit a peer collects can be exchanged for bandwidth service only. Others provide monetary incentives, while some systems support hybrid incentive model, both bandwidth, if a peer can reciprocate the service for the one

Table 1.1: Comparison of P2P-based CDN solutions.

System	CDN-specific	Decentralized	Trustless Publishers	Incentive Type	Sponsoring Content Retrieval for Clients	Addressing Cache Accounting Attack
KARMA [126]	✓	✓	✓	Bandwidth	✗	N/A
Floodgate [106]	✓	✓	✗	Monetary	✓	N/A
Dandelion [120]	✓	✓	✗	Hybrid	✓	N/A
Hincent [93]	✓	✗	✓	Hybrid	✓	N/A
Storj [130] and Filecoin [49]	✗	✓	✓	Monetary	✗	N/A
Torcoin [79] and Mysterium [35]	✗	✓	✓	Monetary	✗	N/A
Gringotts [81]	✓	Partial	✓	Monetary	✓	Requires a trusted CDN node
NOIA [38]	✓	Partial	✓	Monetary	✓	✗
CacheCash	✓	✓	✓	Monetary	✓	✓

providing content, and monetary, if no reciprocation is available.

1.3 Thesis Statement

Given the usefulness and advantages of decentralized CDN services, and the limitations of existing solutions, more efforts are needed in order to develop secure, efficient, and cost-effective systems. These systems need to address the previously mentioned security and performance challenges. Towards this end, the thesis statement of this dissertation can be formulated as follows.

Goal: Building a content distribution service that can supply to any client a copy of the requested content that is identical to the one owned by the publisher. Such a service needs to enable publishers to form dynamic networks of caches without involving any trusted entity, and to sponsor content retrieval for their clients. The developed system should reward any cache with the promised payment for the service requests it handles, and ensure that every cache has earned its payments.

Non-goal: Several design issues related to CDN services are outside the scope of this work. These include digital rights management, preserving clients' privacy,

optimizing cache selection when handling content requests, optimizing cache management, i.e., how content is distributed among caches, and supporting variable data chunk size and variable cache set size during the content distribution process. Addressing these issues while preserving the low overhead of CacheCash is part of our future work.

1.4 Contributions

To fulfill our thesis statement, we propose CacheCash, a cryptocurrency-based decentralized content distribution network designed to address the challenges listed in Section 1.1 and the limitations of existing solutions discussed in Section 1.2. CacheCash bypasses the centralized approach of CDN companies for one in which end users organically set up new caches in exchange for cryptocurrency tokens. Thus, it creates a fully distributed and trustless bandwidth market that enables publishers to hire caches on an as-needed basis, without constraining these parties with long-term business commitments.

In order to support such a flexible service model, CacheCash introduces a novel service-payment exchange protocol that reduces the risks of dealing with distrusted parties in P2P networks. This protocol utilizes gradual content disclosure and partial payment collection to encourage honest collaboration between peers. This is achieved by devising CAPnet, a defense against cache accounting attacks, and a highly efficient probabilistic micropayment scheme, MicroCash, to permit fine-grained monetary rewards for caches at a low overhead. CacheCash's service-payment exchange protocol combines CAPnet and a modified version of MicroCash to ensure that caches earn their payments, and that publishers compensate for the service properly. The security of this protocol, which is guided by a thorough threat model built using our framework ABC, is enforced using both cryptographic approaches and rational financial incentives that make honesty the most profitable option.

In summary, the contributions of this dissertation include the following:

- We propose ABC (Chapter 2), an asset-based cryptocurrency-focused threat

modeling framework that is capable of identifying the complex collusion cases and new threat vectors that cryptocurrencies introduce. Its design is motivated by the observation that traditional threat modeling frameworks do not fit cryptocurrencies, thus leaving them vulnerable to unanticipated attacks.

- We conduct a user study to evaluate ABC against the popular threat modeling framework STRIDE [88]. We show how generalized threat models that are not designed for large-scale distributed systems, and do not consider the attacker's financial motivations or possible collusion between these attackers, could cause analysts to overlook serious threats to the system.
- We apply ABC to real world systems including two of our designs (MicroCash and CacheCash), in addition to Bitcoin [107] and Filecoin [49]. These use cases attest to the usefulness of ABC's tools as they integrated well into CacheCash's and MicroCash's design phases, and they revealed several missing threat scenarios in the public design of Filecoin.
- We propose CAPnet (Chapter 3), the first technique that lets untrusted caches, such as peers with unknown computational and latency characteristics, join a peer-assisted CDN while providing a bound on the effectiveness of accounting attacks. Our key innovation is a lightweight cache accountability puzzle that clients must solve before caches are given credit. The puzzle solution serves as a content retrieval confirmation to assure publishers that the claimed data transfer has taken place.
- We analyze the security of CAPnet, showing how to configure the system parameters to make honesty the most profitable option. We also evaluate its performance and demonstrate how the setup of the design parameters affect the productivity of content distribution.
- We introduce MicroCash (Chapter 4), the first decentralized probabilistic framework that supports concurrent micropayments using a single escrow and at a fast rate. MicroCash allows payment exchange using a single round of communication, and features a lightweight lottery protocol, requiring only hashing, to aggregate the small payments.

- We derive a lower bound for the payment escrow a payer needs in MicroCash to cover all payments with probability $1 - \epsilon$ (for some small ϵ). We also derive a lower bound for the penalty deposit needed to deter cheating using a game theoretic analysis.
- We evaluate the performance of MicroCash in comparison to MICROPAY [110] (a sequential probabilistic micropayment scheme). We show how concurrency reduces the amount of data logged on the blockchain, and how the lightweight design of our scheme optimizes the payment processing rate.
- We combine the previous techniques to secure the service-payment exchange process used in CacheCash (Chapter 5). We cover both the service setup and content distribution phases. These involve how publishers can form dynamic CDNs in the system, how caches can join these CDNs, and how client requests are handled to deliver the desired content.
- We devise a unique way to compute a cache payment value in order to minimize any losses potentially caused by cheating, and to stabilize the work relation between a cache and a publisher. Instead of collecting one lottery ticket per data chunk served, CacheCash divides this payment into two tickets that are collected at two stages. One when the chunk is served and one when this chunk is proved to be correct. CacheCash ties the currency value of both tickets together and makes it accumulate over time as the cache continues working with the same publisher.
- We present a thorough game theory analysis of the financial defenses that CacheCash adopts. This involves configuring the payment function used to reward caches, and devising mechanism to be used by caches to prioritize client traffic and select which publisher to work with in the system.
- We derive a lower bound for the payment escrow a publisher needs in CacheCash, under the two-ticket model, and a lower bound for the penalty deposit needed in this modified model.
- We evaluate the performance of CacheCash by conducting benchmarks to demonstrate the efficiency gain of the various performance optimization tech-

niques adopted in the system.

1.5 Dissertation Roadmap

The rest of this dissertation is organized as follows. Chapter 2 introduces ABC, including its design, a user study-based evaluation of its effectiveness, and a set of of real-world use cases. Next, our defense mechanism against cache accounting attacks, CAPnet, is presented in Chapter 3, along with a detailed analysis of its security and performance benchmarks. Chapter 4 introduces our low-overhead and concurrent micropayment scheme, MicroCash. This covers the system design, analyzing its security, and deriving bounds for the escrow balances needed for payment and penalty deposits, in addition to a thorough evaluation of the system efficiency. CacheCash, the core of this dissertation, is presented in Chapter 5 showing how it utilizes ABC to identify the impactfull threat cases, integrates CAPnet and a modified version of MicroCash in its service-payment exchange process, and secures the system operation using both cryptographic and financial techniques. Finally, this dissertation concludes in Chapter 6, where we discuss future directions to address more performance and security challenges in distributed CDNs.

Chapter 2

ABC: A Cryptocurrency-Focused Threat Modeling Framework

This chapter is based on joint work with Allison Bishop and Justin Cappos [53, 54].

2.1 Overview

Despite the many advantages they offer — decentralization, transparency, and lowered service costs — there is still a big gap between the promise of cryptocurrencies and their performance in practice. A major stumbling block is the perception that these systems are not secure, and the large number of security breaches announced in the past few years give credence to these doubts [14, 15, 17, 28, 43, 45, 46]. Therefore, a better understanding of the security of cryptocurrencies is needed in order to ensure their safe deployment in emerging applications.

The best practice for designing a secure system requires a threat modeling step to investigate potential security risks. Such a model can guide system designers in deploying the proper countermeasures, and assessing the security level of a system. Although threat modeling has been thoroughly studied in the literature, existing paradigms primarily target software applications [88] or distributed systems that have a small number of participant types [105]. These techniques were not designed to be scalable for a set of diverse, mutually distrustful parties as found in cryptocurrencies. Such systems, especially these providing distributed services, consist of parties that play different roles (miners, clients, and different types of servers), and an attacker may control any subset of these parties. Adding to the complexity, the attacker may seek to target any role in the system and may launch a diverse array of attacks with

different intended outcomes. The complexity of reasoning about and managing threat cases becomes unwieldy as these sets grow.

To address these issues, we propose ABC, an Asset-Based Cryptocurrency-focused threat modeling framework. ABC introduces a novel technique, called a collusion matrix, that allows users to investigate the full threat space and manage its complexity. A collusion matrix is a comprehensive threat enumeration tool that directly addresses collusion by accounting for all possible sets of attacker and target parties. ABC helps in reducing the combinatorial growth of these cases by ruling out irrelevant ones and merging threat cases that have the same effect. Such explicit consideration of attacker collusion is particularly important in permissionless cryptocurrencies that allow anyone to join.

ABC's models are also tailored to better consider the threat domain of cryptocurrencies. This is done by introducing new threat categories that account for the financial motivations of attackers and the new asset types, i.e., critical system components, these systems introduce. ABC identifies these categories by listing the assets in a system, such as the blockchain and the peer-to-peer network, and outlining the secure behavior of each asset. Then, the threat categories are defined as any violation of the security requirements for these assets. This approach produces a series of system-specific threat classes as opposed to an a priori-fixed list of generalized ones.

Another feature of ABC is acknowledging that financial incentives and economic analysis can play major roles in other steps in the design process. These tools can be used in risk assessment and in mitigating some types of attacks that cannot be neutralized cryptographically.

To demonstrate the framework's effectiveness, we conducted a user study and prepared use cases. The study compared the performance of subjects in building threat models using ABC and the popular framework STRIDE. Among the obtained results, we found that around 71% of those who applied ABC were able to identify financial threats in a cryptocurrency system, as compared to less than 13% of those applying STRIDE. In addition, while none of the STRIDE session participants spotted collusion between attackers, around 46% of those who used ABC identified these

scenarios. For the use cases, we applied ABC to three real-world systems, including Bitcoin, Filecoin, and CacheCash. These cases attest to the usefulness of ABC's tools where they integrated well into CacheCash's design phase, and they revealed several missing threat scenarios in the public design of Filecoin. These findings confirm the potential of ABC as an effective tool for assessing and improving the security of cryptocurrency-based systems.

2.2 Related Work

In this section, we summarize some prior work done around threat modeling. We also highlight relevant works on threat identification and security analysis in cryptocurrencies.

Threat modeling frameworks. The STRIDE framework, developed by Microsoft as part of its Security Development Lifecycle (SDL), is one of the earliest and most popular works in this field [88, 123]. STRIDE is an acronym of the threat categories the framework covers, namely, Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. This framework is a multistep procedure that involves understanding the software application functionality, capturing its operation flow using data flow diagrams (DFDs), mapping the components of these DFDs to the threat categories mentioned previously, and employing threat tree patterns to derive concrete threat cases.

Though several solutions have extended STRIDE to accommodate more complex systems [105], and cover other security requirements, e.g., privacy [73], its model does not fit cryptocurrencies. Another study [125] has borne out this premise, where the authors extended STRIDE's threat categories to handle Bitcoin-like community currencies. However, their approach targets only community fund operation, where more modifications would be needed to handle other components of the currency exchange medium, and other types of distributed services.

Other paradigms have pursued slightly different approaches. KAOS [124] is a

goal-oriented requirements engineering framework that has been extended to cover security. ANOA [63] is a generic framework to define and analyze anonymity of communication networks. And the framework presented in [87] targets the secure design of data routing protocols. Finally, other works build specialized threat models for specific classes of distributed systems, e.g., storage systems [84] and virtual directory services [71], rather than introducing a framework. These works indicate that different types of systems have different requirements when performing threat modeling. This reinforces the idea that emerging systems, such as cryptocurrencies, need specialized threat modeling tools.

Security analysis of cryptocurrencies. Most of the work done so far in this category can be divided into two classes. The first formalizes the security properties of consensus protocols and blockchains [110], while the second discusses specific security attacks on cryptocurrencies. For example, in a series of studies on Bitcoin, Bonneau et al. [67] presents several security threats, Karame et al. [91] analyzes double spending in fast payments, Androulaki et al. [58] evaluates its anonymity property, and Gervais et al. [78] studies how tampering with the delivery of blocks and transactions affects the participants view of the blockchain. Work on other cryptocurrencies include Luu et al. [96, 97], who focus on security threats to smart contracts in Ethereum [131], while Sanchez et al. [103, 104] analyze the security of Ripple [40] and the linkability of wallets and transactions in its network.

While these attack descriptions are very useful, they only outline specific threat scenarios for a given system. Our goal, however, is to develop a framework that allows reasoning about the full set of potential attacks facing any cryptocurrency-based system.

2.3 Stepping through the ABC Framework

Having highlighted the need for a cryptocurrency-specific threat modeling framework, we now present the ABC model. As a systematized approach, applying ABC starts

by understanding the functionality of the system under design with a focus on its asset types and the financial motivations of the participants (Section 2.3.1). This is followed by identifying the impactful threat categories and mapping them to the system assets (Section 2.3.2). After that, ABC directs system designers to extract concrete attack scenarios using a new tool called a collusion matrix, which helps in exploring and analyzing the full threat space (Section 2.3.3). Lastly, ABC acknowledges that financial incentives affect other design steps including risk assessment and threat mitigation (Section 2.3.4).

To make the discussion easier to follow, we illustrate the ABC process by describing its application to the following simplified system, which was inspired by Golem [27]:

CompuCoin is a cryptocurrency that provides a distributed computation outsourcing service. Parties with excessive CPU power may join the system as servers to perform computations on demand for others. Clients submit computation jobs to servers, wait for the results and proofs of correctness, and then pay these servers cryptocurrency tokens. The mining process in CompuCoin is tied to the amount of service provided to the system. That is, the probability of selecting a server to mine the next block on the blockchain is proportional to the amount of computation it has performed during a specific period.

The full threat model of CompuCoin is available online [2]. Several excerpts from this model are embedded in the discussion of ABC steps that follows.

2.3.1 System Model Characterization

Understanding the system is an essential step in the threat modeling process. A misleading or incomplete system description can lead a designer to overlook serious threats and/or incorporate irrelevant ones. Therefore, an accurate system model must outline the use scenarios of the system, the assumptions on which it relies, and any dependencies on external services. In addition, this model must be aware of all participant roles, and any possible motivation they might have to attack the system. For the latter, evaluators need to consider how the financial interests of these

Functionality description. Outlined in CompuCoin description introduced earlier.

Participants. Clients and servers.

Dependencies. May rely on a verifiable computation outsourcing protocol.

Assets. Computation service, service rewards (or payments), blockchain, currency, transactions, and the communication network.

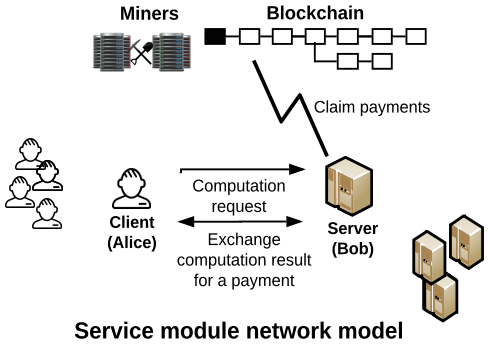


Figure 2.1: System model characterization of CompuCoin.

participants shape their behaviors.

Moreover, a system model must define the critical components that need to be protected from attackers. These components represent the assets that would compromise the whole system if compromised. To capture the features of the system, ABC identifies these assets based on its functionality. In detail, ABC divides the system into modules, and labels the valuable components of each module as assets, which could be concrete or abstract resources [105]. For example, the blockchain and the currency can be considered concrete assets, while preserving user privacy would be an abstract asset.

Finally, a system model includes graphic illustrations of its work flow. For distributed systems, it is useful to draw network models [105], in which system modules are represented by graphs showing all participants, assets, and the interactions between them. These graphs are helpful when enumerating the concrete threat scenarios as we will see in Section 2.3.3.

Running example application. Figure 2.1 illustrates how this step would look in CompuCoin. It shows the various components of the system model, in addition to a network model of the computation outsourcing service. It should be noted that the asset list in this figure is not exhaustive, and is limited by the rather brief description provided for CompuCoin.

As shown in the figure, CompuCoin is open to all participants to join as clients and servers, with servers also filling the role of miners. Dependency on other systems may include reliance on a verifiable outsourcing computation protocol, e.g. [109]. In terms

of assets, one may define three in CompuCoin: the *Service* promised to clients, the *Payments* used to compensate for the service, and the *Currency Exchange Medium* that covers four sub-assets (in light of the extended review of Bitcoin [67]): the blockchain, currency, transactions, and the communication network that connects the parties together. Here one may merge the currency with the transactions in one asset, as transactions are usually the currency tokens that state the coin's ownership. Another option is to merge currency with the payment asset to cover all currency flow in the system. However, we believe that a fine-grained division provides a more comprehensive treatment when identifying threats.

2.3.2 Threat Category Identification

After understanding the system model, the next step is to identify the broad threat categories that must be investigated. For each system component, system analysts outline all threat classes that may apply. Here ABC steps away from conventional practice of using an a priori-fixed list, and instead uses an adaptive approach inspired from requirements engineering [73] that defines threats as violations of system security goals. Given that assets are the target of security breaches, ABC defines these threat classes as violations of asset security requirements. This allows deriving cryptocurrency-specific threat categories because ABC identifies the assets in a way that aligns with the functionality of these systems.

Accordingly, in this step, an evaluator examines each asset in the system and applies the following procedure to identify its threat classes:

- Define what constitutes secure behavior for the asset, and use that knowledge to derive its security requirements. These requirements include all conditions that, if met, would render the asset secure. For example, CompuCoin's servers provide a computation outsourcing service and collect payments in return. One may consider the service payment asset secure if: a) servers are rewarded properly for their work, and b) that they earned the payments they collected.
- Define the threat categories of an asset as violations of its security requirements. Tying this to the above example, the service payment asset would have

Table 2.1: CompuCoin threat categories.

Asset	**Security Threat Category**
Service	Service corruption (provide corrupted service for clients).
	Denial of service (make the service unavailable to legitimate users).
	Information disclosure (service content/related data are public).
	Repudiation (the server can deny a service it delivered).
Service payments	Service slacking (a server collects payments without performing all the promised work).
	Service theft (a client obtains correct service for a lower payment than the agreed upon amount).
Blockchain	Inconsistency (honest miners hold copies of the blockchain that may differ beyond the unconfirmed blocks).
	Invalid blocks adoption (the blockchain contains invalid blocks that does not follow the system specifications).
	Biased mining (a miner pretends to expend the needed resources for mining to be elected to extend the blockchain).
Transactions	Repudiation (an attacker denies issuing transactions).
	Tampering (an attacker manipulates the transactions in the system).
	Deanonymization (an attacker exploits transaction linkability and violates users' anonymity).
Currency	Currency theft (an attacker steals currency from others in the system).
Network	Denial of service (interrupt the operation of the underlying network).

the following threat classes: service slacking, where a server collects payments without performing all the promised work, and service theft, where a client obtains service for a lower payment than the agreed upon amount.

The previous steps are highly dependent on how system analysts define the security properties of an asset, especially if there is no agreed-upon definition in the literature. For example, several works studied the security of the blockchain [67,110]. Yet, there is no unified security notion for the service asset because each type may have different requirements.

Running example application. Applying this step to CompuCoin produced the threat categories listed in Table 2.1 (the detailed process can be found in Appendix A). We found this table useful when building threat models for all the use cases found in Section 2.5, where we mapped the listed categories to the assets in each system. In this process, we found that some threat types were not applicable due to the absence of some assets. Notably, Bitcoin's only assets are the ones related to the

currency exchange medium. On the other hand, some systems required replicating some of these categories among all instances of an asset, e.g., in Filecoin all service asset threats were replicated for the two service types this system provides, namely, file storage and retrieval. This shows how the system characteristics affect the threat category identification step in ABC.

2.3.3 Threat Scenario Enumeration and Reduction

Once the threat categories have been identified, the next step is to enumerate concrete attack scenarios under each threat type. It is important in this step to be as comprehensive as possible by considering all potential attackers, target parties, and the set of actions attackers may follow to achieve their goals. This also involves considering collusion cases where several attackers may cooperate on attacking the system.

Detecting collusion is particularly important in cryptocurrencies. This is because the presence of monetary incentives may motivate attackers to collude in more ways than traditional distributed systems. The popular centralization problem caused by mining pools attests to this fact, where these pools can collude and perform devastating attacks. Even miners may collude by accepting, or rejecting, updates on the network protocol which leads to hard forks in the system. ABC enables system designers to detect these, and other, collusion cases at early stages of the system design.

To achieve this, ABC introduces collusion matrices that instruct analysts to enumerate all potential collusion cases, and reason about the feasibility of all threat scenarios in the system. A collusion matrix is two-dimensional, with the rows representing potential attackers and the columns representing the target parties. For the rows we list all participant roles in the system, both individually and in every possible combination. We also add a category called "external" that represents all entities outside the system. The same is done for the columns, with the exception that "external" is excluded. By definition, an external party is not part of the system, and hence, can not be a target. Each cell in these matrices represents a potential threat case to be investigated.

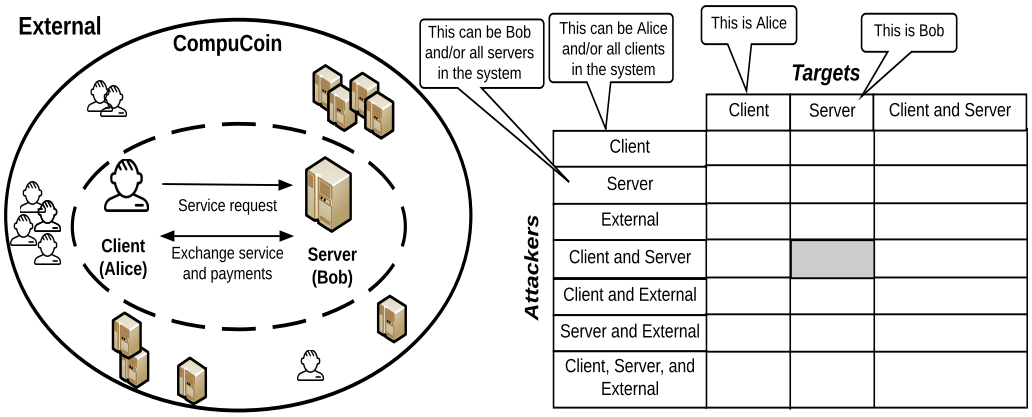


Figure 2.2: Collusion matrix of service theft threat in CompuCoin.

An example of a collusion matrix for the service theft threat in CompuCoin is shown in Figure 2.2. The dashed ellipse in the accompanying network model encloses a service session, which is an interaction between a server and a client. Any entry with multiple parties on the attacker side in this matrix indicates collusion. Note that a participant label may represent slightly different roles depending on where it is placed. For example, in Figure 2.2 the label "server" on the target side corresponds to a single server (i.e., Bob) since a service session involves only one server. However, the label "server" on the attacker side represents all servers in the system, including Bob. Hence, the cell in grey shade in Figure 2.2 does not suggest that a server colludes to attack itself, but instead, it represents the case where other servers collude with Alice against Bob.

For each threat category mapped to the assets in the system, a separate collusion matrix is created and analyzed as follows:

1) Enumeration: In this step, system analysts examine each cell and enumerate all strategies that attackers can use against the target parties, while documenting the process. It is useful to consider the network model of the system components as they show the interactions between the participants and the system assets.

2) Reduction: While examining each cell, system analysts reduce the number of threat cases by:

- Eliminating cells representing scenarios that will not produce a threat to the

system and recording the rationale for elimination. For example, in Figure 2.2, the cells that have the client as a target are irrelevant to the service theft threat. This is because a client does not provide a service to others. Other cases can also be crossed out if they are neutralized by system assumptions or by early design choices. For example, requiring all transactions to be signed by their originators rules out transaction repudiation and tampering attacks.

- Merging together scenarios (and the corresponding cells) that have the same effect, or those that do not become stronger with collusion. For example, in Figure 2.2, the grey shaded cell in which Alice is colluding with other servers to avoid paying Bob is reduced to the case that Alice is a sole attacker. This is because only Alice pays for the service she receives from Bob, whereas other servers are not part of the protocol[1].

3) Documentation: System analysts should document all threat scenarios that remain after the reduction step. That is, each documented case should outline the attack description, the target asset(s), the attacker(s), the flow of actions, all preconditions that make the attack feasible, and the reasons behind merges and deletions (if any).

The overall number of matrices and the size of each matrix depend on the system parameters, such as number of participant roles and assets. The above reduction steps eliminates a substantial number of cells in a principled way, which saves time and effort.

Running example application. The CompuCoin threat model has 11 collusion matrices [2]. We present one of them here; the service theft threat collusion matrix as illustrated in Figure 2.3. As shown, 21 cells can be reduced to just 2 threat scenarios (merged and ruled-out cells are displayed in pink and black shades, respectively). In this matrix, ten cases have been ruled out. This includes all cells under the column with the "client" header, for the reasons explained previously, and the first three

[1]The case that these clients drop/withhold these payments in collusion with Alice is part of other threats, such as DoS attack.

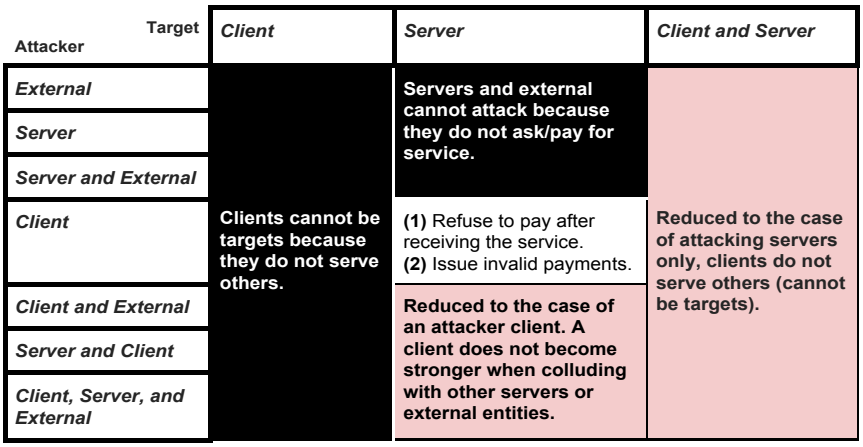


Figure 2.3: Analyzing service theft matrix in CompuCoin.

cells under the column with the "server" header. This is because "external" and/or "server" cannot be attackers because they do not ask/pay for the service[2].

Ten other merged cases are shown in Figure 2.3. This includes all cells under the column with the "client and server" header, which are reduced to attacking only servers. This is again because clients do not serve others. The rest of the merges cover the last three cells in the column with the "Server" header. In these cells, a client is colluding with an external entity and/or other servers to make the target server lose payments. Such collusion will not make a client stronger (these parties can drop/withhold payments, however, this is covered under DoS threat). Hence, all these cells are reduced to the case of a solo client attacker.

2.3.4 Risk Assessment and Threat Mitigation

The outcome of the threat modeling process, i.e., the documented list of impactful threat cases, can provide the designers with a guiding map to secure the system. During this process, it is useful to prioritize threats based on the amount of damage they can cause. This falls under the purview of risk management, a separate task from threat modeling, carried out using frameworks like DREAD [88] or OCTAVE [52].

ABC integrates with risk management by leveraging existing techniques for threat

[2]One may say that an external may join the system as a client to perform the attack. This case is covered under the client role in the matrix.

mitigation. For example, many threat vectors can be addressed using rational financial incentives that are often called detect-and-punish mechanisms. That is, when a cheating incident is detected, the miners punish the attacker financially. These approaches can use a game theoretic approach [122] to set the design parameters in a way that makes cheating unprofitable compared to acting honestly. By modeling interactions between the players as an economic game, the financial gain of all player strategies can be computed. Then, the parameters are configured to make honest behaviors more profitable than cheating. The same procedure can be used to quantify the damage these financial threats may cause. In other words, a threat that could give the attacker a big payoff should be prioritized over a threat that yields minimal profits. This reinforces the idea that cryptocurrencies require an expanded model for exploring risks and countering them.

Running example application. To illustrate this step in CompuCoin, we consider the distilled threat scenarios found in Figure 2.3. Both threats can be neutralized financially by designing proper techniques to make the client lock the payments in an escrow, along with a penalty deposit. The client loses the latter if he should cheat, perhaps by issuing invalid payments that carry his signature. The penalty deposit amount can be computed as the maximum payoff a client may obtain by cheating. This makes cheating less profitable than honesty, and hence, unappealing to rational clients.

2.4 Evaluation

To evaluate the effectiveness of ABC, we set up an empirical experiment that compares how it performs against STRIDE [88], a widely used threat modeling framework. We chose STRIDE for this comparison because it is a popular example of the type of a model a system designer will turn to in the absence of a cryptocurrency-specific framework [125]. The experiment took the form of a user study in which participants were asked to build threat models for a simple cryptocurrency system using one of these two frameworks. As our primary goal was to test whether financial incentives

and collusion could influence the type of threats discovered, our evaluation focuses on answering the following questions:

1. Does a threat modeling framework affect how subjects characterize a system model?
2. Do the threat categories of each framework influence the broad threat classes identified by the subjects?
3. Do participants build more accurate threat models when using ABC than when using STRIDE?
4. Do participants find the ABC/STRIDE method easy to use in completing the study?

In what follows, we discuss the study methodology and some of the insights drawn from the findings.

2.4.1 Methodology

We recruited 53 participants, primarily masters students in systems security programs. We used five subjects as a pilot group to test and refine our materials. The remaining 48 participants were divided randomly into two groups of 24, one of which built the threat model with STRIDE, whereas the other used ABC.

Each testing session spanned three hours and was divided into two parts: a group tutorial and individual completion of threat models. The group tutorial started with a 20 minute overview of cryptocurrencies, followed by a one-hour training in the framework to apply. The ABC tutorial contained a summary of the steps found in this paper, and for STRIDE, we prepared a tutorial based on material found in [32, 73, 88, 123]. The participants were then given a 25 minute break to reduce any fatigue effects. The session resumed with a 15 minute overview of ArchiveCoin, the system for which subjects will build a threat model. ArchiveCoin is a simplified Filecoin [49]-inspired cryptocurrency system that focuses mainly on the service and its rewards in order to fit the study session period.

The individual completion of threat models spanned the remaining hour of the study session. Given that the allocated time was short, we asked the subjects to look into just one threat category in Step 3, namely, the service theft of file retrieval. This category was not used in the clarifying examples of the tutorials to avoid biasing the results. Participants performed Steps 1 and 2 (system model characterization and threat category identification), and then submitted their answers. Only at this point were they given the materials for Step 3, in which they were asked to elicit threat scenarios for service theft of file retrieval. This was done so that participants who missed this threat when answering Step 2 could not alter their responses. At the end, the participants were asked to fill out a short questionnaire in which they rated how easy or hard it was to apply the threat modeling framework they employed. Our study instrument and all supporting materials are available online [2].

2.4.2 Findings

In this section, we discuss the main results produced by the study.

Effect on the System Model Characterization Step

In the first step of each threat modeling framework, the subjects were asked to characterize the system model by defining its modules, its assets, and the participant roles, in addition to drawing either a network model of the system, in case of ABC, or a data flow diagram (DFD), in case of STRIDE. To quantify the influence of the framework on this step, we compute the subject scores based on reference threat models we built for ArchiveCoin[3]. We report these scores after normalization, meaning that we divide them by the maximum score value one may obtain when answering everything correctly.

The results for Step 1 are found in Figure 2.4[4]. As shown, ABC scored higher

[3]We built two reference models, one using STRIDE and one using ABC to evaluate the responses of each framework session. Nonetheless, both models produced the same list of elicited threat cases in the last step.

[4]This figure is a box plot [21], which displays the distribution of the data points by showing the maximum and minimum (the whiskers above and below the box), the median (horizontal line inside the box), and the data points that span the first to third quartiles (the box itself). In case most of

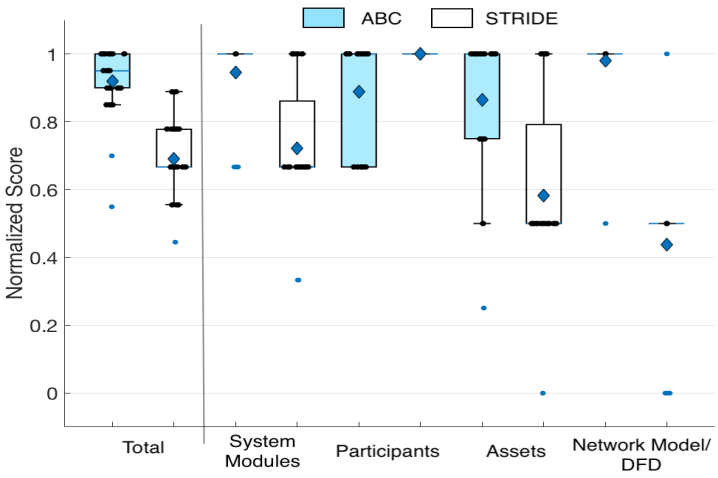


Figure 2.4: Subject scores for Step 1. Diamonds indicate the mean.

than STRIDE, with total average values of 0.92 and 0.69, respectively. Analyzing the responses for the sub-steps in Step 1 revealed several interesting observations. The first one is related to identifying the financial assets and modules in the system. As the figure shows, several subjects who applied STRIDE did not identify the payments (or currency) as an asset. Instead, their focus was on the user files stored in the system. Similarly, most of them did not identify the payment process as a system module, and focused only on file storage and retrieval processes. On the other hand, most of the subjects in the ABC session identified these financial related assets and modules, as reflected in the lower average scores of STRIDE participants reported in Figure 2.4, and the frequency results found in Figure 2.5. These results indicate that employing conventional threat modeling frameworks, instead of ones that are customized for monetary-incentivized systems, could lead evaluators to neglect the financial aspects of the system. This, in turn, could cause evaluators to miss important threat cases, and thus, leave the system vulnerable to attacks.

The second observation is related to how subjects defined the participant roles in the system. As shown in Figure 2.4, STRIDE achieved an average score of 1 in this category as compared to only 0.89 for ABC. All STRIDE session subjects defined the participant roles correctly, which in that model, included only clients

these points are very close this box is suppressed into a line.

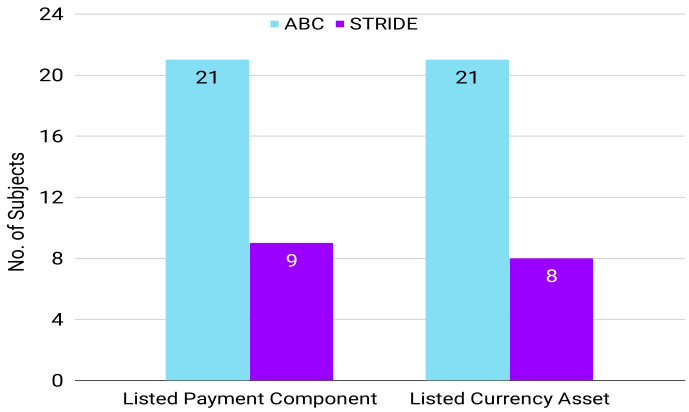


Figure 2.5: Identifying payment related modules and assets.

and servers. For ABC, we mentioned in its tutorial that an "external" entity must be considered among the participant roles. Yet, not all the subjects in the ABC session recorded the "external" role in their responses. This points to an important observation. Evaluators may only consider insider attackers because they interact with the system, and forget that external entities could be also motivated to, and capable of, an attack. Considering external attackers affects not only the eliciting of concrete threat scenarios in Step 3, but also the identification of the threat categories in Step 2. Therefore, more emphasis need to be placed on this role early on in the threat modeling process.

Lastly, the third observation is related to the framework influence on how subjects represented the system modules graphically. Figure 2.4 shows that the average scores for the network model/DFD sub-step were found to be 0.98 and 0.44 for ABC and STRIDE, respectively. STRIDE session subjects struggled to draw a DFD for ArchiveCoin because such a representation is more suitable for software applications than distributed systems. On the other hand, ABC's use of network models made this task easier for its session subjects, and almost all of them sketched diagrams correctly. As mentioned previously, this graphic representation helps in eliciting the concrete threat scenarios in the system (Step 3), and hence, inaccurate diagrams may affect the outcome of this process.

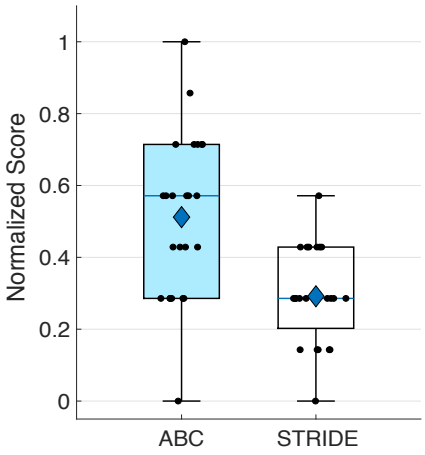


Figure 2.6: ABC and STRIDE scores for Step 2. Diamonds indicate the mean.

Effect on the Threat Identification Step

In Step 2 of both frameworks, the subjects were asked to define the broad threat categories to be investigated. As part of the study material, participants who applied STRIDE were given its threat category list, along with the component mapping table found in the STRIDE user guide [32]. Similarly, participants who applied ABC were given the list found in Table 2.1 (covering only the service and service reward assets). The subjects in both groups defined the categories to be considered for ArchiveCoin by mapping these lists either to the system assets (in case of ABC), or to the DFD components (in case of STRIDE). The reference models we built indicated that the mapping outcome for both frameworks would include the following threat classes: service corruption, DoS, information disclosure, service slacking and service theft for both service types that ArchiveCoin provides (file storage and retrieval).

Based on the scores for the threat identification step found in Figure 2.6, the cryptocurrency-tailored categories of ABC made it easier for the study participants to identify the threat categories in question as compared to STRIDE. This is despite the subjects having little experience with cryptocurrency-based systems. The average score for ABC subjects is around 0.51, compared to 0.29 for STRIDE (note these scores are normalized as mentioned before). The generalized categories used by STRIDE fit software applications well, but they do not suit monetary-incentivized distributed systems. System analysts, using these generalized categories, would need

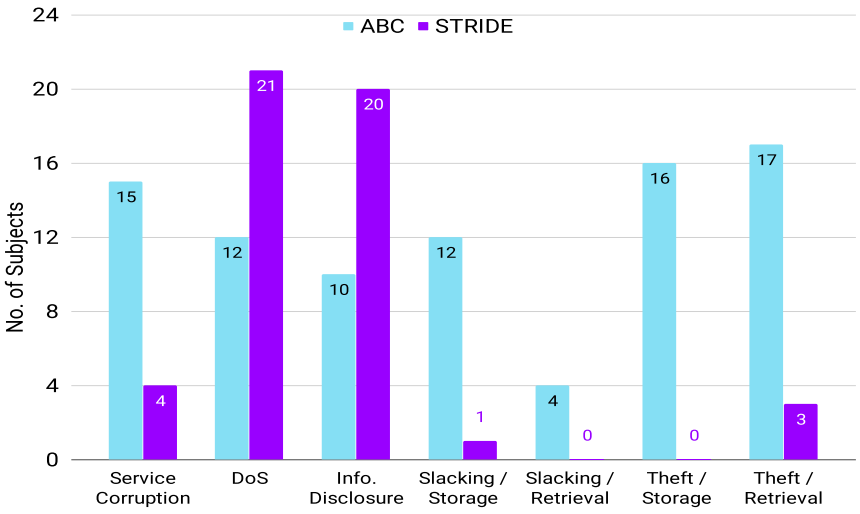


Figure 2.7: Frequency of threat category identification.

to expend more time and effort in order to identify the more specific threat classes of interest.

To provide more insights about the effectiveness of the threat categories of each framework, we analyzed the number of subjects who identified each threat category as depicted in Figure 2.7. ABC is ahead of STRIDE for all financial-related threats, i.e., service slacking and theft, as well as the service corruption threat. For the service theft of file retrieval threat, which is the category that we asked the participants to investigate in Step 3, only three participants in the STRIDE session spotted this threat, while 17 subjects in the ABC session did so, or around 13% and 71%, respectively. For service theft of file storage and slacking of file retrieval, none of STRIDE participants spotted these threats, and only one participant spotted service slacking of file storage. In contrast, 67%, 17%, and 50% of ABC participants identified these threats, respectively. This, again, shows that the ABC categories guided the subjects toward service and payment related threats in a better way than the general threat classes included by STRIDE.

For the rest of the threat classes, we found that STRIDE's subjects are ahead of ABC's session participants for both DoS and information disclosure threats. As shown in Figure 2.7, around 88% and 83% of STRIDE subjects identified these categories, respectively, while around 50% and 42% of ABC's subjects did so. Although we do

not have a precise justification for this outcome, we think that this is due to the fact that these threats are thoroughly explained in the threat category table of STRIDE, which includes attack examples as well. Hence, we believe that the ABC tutorial needs to stress these threats and explain them in greater depth.

Threat Model Accuracy

To quantify accuracy, we compute the recall and precision values for the concrete threat scenarios found by each subject as compared to the reference threat models we built for ArchiveCoin. The recall is computed as $TP/(TP + FN)$, and precision is computed as $TP/(TP+FP)$, where a true positive TP is a correctly identified threat, false negative FN is an undetected threat, and false positive FP is an incorrectly defined threat. The recall (precision) indicates how many valid (invalid) threats a subject defined. Both quantities take values between 0 and 1.

Based on the results of Step 3 in each framework (i.e., eliciting concrete threat scenarios), we found that participants who applied ABC produced a larger number of valid threat cases than STRIDE session subjects, with average recall values of 0.48 and 0.4, respectively. At the same time, participants using ABC identified a lower number of irrelevant cases than those who applied STRIDE. The former scored an average precision value of 0.57, as opposed to 0.48 for the latter[5]. We believe that this is due to several factors. First, ABC directed participants to consider the financial aspects of the system, which affected the elicited threat scenarios. Second, the use of the collusion matrices helped ABC participants to reason about the threat space in an organized way that reduced random speculations, as opposed to the STRIDE threat tree patterns that work well when applied to software applications. Third, ABC's collusion matrices guided participants to spot threat cases caused by collusion, as opposed to STRIDE's tree patterns that focus only on solo attackers. The results show that none of the subjects in the STRIDE session identified a possible collusion case between a client and servers, while 11 subjects in the ABC session identified

[5]One participant in ABC session has 0 false negative and 0 true positive, we excluded him/her when we computed the average.

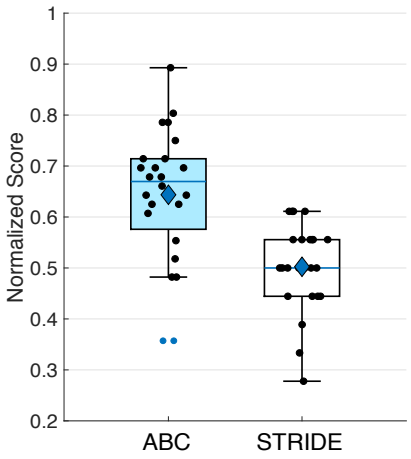


Figure 2.8: Total normalized scores (diamonds indicate the mean).

this collusion case[6]. This confirms the importance of considering collusion when investigating threat cases, and shows the usefulness of ABC matrices in handling this task.

All these factors affected the overall correctness of the threat models built by the subjects. As shown in Figure 2.8, the ABC session scored an average of 64% as compared to 50% for STRIDE. This is expected based on the reported results for the modeling steps, where ABC scores were ahead of those of STRIDE.

Framework Ease-of-use

As mentioned previously, at the end of each session, we asked participants to report on how easy they found applying the framework in question. Ease-of-use was measured on a Likert scale in which 1 indicates the lowest value and 5 indicates the highest value. The average values are 3.9 for ABC and 3.8 for STRIDE[7]. This result becomes somewhat more significant when we point out that the participants already had some exposure to STRIDE and its threat tree patterns. Even though it is a new framework that introduced several new concepts to the participants, ABC still achieved a comparable ease of use level. This suggests that participants were able to

[6]Most of them, however, did not provide a clear description of the attack scenario. Hence, these incomplete descriptions *were not* counted as correct threats when grading Step 3.

[7]Three participants did not complete the questionnaire in the STRIDE session.

grasp its concepts through just a single hour of training, and therefore ABC shows potential as a usable method for threat modeling.

2.4.3 Threats to Validity

We acknowledge that a few limitations must be kept in mind when considering the study results. Empirical studies of threat modeling usually span a longer time frame, often on the orders of months, e.g., [118]. However, the fact that we were able to glean several important observations suggest that there may be lessons to be learned from short focused studies. In addition, we feel the design of our study might serve as a guide for determining promising areas for extended research before a large commitment of time and resources is made. Another constraining factor could be the age and experience level of our subjects. Different responses might have been obtained if we tested system security experts. However, we believe that the inexperience of our participants match the cryptocurrency space well, which attracted users and researchers from various fields, even those from outside systems security. Therefore, our results give indications on how they might perform when investigating the security of cryptocurrencies.

2.5 Use Cases

To demonstrate how ABC would function when applied to complex real-world systems, we developed use cases in which we built threat models for three cryptocurrencies. Each of these cryptocurrencies represents a different stage in a system design lifetime. Bitcoin [107] is a well-established system, Filecoin [49] is under development and close to being launched, and our system, CacheCash, is still in its early development stages. The analyses for Bitcoin and Filecoin stopped before the risk management/mitigation phase. However, CacheCash's analysis involves also the risk mitigation step as we later describe.

Table 2.2: Threat model comparison.

Aspect	**Bitcoin**	**Filecoin**	**CacheCash**
ABC steps covered	Steps 1 - 3	Steps 1 - 3	Steps 1 - 4
Completion time (hr)	10	47	Not tracked
No. of collusion matrices	5	14	9
Total threat cases	105	882	525
Distilled threat scenarios	10	35	22

2.5.1 Bitcoin Analysis (Steps 1-3)

Bitcoin is by far the most valuable cryptocurrency with a capital market share of around $92 billion as of April 2019 [24]. As shown in Table 2.2, the Bitcoin threat model has significantly fewer collusion matrices and threat cases than other systems[8]. This is because it provides only a currency exchange service, which reduces the number of assets. Furthermore, it involves only two types of participants, miners and clients, which reduces the size of the collusion matrices. These factors, in addition to our familiarity with Bitcoin design details, contributed in reducing the completion time of Bitcoin's threat model as shown in the table. Moreover, at the time we were working on this model, we had already completed the design of ABC. This suggests that deep understanding of the system model, and the availability of suitable tools impact not only the accuracy of the results, but also the time and effort expended in the threat modeling process.

We drew two main observations about the threat model we generated for Bitcoin. First, all the known threats to Bitcoin, such as double spending, Eclipse attacks [85], Goldfinger attack [94], and delaying blocks and transaction delivery [78], were mapped to the collusion matrices produced by ABC. Second, collusion between participants can play a major role in Bitcoin's security. That is, several threats are neutralized by the assumption that at least 50% of the mining power is honest. Yet, mining pools have been formed to concentrate mining power. At the time of this writing, around 95% of the mining power is in the hands of just 10 mining pools [9]. If the managers

[8]The full threat model of Bitcoin is available online [2]

of these pools decide to collude, they could break the honest majority barrier and take the system down. In fact, serious security attacks can be performed with less amount of mining power. Sompolinsky et al. [121] attest that a selfish mining attack, or blocks withholding, in which an attacker controls even less than 30% of the mining power, would be able to undermine the fairness of the mining reward distribution.

Furthermore, miner collusion may take a different form represented by rejecting specific updates on the network protocol. As cryptocurrencies are still in the development stage, features and updates are still being added to their protocols. These updates can create soft and hard forks in the network [61]. In this form of collusion, a subset of the miners do not agree to adopt the new version of the protocol. This causes a split in the network, i.e., inconsistency in the blockchain view, and may extend to spinning out a new version of the currency. This happened in Bitcoin, where there two new cryptocurrencies split from its network including Bitcoin Cash [11] and Bitcoin Gold [12].

Usually, the >50% threat, or miners' collusion in general, is argued about informally using incentive compatibility. This idea asserts that rational miners are more interested in keeping the system running to preserve the value of their rewards. However, this claim is hard to verify and remains as an open question [67]. In addition, this assumption might be valid when all parties belong to the same system. Yet, miners could be working in several systems and it could be the case that destroying one to strengthen the other would be more profitable. Nonetheless, such observations highlight two key points. First, it indicates the importance of validating all the security assumptions a system makes in its design. And second, it points to the need for rational economic incentives to address some types of security threats that cannot be addressed by using only cryptographic approaches. The design of ABC accounts for such observations as mentioned earlier in this chapter.

2.5.2 Filecoin Analysis (Steps 1-3)

Filecoin [49] is a cryptocurrency-based distributed file storage and retrieval system. Any party may join the system as a storage or retrieval miner to offer service to

others. Filecoin operates distributed retrieval and storage markets where clients and miners can submit storage/retrieval bids and offers. Once these offers are matched, the service-payment exchange process, in which clients pay the miners in the Filecoin currency in exchange for receiving correct service, may start. The mining process in Filecoin is tied to the storage service miners put into the system. Recently, the Filecoin team raised around $250 million through an ICO (initial coin offering) [1] in preparation for an official launch.

Filecoin is a more complicated system than Bitcoin as it provides two types of services on top of the currency exchange medium. In addition, its protocol involves three participant roles: clients, retrieval miners, and storage miners, with the latter filling the traditional roles of miners in maintaining the blockchain. This complexity is reflected in the number of collusion matrices and threat cases produced as shown in Table 2.2. Moreover, all threat categories that target the service asset were replicated for each service type, which contributed to the large size of the threat model. These factors affected the completion time to build the model, which was 4.7x the time needed to build Bitcoin's model. This cost in time commitment is a natural result of working with newly developed and complex systems that provide a rich set of features.

In threat modeling Filecoin's whitepaper, we found three unaddressed issues, mostly dealing with collusion cases that were not considered. Additionally, there are many places where the system is underspecified and so it is not possible to reason about whether or not it meaningfully addresses a threat.

Ethics and disclosure. We reached out to the Filecoin team, which mentioned efforts they have undertaken to resolve these problems. We withhold details about these issues until later as part of the responsible disclosure process.

2.5.3 CacheCash Analysis (Steps 1-4)

We developed ABC during the early stages of our work on designing CacheCash. As the work progressed, we realized that most of the threat cases we encounter are

related to the financial aspects of the system, and the possible collusion between participants. Such aspects, as mentioned previously, are not explicitly addressed by traditional threat modeling frameworks. At that time we realized that none of these frameworks suited our needs, which lead to developing ABC.

Beyond threat modeling, we used ABC while designing threat mitigation techniques in the CacheCash system. During that time, we observed the importance of rational financial incentives in this process. This includes employing detect-and-punish mechanisms in which the penalty deposit of a party is revoked upon detecting that it is cheating, or designing algorithms that, when implemented in a malicious way, can cost the attacker more in resources than would working honestly. Furthermore, we realized the value of game theory and economic analysis in assessing the effectiveness of these economic threat mitigation approaches, and in quantifying the risk, or amount of damage, that financial attacks may cause. To date, we found ABC useful for CacheCash in both the pre-design threat modeling step, and the after-design security analysis of the system modules. More about the threat model of CacheCash can be found in Chapter 5.

2.6 Conclusion

In this chapter, we introduced ABC, a cryptocurrency-focused threat modeling framework. Its design is motivated by the observation that traditional threat modeling frameworks do not fit cryptocurrencies, thus leaving them vulnerable to unanticipated attacks. ABC introduces collusion matrices, a technique that allows designers to investigate hundreds of threat cases in a reasonable amount of time. Both the user study and the use cases confirm that our framework is effective in unraveling hidden threat cases. This shows the potential of ABC to improve the security of a wide array of distributed systems.

We found ABC useful when building threat models for MicroCash (Chapter 4) and CacheCash (Chapter 5), where it allowed identifying the threat cases that the designs of these systems must address in order to secure their operation. Among these

threats, we have cache accounting attacks. In the next chapter, we present CAPnet, our defense mechanism against this type of attacks, which is a basic building block in CacheCash's design.

Chapter 3

CAPnet: A Defense Against Cache Accounting Attacks on Content Distribution Networks

This chapter is based on joint work with Kevin Kelley, Allison Bishop, and Justin Cappos [57].

3.1 Overview

In this chapter, we introduce CAPnet, the first technique that lets untrusted caches, such as peers with unknown computational and latency characteristics, join a peer-assisted CDN while providing a bound on the effectiveness of cache accounting attacks. Our key innovation is a lightweight *cache accountability puzzle* that clients must solve before caches are given credit. The puzzle solution serves as a content retrieval confirmation assuring publishers that a minimum, pre-configured bandwidth cost has been expended by caches.

For each content request, the publisher generates a puzzle that a client must solve by processing the data chunks retrieved from caches (each of which is encrypted with a request-specific key). Solving this puzzle requires the client to sequentially touch small pieces of these chunks in an unpredictable order. Because of this unpredictability, the communication overhead of generating the solution without having the data colocated is significant. Combined with the use of a *completion mask*, a secret that is used to conceal an encrypted data chunk until it has been completely transferred, this processing pattern forces colluding parties to expend the required bandwidth amount, which can be configured to be similar to retrieving the content, and thus removing any motivation to cheat.

Equally important for its use in practical applications, CAPnet does not sacrifice efficiency for enhanced security. Its tools are built on computationally-light operations (symmetric encryption and hashing). CAPnet is also designed to be scalable; while a client needs to process a large portion of the retrieved content when solving a puzzle, a publisher needs only to process a small, server-configurable number of pieces to generate a challenge. Because of this asymmetry, our scheme can meet the deployment demands of large-scale content distribution applications.

To demonstrate that CAPnet is effective at mitigating cache accounting attacks, we configure the system parameters based on an analysis of the bandwidth cost incurred by malicious puzzle-solving strategies. Our analysis shows that the publisher can ensure that a malicious actor must expend a substantial amount of bandwidth, even given unrealistically strong assumptions about the malicious actors capabilities. To evaluate CAPnet's efficiency, we experimentally evaluate the computational overhead of our scheme under various configurations. The benchmark results show that a modest client machine can solve puzzles at a rate sufficient to confirm the retrieval of around 170 Mbps. Even a single core low-end publisher machine can generate enough puzzles to support a bitrate of around 4 Tbps.

3.2 Related Work

To orient readers to current state-of-the-art defenses for cache accounting attacks, this section reviews prior work done in this area. We also present information about a related topic — proofs of data storage — and discuss why this proof paradigm is not applicable to cache accounting attacks.

Cache accountability in peer-assisted CDNs. One technique used in peer-assisted CDNs is to rely on the peers themselves to report statistics about content delivery. For example, clients in Akamai Netsession [4] share reports about their upload and download activity, and this information is used to manage network resources. Even some systems that exchange service for monetary rewards, e.g., [106], rely on these types of reports to track the service contributions of peers in order to

pay them accordingly. However, in such an open environment that allows anyone to join, peers may fabricate these accounting reports. This has been confirmed through empirical studies [51, 95].

Specialized cache accountability defenses work to address this issue by making clients commit to these activity logs. This is done by requiring participants to maintain tamper-evident logs, cryptographically sign all messages sent to the network, and periodically exchange these logs with a verifier. The verifier in turn checks the consistency of the reported information and performs anomaly detection to identify cheating based on a protocol reference implementation. The repeat and compare scheme [100] utilizes this technique to address the problem of corrupted content distribution. PeerReview [83] employs a similar approach to detect Byzantine faults. And RCA (Reliable Client Accounting) [51] exploits such logs to address the same collusion problem we are interested in. However, this approach cannot prevent colluding parties from fabricating consistent and valid-looking, signed and consistent, log reporting content transfers that did not take place. Thus, cheating clients and caches still can collude to collect rewards for work they did not perform.

A prior bandwidth puzzle-based defense, proposed by Reiter et al. [113], works by issuing challenge puzzles to all caches and clients that possess the content. These parties have to solve the issued puzzles over the retrieved content in a short period of time to receive payment, where it assumes knowledge of the computational abilities and communication latency of all parties. This scheme has several downsides when compared to our approach. First, every time a new party retrieves the content, puzzles must be solved by *all* peers that have a copy of this content, even those uninvolved in the transfer. Second, the security of the scheme is based on knowing a bound for the attacker's hashing power, which is used to quantify the number of challenge puzzles that must be presented within a time window. Third, the latency of peers must also be known for the scheme to resist cheating. This latency constraint may cause peers to lose their rewards in the event of lost or delayed messages. In addition, an attacker that can fool others into believing they have high latency can cheat because she has more time to solve puzzles, and hence, she can collude with

other caches to solve their puzzles.

Relation with data possession proofs. Several works in the literature have tackled a related problem — how to prove that a server to which a client has outsourced files is actually storing those files, e.g., ensuring correct data storage in the cloud [127]. Solutions to this problem include proofs-of-retrievability [69], proofs of data possession [75], and proofs-of-storage [74]. Such proof systems, at first glance, could be viewed as potential defenses against cache accounting attacks. That is, a publisher can ask a client to prove storing a local copy of the retrieved content. However, this does not confirm that caches have served the content. These colluding caches can generate valid proofs of storage for any client because they have the full raw content. Similarly, a client that retrieves some content only once can produce valid proofs, for itself or others, for all future requests that ask for the same content. While useful, these proof systems are not applicable for fighting cache accounting attacks.

3.3 CAPnet Design

CAPnet defends against cache accounting attacks by both mandating proof of delivery, and making honest choices more profitable than cheating. In this section, we describe how the design of CAPnet manages this defense. We start by defining the work environment for content distribution systems our scheme targets, then provide a high level view of the primary operations, after which we present the technical details of these operations.

3.3.1 Work Environment Model

CAPnet targets the general paradigm of peer-assisted content distribution systems, which also represents the basic network model of CacheCash. Such a model consists of three participant types: publishers, caches, and clients. A *publisher* owns content, e.g., videos or software packages, that *clients* want to retrieve. A publisher hires *caches* to distribute this content in exchange for rewards, such as monetary incentives,

which are tied to the amount of service these caches provide. Each cache is defined by its IP address, which the publisher monitors to detect Sybils. When a cache joins a publisher's network, it gains access to the content to be served, which we assume to be divided into equally-sized data chunks. A client request can fetch a range of n chunks within the object, e.g., movie, it wants to retrieve.

During the content distribution process, a publisher acts as a dispatcher assigning caches to fulfill client requests. Therefore, clients must contact the publisher first, asking for n data chunks, to obtain the list of n caches that will provide the service. The publisher selects this set randomly such that each cache will serve a single data chunk among the requested set.

As will be shown shortly, confirming that the content has been retrieved is done over individual content requests. In other words, even if the client wants to retrieve a large object, e.g., a movie of size 1 GB, it computes the retrieval confirmation over each n chunks separately. Such an approach reduces the amount of memory that CAPnet requires, e.g., for $n = 4$ and a chunk size of 1 MB, a client/publisher would need only a 4 MB storage to hold the chunks needed for processing any request.

CAPnet enables the publisher to set a bound on the amount of bandwidth the attacker must expend, with respect to the original content amount, which we call the *δ-bound*. So, for 4MB of content, a 0.75-bound attacker in our scheme is expected to expend 3MB to provide a valid content retrieval confirmation for a content request. The δ-bound is controlled by the number of rounds in the cache accountability puzzle of CAPnet. The larger δ, the larger the computation cost of generating and solving this puzzle. Therefore, system designers need to configure this parameter based on the desired security-efficiency trade-off they want to achieve.

Lastly, we work in the random oracle model, where hash functions are modeled as random oracles. We also work in the ideal cipher model, where a block cipher is modeled as a random permutation. In addition, we deal with efficient adversaries that cannot break secure cryptographic primitives, such as AES, SHA256, and Pseudorandom Functions (PRFs), with non-negligible probability.

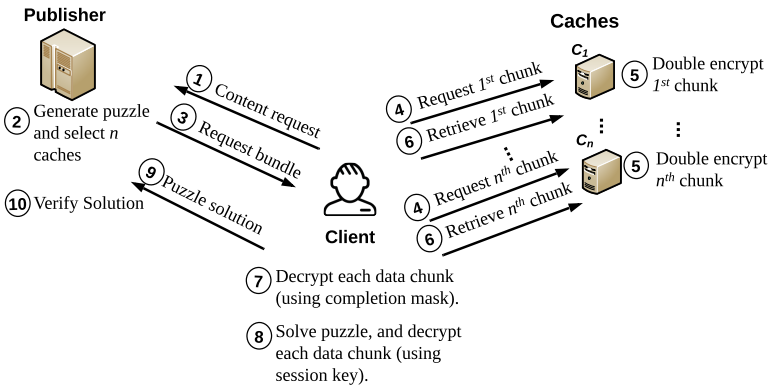


Figure 3.1: CAPnet integration in content distribution (n is the number of caches selected for a content request).

3.3.2 CAPnet in a Nutshell

CAPnet consists of a set of actions integrated into the content delivery process. Collectively, these actions demonstrate that the required bandwidth amount was actually expended, even in the face of malicious, colluding client and caches. In what follows, we provide an intuitive discussion of these actions to highlight the motivation behind the design (a rigorous security analysis of how these actions defend against cache accounting attacks is found in the next section).

As shown in Figure 3.1, to request content in the CAPnet model, a client retrieves a request bundle from the publisher that enables the retrieval of n data chunks (**steps 1, 2,** and **3**)[1]. This bundle stipulates which caches to contact, and includes the client's IP and a request number. In addition, the bundle contains a puzzle, that when solved, enables the client to prove that the requested chunks (or at least an amount of data within the δ-bound) were indeed retrieved.

The client contacts caches, possibly in parallel, and provides the request bundle that instructs each cache what specific data chunk to serve (**step 4**). Each cache will encrypt its data chunk with a unique per-request key, and additionally encrypts the produced ciphertext using a fresh per-request completion mask (**step 5**). The double encrypted chunk, appended with the completion mask, is then delivered to the client (**step 6**). Once all chunks are received, the client decrypts each chunk using

[1]If the content has more than n chunks a client will send several requests.

the completion mask (**step 7**), and solves the puzzle using the single-layer encrypted chunks (**step 8**). With the puzzle solution the client can decrypt the data chunks to obtain the raw content (**step 8**), and confirm to the publisher that these chunks were retrieved (**steps 9** and **10**).

When a cache begins serving content for a publisher, they establish a shared secret called a master key. Along with the request number, both parties use this key to non-interactively generate a fresh per-request session key that is used to encrypt the data chunk the cache will serve. Since this key is unique and cannot be distinguished from random[2], each request returns a different, random looking, encrypted chunk[3], even if the raw content is the same. Also, since the session key is a secret known only to the publisher and that cache, a malicious cache does not know the encrypted content that an honest cache would serve.

In addition, CAPnet ensures that a client retrieves the entire encrypted chunk before it can start solving the puzzle. This is done by having each cache select a random *completion mask*, i.e., a random key, that is used to encrypt the chunk ciphertext. A cache appends the completion mask to the transmitted, double encrypted chunk. Thus, the client has to download the entire chunk before being able to decrypt any part of it.

A puzzle is computed by processing small (e.g., 16 byte long) pieces of the (single-layer) encrypted chunks. Starting at a randomly selected piece in the first chunk, one computes the hash of this piece and maps it to a piece index in the second data chunk. In the random oracle model, this mapping randomly jumps to a piece in the second chunk. The hash is now computed over the previous hash and the second piece and is used to select another piece in the third chunk, and so on. Once the data chunk from each cache is visited, this completes a *round*. The next round is begun by mapping the last hash of the prior round to a piece index in the first chunk. This continues for the number of rounds chosen by the publisher to achieve the desired δ-bound.

The publisher and the client compute the puzzle in slightly different ways. The

[2]This is by the security of PRFs.

[3]Recall that we work in the ideal cipher model.

publisher randomly chooses a "starting piece" in the first chunk and computes one puzzle to produce a challenge for the client. This challenge does not contain any information about the starting piece. Hence, the client will attempt to solve the challenge by computing candidate, or trial, puzzles initiated at various pieces in the first chunk until the solution is found. This forces the client to process a δ portion of the content before finding the solution.

Increasing δ strengthens the security guarantees of CAPnet by causing malicious parties to retrieve more content, but also increases the computation cost for publishers and honest clients as larger number of rounds are needed. Caches, on the other hand, have uniform computational cost independent of the δ-bound.

3.3.3 Design Details

We now describe the CAPnet actions in more detail, including puzzle generation, solving, and verification.

Puzzle Generation

The publisher generates a challenge puzzle based on the data chunks a client wants to retrieve. Figure 3.2 depicts this action through a clarifying example involving two data chunks. In this figure, L stands for location, H stands for hashing, E stands for encryption, $||$ is a concatenation operation, and the arrows indicate the sequence of pieces selected when computing a puzzle.

As shown, a puzzle starts at the first data chunk and proceeds by processing a number of small data pieces selected at random. Given that a puzzle round processes encrypted pieces, the publisher encrypts the piece selected at each step. It then hashes the encrypted piece along with prior hash or location value. The output hash is mapped to a piece index within the next chunk (or the first chunk is this is the beginning of a new round).

This computation pattern imposes three aspects. First, each location value encapsulates all encrypted pieces processed so far, which enforces sequential computation of the puzzle. Second, processing encrypted pieces prevents any correlation between

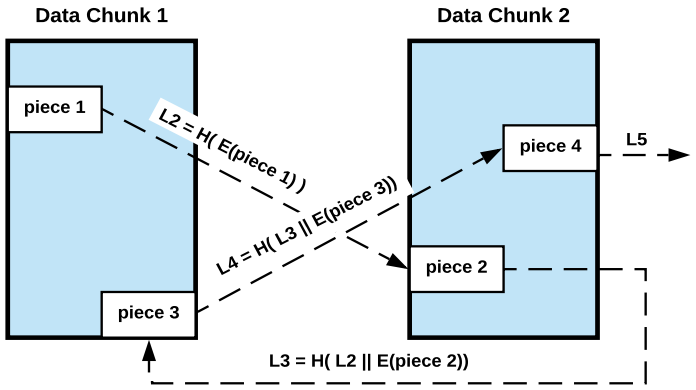


Figure 3.2: An example of puzzle challenge generation with two chunks and two rounds. The puzzle challenge is $H(L_5)$ and its solution is L_5.

puzzles generated for different requests, even if they are for the same raw content. Third, touching all data chunks in a round robin order makes all chunks contribute equally in solving a puzzle. This ensures that a puzzle solution is computed over all chunks, confirming their retrieval.

Once the puzzle computation is completed, which happens when the designated number of rounds is reached, the publisher uses the hash of the last location, i.e., $H(L_5)$ in the figure, as the challenge, and it asks the client to return the preimage of this hash, i.e., L_5. This is done without revealing $piece_1$. Instead, the client tries all data pieces in the first chunk. Hiding the starting piece causes the client to touch a large percentage of the pieces in each chunk while enabling the publisher to touch very few. This minimizes the computational load for publishers, allowing them to process a large number of client requests concurrently.

The technical details of this process are captured by Algorithm 1. In this algorithm, D_j is the j^{th} data chunk, $piece_i$ is the i^{th} data piece in a data chunk, R_{puzzle} is the number of puzzle rounds, and $pieces_{total}$ is the total number of pieces in any data chunk. The data pieces inside a chunk are referenced using their indices, where the first piece has an index 0, the second has an index 1, and so on. We use $index(\cdot)$ to denote the index of a given piece.

Algorithm 1 shows two phases: an initialization phase, and a challenge preparation phase. The initialization phase is needed to allow caches and the publisher to agree

Algorithm 1 Puzzle challenge generator.

Input: Data chunks $D_1, \ldots, D_n$
Output: Challenge

▷ Initialization
for $j = 1$ to n **do**
    Generate k_j and $ctr_{j,initial}$
end for

▷ Puzzle generation
Select $index(piece_1)$ randomly from D_1
Set $j = 1$, $L_1 = 0$, $r = n \cdot R_{puzzle}$
for $i = 1$ to r **do**
    Fetch $piece_i$ from D_j based on $index(piece_i)$

▷ Compute ctr_i and encrypt $piece_i$
    $ctr_i = index(piece_i) + ctr_{j,initial}$
    $c_i = \text{AES-CTR}(k_j, ctr_i, piece_i)$

▷ Compute location and index of $piece_{i+1}$
    $L_{i+1} = \text{SHA256}(L_i || c_i)$
    $index(piece_{i+1}) = L_{i+1} \mod pieces_{total}$
    $j = (j \mod n) + 1$
end for
Set Challenge = $\text{SHA256}(L_{r+1})$
return Challenge

on the encryption setup. We use AES in the counter mode (AES-CTR) for encryption because it allows a publisher to encrypt any individual data piece without encrypting the whole data chunk. To generate identical encrypted data, the publisher and each cache C_j must generate the same session key k_j and the initial value of the AES-CTR counter $ctr_{j,initial}$.

Generating the AES-CTR counter and session key is done using a one time setup without any per-request interaction between the publisher and cache. As mentioned before, a cache C_j shares a master key with the publisher that they both use to derive any future session key k_j. This is done by means of a pseudorandom function (PRF) keyed with this master key, and evaluated over the request number and the client IP to output k_j. The same PRF idea, but keyed with the session key, is used to generate $ctr_{j,initial}$. Accordingly, in the initialization phase, the publisher generates

all keys and counters for all data chunks that will be used by the caches involved in the service session.

The puzzle generation phase proceeds as described previously. After selecting a piece index at random from the first data chunk, the publisher proceeds by computing the location of the next piece (line 16), and mapping this location to a piece index (line 17). The new location computation requires encrypting the prior piece, which in turn requires computing the correct AES-CTR counter value (line 13). By doing so, the publisher produces the same ciphertext of the selected piece that a cache will produce. The aforementioned process is repeated for the required number of iterations, as found in lines 10-19. Lastly, the algorithm outputs the hash of the last location as the puzzle challenge.

After the puzzle is generated, the publisher informs the client about the puzzle challenge it has to solve as part of the request bundle mentioned earlier. To allow the client to decrypt the retrieved data chunks, the publisher can either provide the keys in response to the puzzle solution reported by the client, or simply encrypt all session keys using the puzzle solution and share the ciphertext as part of the request bundle. Either way, once the client solves the challenge puzzle it can recover these keys and decrypt the received data.

Puzzle Solving

The client receives the puzzle challenge, along with the cache contact information, within the request bundle sent by the publisher. With this bundle, the client can start the content retrieval process, where it connects with the listed caches and requests the specified data chunks. Caches will deliver double-layer encrypted chunks, with the completion mask appended to each chunk. The client uses the completion mask as the decryption key to remove the second encryption layer of the chunk. By repeating this process for all chunks, the client obtains the single-layer encrypted data chunks.

The client can now perform the second action in the CAPnet process — solving the challenge puzzle. It uses a similar algorithm to the one used by the publisher with three differences. First, since the client retrieves encrypted data chunks from the

caches, it does not encrypt the data pieces before applying the hash, thus skipping lines 4-6 and 13-14 in Algorithm 1. Second, since the client does not know the starting piece, it computes a puzzle for every piece in the first data chunk until the correct solution is found. In other words, it repeats lines 9-11 for each candidate starting piece. And third, once the client solves the puzzle, the output is the puzzle solution, which is the last location in the correct puzzle.

Puzzle Verification

The last action in the CAPnet process is verifying the correctness of the reported puzzle solution. While it would be possible to keep a record of the puzzle challenges and their solutions for each client, we devise a computationally-lightweight technique that does not require maintaining any per-client state.

In this technique, the publisher generates a unique secret token for each content request. This is done by evaluating a secret PRF over the request number and the client IP. The publisher then encrypts the secret token using the puzzle solution, and sends the encrypted token to the client as part of the request bundle. Once the client solves the puzzle, it can decrypt the token and send it back to the publisher along with the request number. The publisher can simply evaluate the secret PRF over the request number and the client IP, and thus, verifies whether the output equals to the token value reported by the client. This enables the publisher to quickly check the correctness of a puzzle solution.

3.4 Security Analysis

In this section, we analyze the effectiveness of CAPnet in fighting cache accounting attacks. We begin by outlining the setup of this analysis, after which we discuss how the security of CAPnet changes as the configuration of the design parameters changes.

3.4.1 Setup

The analysis setup defines how we model our adversaries, and explains the security properties that CAPnet is designed to achieve. The set of notations that this analysis uses is shown in Table 3.1.

Adversary Model. We consider a client colluding with a set of $n_m \geq 1$ caches[4]. (If a client does not collude with any cache, it must retrieve all the data chunks to solve the puzzle just like an honest client.) This collusion can be modeled as an interaction between two parties: the client and a collective entity $\mathbf{C}_m$. Any cache C_j in $\mathbf{C}_m$ can pool all encrypted data chunks from the rest of the malicious caches at a very low cost. That is, given that each cache has a full copy of the raw content, C_j needs only the session keys of these caches to produce their encrypted data chunks locally. When we say that a client retrieves data pieces from $\mathbf{C}_m$, we mean that this client is interacting with the cache that pooled the data chunks.

In order to have a strong bound on attacker capabilities, we consider an attacker with full knowledge about the piece distribution across all the trial puzzles a client will compute. In other words, the attacker knows the selection frequency of data pieces, i.e., how many times a piece has been processed by all puzzles, in all chunks rather than just in the puzzles for which this attacker has enough prior pieces. The attacker may use this information to retrieve the most frequent pieces when solving the challenge puzzle.

Despite the hash function being modeled as a random oracle with a uniform and random output, this piece frequency still matters. Suppose that we have a chunk composed of 4 pieces that are randomly chosen to be in 4 trial puzzles. Over 90% of the time one of these pieces is chosen at least twice (only about 9% of random draws of 4 items choose one from each). On average more than 1 piece is likely not to be chosen for any trial puzzle. An intelligent attacker would choose the piece used in the most trial puzzles since it gives the greatest chance to solve the challenge. So,

[4]All these caches are different, i.e., not Sybils run on the same machine. This is due to the assumption that a publisher monitors the IP addresses of its caches to detect Sybils.

Table 3.1: Notations.

Symbol	Meaning
n	Number of caches selected to serve a content request.
$\mathbf{C}_m$	Set of malicious caches among the n caches.
n_m	Size of $\mathbf{C}_m$, where $n_m \leq n$.
h_{size}	Hash output size.
$chunk_{size}$	Data chunk size.
$piece_{size}$	Data piece size, where $piece_{size} \leq \frac{h_{size}}{m}$.
$pieces_{total}$	Total number of pieces in a data chunk, where $pieces_{total} = \frac{chunk_{size}}{piece_{size}}$.
R_{puzzle}	Number of puzzle rounds.
Y	A random variable that represents the number of pieces a malicious puzzle solver retrieves.
$\mathbb{E}[Y]$	The expectation of Y.
δ	The ratio between the bandwidth amount a malicious puzzle solver would spend and the amount that an honest solver would use. This is computed as $\delta = \frac{\mathbb{E}[Y]}{n \cdot pieces_{total}}$.

accounting for the fact that the actual frequencies may not be uniform, even with a random function, more accurately models the attacker's capabilities. Furthermore, providing the attacker perfect information about these actual frequencies implies that the security bound we infer will be conservative.

The client and $\mathbf{C}_m$ want to solve the puzzle while expending as little bandwidth as possible. In quantifying this cost, we compute the download bandwidth consumption for the colluding group.

Our adversary model is subject to the following assumptions:

1. **Secure cryptographic primitives.** Efficient adversaries cannot break the basic cryptographic building blocks (SHA256, AES, and PRFs) with non-negligible probability.
2. **Clients do not already possess the content.** At the beginning of a service session, a client does not have a copy of the content it will request. This can be achieved by having publishers track which clients have retrieved which content. However, even if this assumption is violated, the client still must retrieve data

chunks from honest caches in order to solve a puzzle, leading to the retrieval of at least $\delta = \frac{n-n_m}{n}$ of the requested chunks.

3. **Free adversarial metadata communication.** It is difficult to know the minimal size of information adversaries would need to communicate when coordinating the puzzle solving process. Therefore, we will just assume that such costs are free from a bandwidth standpoint and only count data piece transmission. While it ignores some cost, this makes the overhead numbers conservative in that real attackers will incur *more cost* than what we predict.
4. **Content is already compressed.** The raw content distributed by caches is already compressed. As such, a malicious cache who intends to compress the raw content before encrypting it will only save a very small bandwidth amount.

In addition to the above, and as mentioned previously, we work in the random oracle model (i.e., hash functions are modeled as random oracles), and in the ideal cipher model (i.e., block ciphers are modeled as random permutations).

An intelligent client and $\mathbf{C}_m$ collaborate to solve the puzzle while transferring the least amount of data possible. In this collaboration, the client receives encrypted chunks only from honest caches, while $\mathbf{C}_m$ produce all encrypted chunks of malicious caches by pooling their session keys as explained previously. The strategy then will have a *solver*, either the client or $\mathbf{C}_m$, that attempts to solve the puzzle using the chunks it has in addition to information it requests from the second party, whichever of the client or $\mathbf{C}_m$ is not the solver. The second party acts as a *piece provider* which sends pieces or hashes (i.e., piece locations computed in a puzzle round) to the solver. For reasons we will see shortly, pieces make more sense for the attacker to transmit.

As our analysis will show, the client and $\mathbf{C}_m$ can decide in advance which party will play which role based on which option incurs the least bandwidth cost. This decision depends on the number of malicious caches n_m. If this number is less than half, i.e., $n_m < \frac{n}{2}$, it is more efficient for the client to act as the solver. If it is greater than half, it is more efficient for the client to let $\mathbf{C}_m$ solve the challenge puzzle. If the number of caches is exactly half, it is equally efficient regardless of who is the solver.

Other attack strategies may involve attacking the cryptographic primitives used

in CAPnet design. This may include trying to find the preimage of the puzzle challenge by inverting the hash, or predicting the session keys used by honest caches to produce their encrypted chunks locally by a colluding cache. By the security of the underlying cryptographic primitives CAPnet employs, i.e., the use of secure PRFs and first-preimage resistant hash functions, such strategies will succeed with negligible probability.

It should be noted that the above list is not known to be comprehensive. There could be other attack strategies outside the analysis presented in this section (although we are not aware of any of such strategies). Performing a rigorous analysis over the full potential threat space is an open problem for future work.

Security Goal. The goal is to ensure that a malicious puzzle solver cannot solve the challenge puzzle in CAPnet unless it expends, on average, a bandwidth amount equivalent to retrieving at least δ portion of the content. This means that the colluding group is expected to expend a total of $\delta \cdot n \cdot chunk_{size}$ bandwidth units. So for $\delta = 0.95$, the attacker has an expected value of 95% of the bandwidth cost even if all metadata overhead are ignored. System designers may set the value of δ based on the security-efficiency trade-off they want to achieve. A larger δ value provides stronger security guarantees, but also increases the computational cost of generating and solving puzzles.

It is not practical to have $\delta = 1$ unless the publisher touches every piece of the requested chunks. Since each chunk must be encrypted with a fresh key, this cost is prohibitive. In fact, if the publisher is willing to touch every piece, it is simpler to compute the hash of the encrypted chunks, and use this hash as the confirmation that a client has to compute. However, this would greatly reduce performance. Assuming that the publisher does not touch every piece, then $\delta < 1$ for the following reason. Suppose that the attacker retrieves every piece of the content except one. If this piece was not touched by the publisher, the attacker can prove that the content was retrieved with $\delta < 1$. Since we only account for the piece transfer costs, at least some of the time (when the attacker does not retrieve untouched pieces) $\delta < 1$, which

makes the expected value of all cases to be less than 1.

3.4.2 Analysis of Puzzle Solving Strategies

In what follows, we analyze the bandwidth cost of the collaborative malicious puzzle solving strategies described earlier, and show how to configure CAPnet's design parameters to achieve the desired δ-bound. These parameters include the data piece size $piece_{size}$ and the number of puzzle rounds R_{puzzle}.

As mentioned previously, an attacker who wishes to solve the puzzle without retrieving all the data chunks will either exchange hashes or retrieve data pieces. By setting the $piece_{size} \leq \frac{h_{size}}{n_m}$ one can ensure that the cost of transmitting a hash is no less than transmitting pieces. That is, even in the event when the piece provider has n_m consecutive encrypted chunks, meaning that given one hash value the provider can process n_m pieces in a puzzle round, transmitting a hash is more expensive than transmitting these n_m pieces. In fact since the pieces may be used in multiple puzzle trials, it is better for the solver to retrieve them. For this reason, we focus on strategies that involve piece dissemination instead.

When retrieving data pieces to solve the challenge puzzle, we conjecture that the best strategy for the solver is to utilize its knowledge of the piece distribution across all trial puzzles. Initially, the solver must possess some piece of each encrypted data chunk to have a chance to solve the puzzle, since each round touches all encrypted data chunks. In order to get pieces from the honest caches, the client must download all double encrypted chunks held by these caches. For malicious caches, the client can retrieve the individual pieces it desires. In selecting which pieces to retrieve, the best approach is to ask for the piece that gives the greatest chance of solving the puzzle. The solver can ask the piece provider to send the piece with the highest frequency among the remaining pieces, and then determine if it enables solving the challenge puzzle. This process continues until the solution is found. Assuming that retrieving the most popular missing piece is optimal, this is the optimal strategy for the malicious solver.

Recall that either the client or $\mathbf{C}_m$ may play the role of the puzzle solver. If $\mathbf{C}_m$ is

the puzzle solver, the client must still be the one to download the chunks from honest caches since the source IP of the request is checked. Hence, the client retrieval from honest caches is a fixed cost. Once this happens, it is more efficient for the party with the most content (either the client or $\mathbf{C}_m$) to act as the solver and get as few pieces as possible from the other party. This means that when $n_m < \frac{n}{2}$, the client will have a larger number of chunks than $\mathbf{C}_m$, thus, the client will be the puzzle solver. On the other hand, when $n_m > \frac{n}{2}$, $\mathbf{C}_m$ will be the puzzle solver asking the client to send pieces from the chunks it received from honest caches. When $n_m = \frac{n}{2}$, either party can be the puzzle solver.

Analyzing the bandwidth cost of the above strategy allows us to configure the number of puzzle rounds to obtain a specific δ-bound. In order to do so, we compute the expected number of pieces $\mathbb{E}[Y]$ the colluding group will retrieve as a function of R_{puzzle} and the number of malicious caches n_m. Then, we compute $\delta = \frac{\mathbb{E}[Y]}{n \cdot pieces_{total}}$, after which we select R_{puzzle} that satisfies the desired δ-bound. To compute $\mathbb{E}[Y]$, we conduct simulations in which we mimic the above strategy and track the number of retrieved pieces. As an example, we consider the following setup, which we believe is similar to what is used in practical content distribution applications. We set $chunk_{size} = 1$ MB, and $piece_{size} = 16$ bytes, leading to $pieces_{total} = 2^{16}$ pieces. We have $R_{puzzle} \in \{1, \ldots, 10\}$, $n = 6$, and $n_m \in \{1, \ldots, 6\}$. The simulations are repeated 10^3 times, where $\mathbb{E}[Y]$, and consequently δ, is computed as the average across all runs. We also report the standard deviation of our measurements. The computed δ values are found in Table 3.2.

As shown, as the number of rounds increases, δ increases. This is expected because a larger number of puzzle rounds means that the challenge puzzle requires a larger number of pieces to be solved. Consequently, the puzzle solver is expected to retrieve more content in order to find these pieces. On the other hand, δ decreases as the number of malicious caches increases for a fixed R_{puzzle} value. Again, this is expected because more malicious caches makes the collusion more effective.

Note that scenarios where $\mathbf{C}_m$ is the solver have significantly lower δ values. This is because $\mathbf{C}_m$ already possesses the majority of the content that has been pooled at

Table 3.2: The δ-bound for various n_m and R_{puzzle} values, $n = 6$ caches (R is R_{puzzle}). For $n_m < 3$ the client is a more efficient puzzle solver, for $n_m > 3$ $\mathbf{C}_m$ is a more efficient puzzle solver, $n_m = 3$ is equivalent for each.

	Client as solver			Either	Cache as solver		
R \ n_m	**0**	**1**	**2**	**3**	**4**	**5**	**6**
1	1	0.87±0.03	0.78±0.06	0.71±0.08	0.45±0.06	0.21±0.03	0
2	1	0.91±0.04	0.86±0.06	0.82±0.08	0.52±0.06	0.24±0.04	0
3	1	0.93±0.04	0.9±0.05	0.87±0.07	0.57 ±0.05	0.26±0.04	0
4	1	0.94±0.03	0.92±0.05	0.91±0.06	0.59±0.05	0.28±0.03	0
5	1	0.95±0.03	0.94±0.04	0.93±0.04	0.6±0.05	0.29±0.03	0
6	1	0.96±0.03	0.95±0.04	0.94±0.04	0.61±0.04	0.29±0.03	0
7	1	0.96±0.02	0.95±0.02	0.95±0.04	0.62±0.04	0.3±0.03	0
8	1	0.97±0.02	0.96±0.03	0.95±0.03	0.63±0.03	0.3±0.02	0
9	1	0.97±0.02	0.97±0.03	0.96±0.03	0.63±0.03	0.3±0.02	0
10	1	0.97±0.02	0.97±0.03	0.97±0.03	0.64±0.03	0.31±0.02	0

no cost (since we assume that metadata retrieval, such as keys, is free, pooling session keys costs no bandwidth). To solve the challenge puzzle, $\mathbf{C}_m$ only needs to retrieve the missing pieces from the client who has the data chunks from the honest caches.

3.5 Evaluation

In order to understand how CAPnet's security impacts efficiency, this section evaluates its performance in the context of content distribution applications. Given that CAPnet imposes a minimal bandwidth cost to exchange a puzzle challenge and its solution, what is left to measure is its computational overhead. Towards this end, we conduct empirical experiments to answer the following specific questions:

- How fast can a publisher generate challenge puzzles?
- How quickly can a client solve these puzzles?
- How does the configuration of the design parameters affect these results?

The rest of this section describes our methodology and discusses the significance of the obtained results.

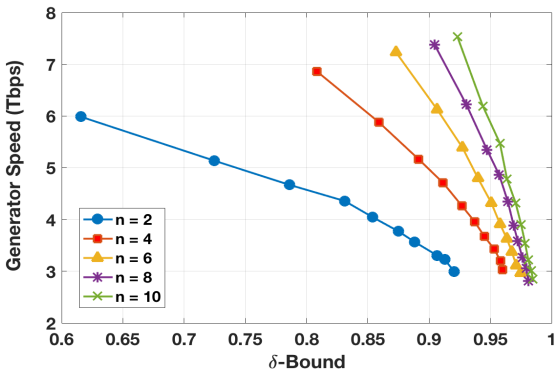
3.5.1 Methodology

To establish our benchmarks, we measured the rate, in puzzles per second, at which a publisher can generate challenge puzzles, and the rate at which a client can solve these puzzles. For the publisher, we considered the case of popular content that large numbers of clients routinely request in close time intervals. For the client, we computed the puzzle solving rate based on the average case, meaning that a client tries half the starting pieces in the first data chunk to find the solution.

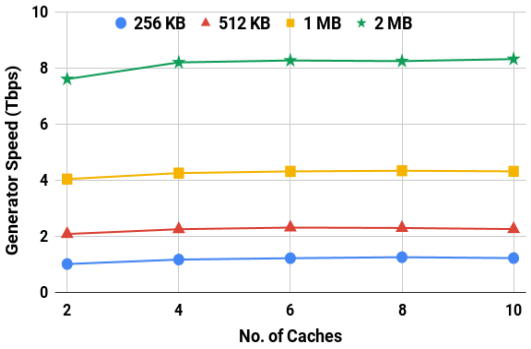
Our experiments were conducted on a modest publisher server with an AMD Ryzen 3 2200G processor and 16 GB of memory, and a low-end client machine with an Intel Core i7-4600U processor and 8 GB of memory. Each puzzle generator and solver has been called at least 5,000,000 and 5,000 times, respectively. Unless otherwise mentioned, all graphs use $R_{puzzle} = 5$, $chunk_{size} = 1$ MB, $h_{size} = 32$ bytes, and $piece_{size} = 16$ bytes. In addition, instead of reporting the puzzle rate for puzzle generator and solver, we compute the bitrate at which content can be requested using these puzzles. Despite both the client and publisher operations being embarrassingly parallelizable, we run each on a single core to show the per-core performance.

3.5.2 Results

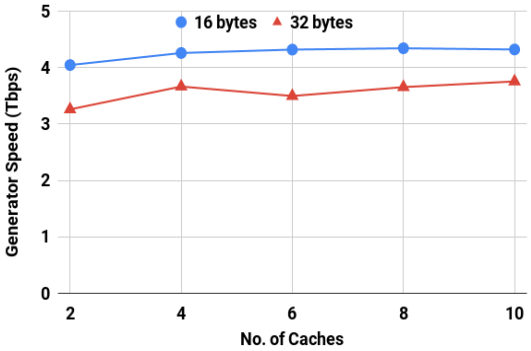
Publisher's bitrate vs δ. We begin by measuring the puzzle generation rate while varying the number of puzzle rounds R_{puzzle} and number of caches n with one malicious cache (Figure 3.3a). As shown in the figure, we compute the δ-bound value that corresponds to each R_{puzzle} value. This produced a curve from 1 round (upper left point on each curve) to 10 rounds (lower right point). The bitrate decreases as δ increases because with larger R_{puzzle} the publisher processes a larger number of pieces when preparing the challenge, which reduces the puzzle generation rate. On the other hand, an increased number of caches n increases the throughput because more data is served per challenge puzzle. This factor also affects the δ-bound of CAPnet. As shown in the figure, for larger n the range δ gets larger for all R_{puzzle} values. That is, the impact of having a malicious cache decreases when n gets larger. This captures



(a) Generator speed (Tbps), δ effect.



(b) Generator speed (Tbps), $chunk_{size}$ effect

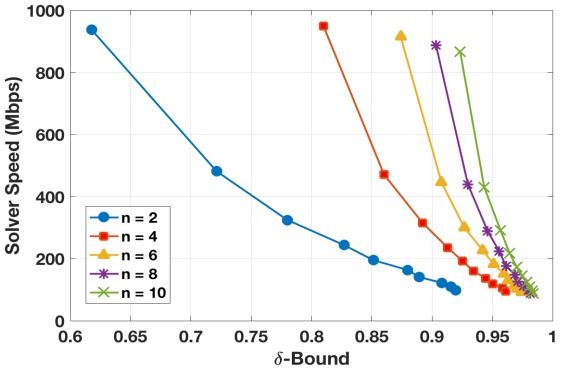


(c) Generator speed (Tbps), $piece_{size}$ effect

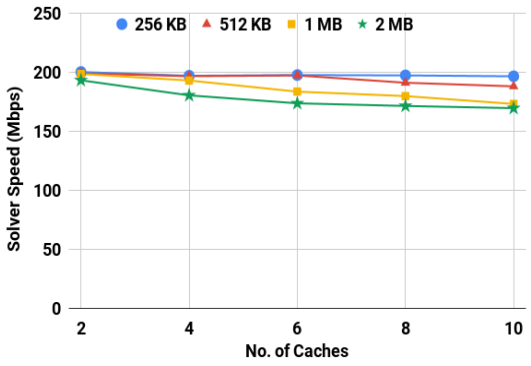
Figure 3.3: Generator speed for various configurations (n is the number of caches in a service session).

what happens in real life, where it will be harder for caches to collude effectively when there is a large number of caches per session. Based on the figure, setting $R_{puzzle} \geq 5$ in practice provides a reasonable δ-bound for $n \geq 4$ ($\delta \geq 0.93$), with diminishing returns thereafter.

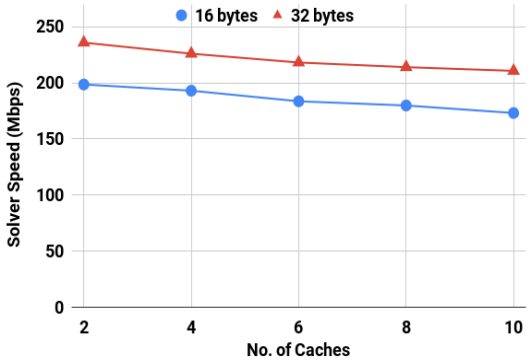
Client's bitrate vs δ. Figure 3.4a shows the bitrate vs δ-bound for the puzzle solver. As shown, for a fixed R_{puzzle} value, the client's effective bandwidth is relatively uniform independent of the number of caches n. However, the R_{puzzle} value, for a fixed n, has substantial impact on the effective bandwidth of a client. Given that the reported speed in the figure dwarfs the 5 Mbps Netflix 1080p quality video rate [37], even using $R_{puzzle} = 5$ (the 5^{th} point on each curve starting from the left), our modest client machine is able to watch dozens of 1080p videos concurrently. If higher performance is desired, then reducing R_{puzzle}, i.e., reducing δ, provides drastically better performance, up to 900Mbps, if needed.



(a) Solver speed (Mbps), δ effect.



(b) Solver speed (Mbps), $chunk_{size}$ effect



(c) Solver speed (Mbps), $piece_{size}$ effect

Figure 3.4: Solver speed for various configurations (n is the number of caches in a service session).

How does $chunk_{size}$ impact client and publisher bitrates? We measured the puzzle generation and solving rates for various data chunk sizes and n values (results are shown in Figures 3.3b and 3.4b). The $chunk_{size}$ has a large effect on the performance of publishers, but minimal effect on client's performance. There are several reasons for this difference. The publisher has almost a fixed puzzle generation rate regardless of the chunk size, because it processes the same number of pieces for fixed R_{puzzle}, $piece_{size}$, and n values. Consequently, a larger $chunk_{size}$ makes the amount of content served per challenge puzzle larger. Alternatively, for a client the puzzle solving rate decreases with larger $chunk_{size}$ because the client has to compute a larger number of trial puzzles. When calculating the bitrate for some n value, the low puzzle rates are multiplied by large $chunk_{size}$ and vice versa. For this reason the client bandwidth is somewhat similar for all $chunk_{size}$ values.

How does $piece_{size}$ impact client and publisher bitrates? To understand

how to set the piece size, we studied its effect on publisher (Figure 3.3c) and client (Figure 3.4c) performance. The publisher can generate enough puzzles to serve over 3 Tbps, regardless of the piece size. However, a piece size of 16 bytes is slightly more efficient because AES-CTR works on 16 byte blocks for encryption. In addition, a smaller piece size means that the publisher processes a smaller amount of content for a fixed number of pieces. The client, on the other hand, tends to benefit from larger piece sizes because they reduce the number of starting pieces, and hence, trial puzzles, a client has to compute. Given that a client with $piece_{size} = 16$ byte already has a high throughput, and given that publishers are usually heavy-loaded entities, we recommend the use of $piece_{size} = 16$ bytes to boost publisher performance.

In summary, the previous results demonstrate that CAPnet is computationally-lightweight. Its security in fighting cache accounting attacks is substantial ($\delta > 0.95$ with generous attacker assumptions), even at bandwidth values that support a publisher serving several Tbps or a client simultaneously watching dozens of 1080p videos.

3.6 Conclusion

In this chapter, we introduced CAPnet, a low-overhead solution to defend against cache accounting attacks in peer-assisted CDNs. CAPnet is the first system that forces malicious caches, even when colluding with clients, to expend substantial bandwidth to demonstrate that content was retrieved. This is done by introducing a cache accountability puzzle that provides strong protections even given unrealistically strong assumptions about the attacker's capabilities. For example, with a 5 round puzzle, if 3 malicious caches out of 6 total caches wish to perform a cache accounting attack, the colluding parties would retrieve on average more than 0.95 of the requested content ($\delta > 0.95$). We analyze the security of CAPnet, and show experimentally that it incurs a low computation cost. This demonstrates the viability of employing our scheme in large scale content distribution applications.

CAPnet is one of the modules that CacheCash employs to ensure security at low

cost. In the next chapter, we introduce another module, namely, MicroCash, that addresses the issue of processing micropayments securely and efficiently.

Chapter 4

MicroCash: Practical Concurrent Processing of Micropayments

This chapter is based on joint work with Allison Bishop and Justin Cappos [56].

4.1 Overview

As mentioned previously, micropayments provide a flexible payment paradigm with a large set of potential applications, such as ad-free web surfing, online gaming, and peer-assisted service networks [111]. To reduce the overhead of processing these small payments, several probabilistic solutions were proposed that allow local exchange of micropayments while aggregating them into few transactions with larger values before processing [70, 99, 111, 114, 115, 129].

At a high level, in these schemes the total amount of payments is locked in an escrow and micropayments take the form of lottery tickets instead of actual transactions. Each ticket enters a lottery where it has a probability p of winning, and when it wins, produces a transaction of β currency units. This means that only one large transaction, on expectation, is processed among a batch of $\frac{1}{p}$ tickets, which reduces the processing costs. Unfortunately, as discussed in Section 1.1, the older solutions [99, 114, 115, 129] require a trusted party to audit the lottery and manage payments. While the new, fully distributed ones [70, 111] support only sequential issuance of lottery tickets using the same escrow because an escrow can pay only one winning ticket. Also, they incur large computation and communication overhead, where they rely on computationally-heavy cryptographic primitives, or require multiple rounds to exchange a payment.

To address these drawbacks, we propose MicroCash, the first decentralized probabilistic framework that supports *concurrent* micropayments. MicroCash features a novel payment setup that allows a customer to issue lottery tickets in parallel and at a fast rate using a *single* escrow. This is achieved by having the customer specify the total number of tickets it may issue and the rate at which it will issue them, and provide an escrow balance that covers all winning tickets with very high probability (with probability $> 1 - \epsilon$ for some small system parameter ϵ). This balance coverage is coupled with ticket tracking, based on sequence numbers, to detect if a customer exceeds the ticket issue rate of its escrow.

MicroCash is also cost effective because it introduces a lightweight non-interactive lottery protocol, requiring only hashing, that allows a payment exchange with only one round of communication. In this protocol, each issued ticket is tied to a *lottery draw* value in a future block on the blockchain. A ticket wins if hashing this value along with the ticket produces an output below a threshold computed based on p. We show that such a simple and highly efficient protocol is secure in the random oracle model and under proper assumptions regarding the security of the underlying cryptocurrency system.

Moreover, as a probabilistic micropayment scheme, MicroCash provides the following properties. First, it enhances system scalability, in terms of the blockchain size and transaction throughput, by tuning the payment setup parameters to reduce the on-chain traffic. Second, it reduces interaction between participants and allows processing of a large number of off-chain payments, or lottery tickets, at a fast rate. This is possible because verifying a ticket involves only lightweight operations and requires just local information merchants have about the escrow setup. Third, MicroCash neutralizes the security threats that arise in probabilistic micropayments using both cryptographic and financial techniques. The latter requires any customer to tie its payment escrow to a penalty deposit that is revoked upon cheating, with a lower bound derived using a game theoretic analysis.

We conduct thorough benchmarks to evaluate MicroCash's performance, where we compare it to a state-of-the-art sequential micropayment scheme, MICROPAY [111].

Our results show that a modest merchant machine in MicroCash is able to process 2,250 - 10,400 ticket/sec, which is around 1.67-4.1x times the rate in MICROPAY, with 60% reduction in the aggregated payment size. Furthermore, a modest customer machine in MicroCash is able to concurrently issue more than 32,000 micropayment/sec using one escrow, while MICROPAY requires the customer to create more than 1000 escrows to support a comparable issue rate. This allowed MicroCash to reduce the transaction fees and the additional blockchain size by around 50%.

4.2 Related Work

In this section, we review prior work done in the area of probabilistic micropayments. In addition, we review an alternative payment aggregation mechanism, called payment networks [72, 112], focusing on its limitations when used to handle micropayments.

Probabilistic Micropayments. The idea of employing a lottery for micropayment aggregation dates back to the seminal works of Wheeler [129] and Rivest [114]. In these schemes, a customer and a merchant run an interactive lottery based on a simple coin tossing protocol [114]. This was optimized later by developing a non-interactive lottery protocol that a merchant can run locally [115]. All of these schemes assume the presence of a centralized bank to hold accounts for customers and merchants, authorize customers to issue lottery tickets, and process winning tickets. The existence of a trusted bank imposes additional overhead on the users who have to establish complex business relationships with this bank. Also, it limits the use of the payment service to only fully authenticated users. Therefore, this centralization issue is viewed as the main reason why there has been only limited adoption of such solutions [70].

Cryptocurrency-based probabilistic micropayments provide a potential solution to address this centralization problem. This paradigm replaces the trusted bank and its private records with the miners and the blockchain. In addition, it allows anyone to join without any identity authentication.

To the best of our knowledge, only two distributed schemes have been proposed to date in the literature, MICROPAY [111] and DAM [70]. MICROPAY translates the scheme of Rivest [114] into an implementation on top of a cryptocurrency system. Instead of using an authorized bank account, customers create escrows on the blockchain that they use to issue lottery tickets. For the lottery protocol, MICROPAY adopts the same interactive coin tossing protocol mentioned above, but also adds an alternative non-interactive version that reduces the communication complexity. However, the latter is computationally-heavy because it requires public key cryptography-based operations and a non-interactive zero knowledge (NIZK) proof system to prove the correctness of the lottery draw. Moreover, MICROPAY only supports sequential micropayments. That is, because a winning ticket receives all escrow funds, a customer cannot issue a new ticket using the same escrow until it is confirmed that the previous ticket did not win. This means that there must be a new escrow on the blockchain for every winning ticket. DAM shares the same sequential constraint but it adds the feature of preserving user privacy (not like MICROPAY that is public), where it extends Zerocash [117] primitives to implement anonymous micropayments.

We believe that the added blockchain transactions needed to create escrows due to the lack of payment concurrency, and the reliance on computationally heavy primitives, may reduce the usefulness of these schemes in large-scale distributed systems. Thus, there is a need for optimized approaches that allow concurrent micropayments at a lower overhead. This need is the motivation behind building MicroCash.

Payment Channels and Networks. This payment paradigm was originally developed to handle micropayments in Bitcoin [13], where it relies on a similar concept of processing most of the small payments locally. However, later on it was geared primarily toward enhancing the scalability of cryptocurrencies [72, 86], where, particularly in Bitcoin, the limited block size and the slow block generation rate allow handling only few transactions per second. This is done by utilizing off-chain pro-

cessing to reduce on-chain traffic helps in increasing the transaction throughput at a lower overhead.

A (micro)payment channel is a contract between a customer and a merchant tied to a shared escrow fund. The ownership of this fund is adjusted over time based on the off-chain transactions, or local payments, made to date. Merchants collect their accumulated payments by releasing the most recent transaction expressing the latest state of the fund ownership. As such, only two transactions are logged on the blockchain per channel, the opening transaction and the closing one, regardless of the number of local payments.

Several schemes were proposed to implement this paradigm. This includes payment channels that allow exchanges only between two parties [13,72,86], and payment networks, e.g., lightning networks [112], that allow any two parties to pay each other if they are connected by a payment path (a consecutive set of payment channels). In addition, several solutions were introduced to optimize these initial proposals. For example, Sprites [101] uses smart contracts to optimize the collateral cost of the lightning networks. Bolt [82] addresses user privacy by constructing paths that hide the identities of the transacting parties and the payment value. And Fulgor/Rayo [98] handle payment linkability in these networks, and prevent deadlocks when concurrent payments cannot be completed due to insufficient channel capacity.

In general, payment networks suffer from the high collateral cost of setting up multiple escrows when constructing payment paths. These costs may indirectly push the network towards centralization [30]. This is because only wealthy parties can afford locking currency in multiple escrows to establish payment channels, and hence, most users will rely on these parties, or hubs, to relay the off-chain transactions. In addition, these networks charge for payment relaying, so a user will have to pay fees to each hub along the path. With micropayments, such a setup would be infeasible because these fees could be much larger than the payments themselves. Probabilistic approaches, on the other hand, are more flexible in allowing several parties to be paid using the same escrow. And by doing so, they reduce the collateral cost and eliminate any fees when exchanging lottery tickets. Hence, distributed probabilistic

micropayments provide a better solution for handling small payments in cryptocurrency systems.

4.3 Threat Model

As discussed in the previous sections, distributed probabilistic micropayments rely on off-chain processing. Merchants accept lottery tickets, as opposed to transactions logged on the blockchain, with a promise that a customer will pay later when a ticket wins the lottery. This local processing of future payments creates the potential for various types of attacks. In this section, we outline a threat model that accounts for these attacks, which guided the design of MicroCash.

In developing this model, we make the following assumptions:

- No trusted party exists.
- Participants are rational, meaning that they may follow the protocol without violation, or deviate from it (either on their own or in collusion with each other), based on what will maximize their utility gain.
- The underlying cryptocurrency scheme is secure in the sense that the majority of the mining power is honest. This means that the confirmed state of the blockchain contains only valid transactions, and that an attacker who tries to mutate or fork the blockchain will fail with overwhelming probability.
- Hash functions are modeled as random oracles, and the hash values of the blocks on the blockchain are modeled as a uniform distribution. This implies that predicting the hash of a future block succeeds with very low probability.
- Efficient adversaries cannot break the basic cryptographic building blocks (SHA256, digital signatures, etc.) with non-negligible probability.
- Communication between customers and merchants takes place over secure channels such as TLS/SSL.

We used ABC (Chapter 2) to build a threat model for MicroCash[1]. During this process, we identified the assets to be protected in distributed probabilistic micro-

[1]A detailed version of the threat model is available online [31]

payments, which include the escrows, the lottery tickets or payments, and the lottery protocol. Then, by analyzing the security requirements of these assets, we produced the broad threat categories in such systems. Our list includes the following:

- **Escrow overdraft:** A customer creates a payment escrow insufficient for honoring the winning lottery tickets tied to this escrow, or creates a penalty escrow that does not cover the cheating punishment imposed by the miners. Such a threat could be the result of creating small balance escrows, or front running attacks in which a customer withdraws the escrows before paying.
- **Duplicate ticket issuance[2]:** A customer issues lottery tickets with the same sequence number to several merchants. This leads to issuing more tickets than what the escrow can cover, i.e., not all winning tickets will be paid. In this way a customer obtains more service than what it pays for.
- **Invalid payments:** A customer hands merchants lottery tickets that do not comply with its payment setup or with the system specifications. Because these tickets will be rejected by the miners if they win the lottery, the customer can avoid paying merchants.
- **Unused-escrow withholding:** An attacker prevents or delays a customer from withdrawing its unused escrows. For example, merchants may delay claiming their winning lottery tickets to keep the payment escrow on hold.
- **Lottery manipulation:** An attacker attempts to influence the outcome of the lottery draw, and hence, bias the payment process.
- **Denial of service (DoS):** This is a large threat category that includes all attack forms that interrupt the system operation and make it unavailable for legitimate users. We focus on attacks related to the payment process. For example, an attacker may monitor the network and drop all lottery tickets or escrow transactions to disturb the service.

Note that in this chapter, we are concerned with the payment scheme design,

[2]This threat is sometimes called double spending in the literature [70, 111]. We refer to it as duplicate ticket issuance to distinguish this threat from double spending attacks on on-chain cryptocurrency transactions.

rather than how to exchange service with a payment, which is part of an application design (part of CacheCash in this work, which we discuss in detail in Chapter 5). That is, dealing with malicious merchants who collect lottery tickets and do not deliver a service is outside the scope of MicroCash, same for dealing with malicious customers who may obtain the service without paying.

In addition, MicroCash does not address payment anonymity (as in [70]). Addressing this issue securely, while preserving the low overhead of our scheme, is a direction of our future work.

4.4 MicroCash Design

Having outlined the security threats to probabilistic micropayments in the previous section, and the limitations of existing solutions in Section 4.2, this section presents the design of MicroCash, a concurrent micropayment system that addresses these issues. We start with an overview of the system operation, followed by a more detailed technical description.

4.4.1 MicroCash in a Nutshell

At a high level, MicroCash operation proceeds in three phases, payment setup, exchange, and redemption, as illustrated in Figure 4.1. As shown, during the *payment setup* (**Step 1**, Section 4.4.2), each customer issues a transaction creating two escrows: payment and penalty. The customer uses the former to make payments in the form of lottery tickets, where this escrow is sufficient to pay all winning tickets with probability $1-\epsilon$ for some small ϵ. While the miners use the latter to financially punish this customer if it cheats, e.g., by issuing tickets with duplicated sequence numbers to different merchants. This transaction is embedded with information about the customer's payment setup, including the lottery winning probability p, the value of a winning ticket β, and the set of beneficiary merchants that can be paid using this escrow, among others.

The *payment exchange* phase starts as soon as the escrow creation transaction

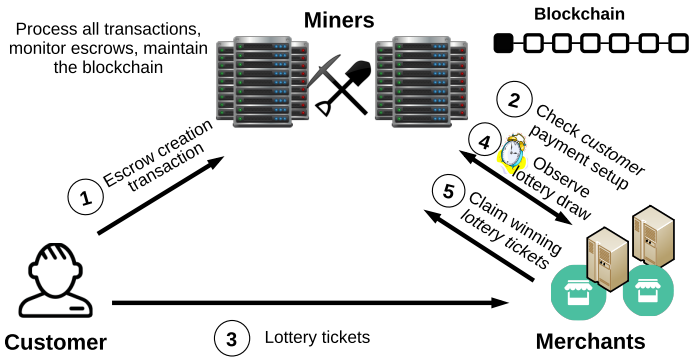


Figure 4.1: MicroCash operation flow.

is confirmed on the blockchain[3]. At that time, merchants can check the escrow setup before transacting with the customer (**Step 2**). In exchange for the delivered service, the customer can issue these merchants lottery tickets as payments (**Step 3**, Section 4.4.3). A customer is limited in the number of tickets it may issue over a set period. To provide a general, global nature of time, all parties track the height of the blockchain and limit the sequence numbers of tickets that can be issued during any round, where a round is the time needed to mine a block on the blockchain. Hence, for each round, a customer uses the assigned sequence number range to issue tickets to merchants.

For *payment redemption*, a merchant keeps each of the tickets it received until the lottery draw time of this ticket. It then observes a value derived based on the block mined at that time to determine if this ticket is a winning one (**Step 4**, Section 4.4.4). A ticket wins if hashing the lottery draw value along with the ticket's sequence number and the escrow ID is below a threshold. If it wins, the merchant can claim the currency value of this ticket, i.e., β coins, from the customer's payment escrow during the ticket redemption period (**Step 5**, Section 4.4.5). This is done by presenting this ticket to the miners who, after validating the ticket, approve transferring currency from the customer's escrow to the merchant's address.

After an escrow's time period has elapsed, remaining funds may now be spent as per usual by the owner-customer. Signaling the expiry of an escrow does not need

[3]After having the block that contains this transaction buried under y blocks, e.g., in Bitcoin $y = 6$.

an explicit transaction logged on the blockchain. This is because miners track the timeframe during which a customer may issue tickets using an escrow, and after all tickets have been paid out or expired, a customer can spend the remaining balance.

4.4.2 Escrow Setup

MicroCash introduces a novel escrow setup that allows multiple winning tickets to be redeemed, which enables both concurrent ticket issuance and reduces the amount of data logged on the blockchain. In order to support this feature, our scheme provides techniques to determine the escrow balance needed to cover all concurrent tickets with very high probability, and to track the issuance of these tickets in a distributed way.

Escrow Creation

As an off-chain payment scheme, MicroCash must have a way to ensure that customers can and will pay. This includes honoring winning tickets when redeemed by merchants, and complying with a stipulated financial punishment if this customer cheats. To satisfy this requirement, each customer is required to create payment and penalty escrows with sufficient funds. Creating these escrows involves two actions: a customer configures the payment parameters and locks funds in the escrows; and miners verify that this setup complies with the system specifications.

Given that each payment escrow must be tied to a penalty escrow, a customer sets up both escrows using one escrow creation transaction. This transaction has two types of inputs. The first is the fund to be locked under each escrow balance, where we refer to the payment and penalty escrow balances as B_{escrow} and $B_{penalty}$, respectively. And the second is a set of parameters that influence computing the value of both B_{escrow} and $B_{penalty}$, and how to spend them. These parameters, whose values are specified by the customer possibly after negotiating the service with merchants, include the following:

- The lottery winning probability p.
- The currency value of a winning lottery ticket β.

- The ticket issue rate tkt_{rate}, which is the maximum number of tickets a customer is allowed to hand out per round. This is used to calculate which ticket sequence numbers are valid within each ticket issuing round.
- The escrow lifetime, denoted as l_{esc}, which is the total number of ticket issuance rounds. Therefore, the total number of tickets a customer may issue equals to $l_{esc}tkt_{rate}$.
- The set of beneficiary merchants that can be paid using the escrow, where the size of this set is denoted as m.

Computing the values of B_{escrow} and $B_{penalty}$ based on the above parameters proceeds as follows. To permit concurrent micropayments, B_{escrow} must be large enough to pay all winning tickets tied to an escrow with probability $1 - \epsilon$, for some small ϵ (such as $\epsilon < 0.05$). While $B_{penalty}$ must be large enough to deter cheating in the system. For a specific ϵ value, and given that the payment probability distribution can be modeled as a binomial distribution parameterized by p and number of trials $t = l_{esc}tkt_{rate}$, B_{escrow} can be computed as:

$$B_{escrow} = \beta F^{-1}(p, l_{esc}tkt_{rate}, 1 - \epsilon)$$

where F^{-1} is the inverse of the cumulative distribution function (CDF) of the binomial distribution described above at the value $1 - \epsilon$. The full details of deriving the above equation are found in Section 4.5.1.

For $B_{penalty}$, we compute a lower bound for this deposit using an economic analysis that accounts for the additional utility gain a customer may obtain by cheating. This bound is given by the following equation:

$$B_{penalty} \geq (m-1)tkt_{rate}p\beta\left(\frac{1}{1-(1-p)^{tkt_{rate}}} + d_{draw} + d_{redeem} - 1\right)$$

where d_{draw} is the lottery draw period in rounds and d_{redeem} is the ticket redemption period in rounds (more about these parameters in Section 4.4.4). This lower bound ensures that the financial punishment exceeds the additional utility gain of cheating,

and hence, makes cheating unprofitable compared to acting honestly. The full details of deriving this bound are found in Section 4.5.2.

Verifying the correctness of a payment setup is performed by the miners upon receiving the escrow creation transaction. They validate the format of this transaction, which involves verifying that the customer owns the input funds. Then, the miners check that the value of $B_{penalty}$ satisfies the above bound, and that B_{escrow} obeys the $1 - \epsilon$ coverage rule. The latter is done by computing the number of winning tickets ψ that the payment escrow can cover, i.e., $\psi = \frac{B_{escrow}}{\beta}$. And then, verifying whether $F(\psi) \geq 1 - \epsilon$, using the CDF formula of the binomial distribution (again, parameterized by p and $t = l_{esc}tkt_{rate}$). If all these checks pass, the miners add the escrow transaction to the blockchain. Otherwise, they reject the escrow by dropping its transaction.

Escrow Management

In MicroCash, the locked funds in the escrows can be spent only for a restricted set of transactions. This set includes claimed winning tickets, proofs-of-cheating, and (after the escrow lifetime is over) enabling a customer to spend its unused escrow funds.

To track the locked funds, miners maintain a state for each escrow in the system. This state includes the following:

- The ID of the escrow, which is a random value generated by the miner that adds the escrow creation transaction to the blockchain.
- The balances of both the payment and penalty escrows.
- The public key of the owner-customer, which is used to verify all signed tickets that are issued using this escrow.
- The values of p, β, l_{esc}, and tkt_{rate}, and the set of beneficiary merchants.
- A refund time for the escrow, denoted as t_{refund}, at which the owner-customer can spend any remaining funds in both the penalty and payment escrows. The value of t_{refund} is the expiry time of the tickets issued in the last round of an escrow lifetime.

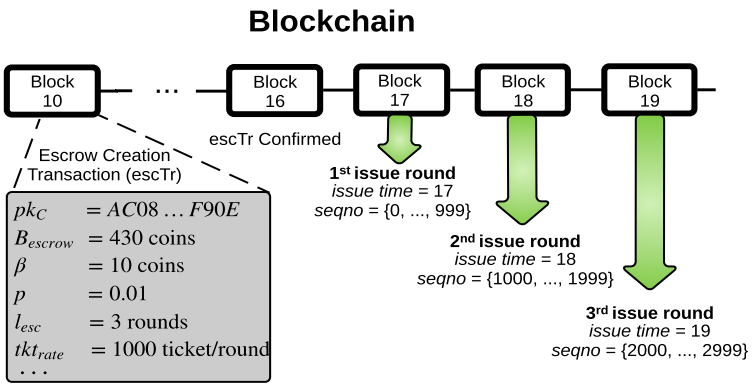


Figure 4.2: Ticket issuing schedule, an example.

Ticket issuance using an escrow must follow a schedule based upon tickets' sequence numbers. That is, if an escrow supports a rate of tkt_{rate} tickets per round, then in the first round tickets with sequence numbers 0 to $tkt_{rate} - 1$ may be issued. Then, tickets with sequence number range tkt_{rate} to $2tkt_{rate} - 1$ can be issued in the second round, and so on until the last round of an escrow lifetime. Merchants will accept tickets in the current round with sequence numbers that follow this assignment schedule. In order to deal with the fact that customers and merchants may have inconsistent view of the blockchain, and hence, may not agree about what the current round is (i.e., current height of the blockchain), merchants will also accept tickets from the prior and next round given that these tickets use the correct sequence number range.

An example of a ticket issuing schedule is found in Figure 4.2. As shown, the escrow creation transaction is published at round 10 and confirmed at round 16 (assuming that a block is confirmed after being buried under 6 blocks). This escrow allows issuing a total of 3000 tickets with 0.99 coverage probability (i.e., $\epsilon = 0.01$ in this case). Thus, at a ticket issue rate of 1000 tickets per round, the customer has 3 ticket issuing rounds, starting at round 17, with the sequence number ranges shown in the figure.

The miners update the escrow state based on the restricted set of the escrow related transactions (mentioned earlier) they process. For example, redeeming a winning ticket reduces B_{escrow} by β coins. And receiving a valid proof-of-cheating

against the customer causes miners to burn the funds in $B_{penalty}$. All these transactions are logged on the blockchain, which permits anyone to validate the current state of any escrow.

The miners discard an escrow state once all tickets tied to this escrow expire, which happens at time t_{refund}, or when an escrow is broken after receiving a valid proof-of-cheating (discussed in Section 4.4.6). At that time, the customer may spend the remaining funds, which is the residual of its payment escrow (if any) and its penalty deposit (if not revoked), normally again.

4.4.3 Paying with Lottery Tickets

After the escrow is confirmed on the blockchain, a customer can start paying for the service by giving merchants lottery tickets. A lottery ticket tkt_L is a structure containing several fields as follows:

$$tkt_L = pk_M||id_{esc}||tseq||\sigma_{\mathcal{C}} \tag{4.1}$$

where pk_M is the public key of the recipient merchant, id_{esc} is the payment escrow ID, $tseq$ is the ticket sequence number, and $\sigma_{\mathcal{C}}$ is the customer's signature over the ticket. The $tseq$ field, along with id_{esc}, identifies a ticket, which also provides means for ticket tracking in the system. Note there is no need to include any information about the escrow setup, including the public key of the owner-customer, in the ticket itself. Merchants and miners can look this up on the blockchain using id_{esc}.

When issuing a ticket, the customer fills in the above fields and signs the ticket using the secret key tied to the public key the customer used when creating the escrow. The ticket $tseq$ can be any sequence number within the range assigned to the current ticket issue round. The customer can continue issuing lottery tickets until it finishes all sequence numbers assigned to the current round. After that, it must wait the next round to generate more tickets.

Upon receiving a ticket, a merchant verifies it as follows:

- Parse the ticket as $(id_{esc}, pk_M, tseq, \sigma_{\mathcal{C}})$.

- Check that the escrow is not broken
- Check that its address, i.e., pk_M that appears in the ticket, is listed as one of the beneficiary merchants of the escrow.
- Verify that $tseq$ is within the valid range based on the ticket issuing schedule. (As mentioned before, to handle inconsistencies in the blockchain view, tickets from the previous or next issuance round can be accepted.)
- Verify $\sigma_{\mathcal{C}}$ over the ticket using the public key of the customer who owns the escrow.

If any of the above checks fails (except the third and the fourth), the merchant simply drops the ticket. Further actions, such as exiting the current payment session, or refusing further transactions with this customer are handled individually by merchants, based on customer behavior and the merchant's own policy. On the other hand, if the ticket has an out-of-range sequence number (i.e., larger than the maximum sequence number allowed) or destined to a merchant that is not listed as a beneficiary of the escrow, the a proof-of-cheating can be issued, which will cost the customer its penalty deposit. Otherwise, if all the above checks pass, the merchant accepts the ticket and keeps it until its lottery draw time.

4.4.4 The Lottery Protocol

MicroCash introduces a lightweight lottery protocol that relies solely on hashing. This protocol does not require any interaction between the customer and the merchant. Instead, it utilizes only the state of the blockchain, where the lottery draw outcome is determined by a value derived from the block mined at the lottery draw time.

To specify the lottery draw time, MicroCash defines a system parameter called d_{draw}. This parameter represents the number of rounds a ticket has to wait after the round in which its sequence number can be issued (which we call t_{issue}) until it enters the lottery. Hence, the draw time t_{draw} of a ticket is computed as $t_{draw} = t_{issue} + d_{draw}$. This means that whether a ticket wins or loses the lottery depends on the block at index t_{draw}. It also means that all tickets belong to the same ticket issuing round will

enter the lottery at the same time.

The lottery draw value is computed using a simple verifiable delay function (VDF) [66] that is evaluated over the block with index t_{draw}. This evaluation takes a period of time, hence the name delay function. Consequently, when a miner mines the block with index t_{draw}, it cannot tell immediately which ticket will win or lose. This miner has to compute the VDF over this block in order to reveal the lottery outcome.

We instantiate the aforementioned VDF using an iterative hash process, where the number of hash iterations is set to a value that delays producing the output by the period specified in the system. In addition, we let the miners compute this function as part of the mining process. That is, when a miner mines a new block, it evaluates the VDF over the previous block. Therefore, the VDF value of the block at index t_{draw} appears on the blockchain when the block at index $t_{draw} + 1$ is mined.

Accordingly, in our protocol a merchant keeps a ticket tkt_L until its lottery draw time t_{draw}. Then, after observing the VDF value of the block mined at that time, the merchant computes the following quantity over this ticket (where H is a hash function):

$$h_{tkt_L} = H(id_{esc}||tseq||VDF(Block_{t_{draw}})) \tag{4.2}$$

A ticket wins if the least significant word of h_{tkt_L} is less than $2^{32}p$. Therefore, within the precision allowed by a 32-bit number, a ticket will have a p chance of winning.

This process is clarified by the example depicted in Figure 4.3. As shown, the ticket has been issued at round 30, and hence, it entered the lottery at round 40. The VDF value of the block with index 40 appears inside block 41. Based on the value of h_{tkt_L}, the ticket in the figure is a winning one.

Note that h_{tkt_L} involves only the ticket fields that are part of the escrow state. In other words, it relies on values that the issuing customer cannot manipulate. These do not include the merchant recipient address, which means that a ticket chance of winning the lottery is not affected by who owns this ticket. In addition, this

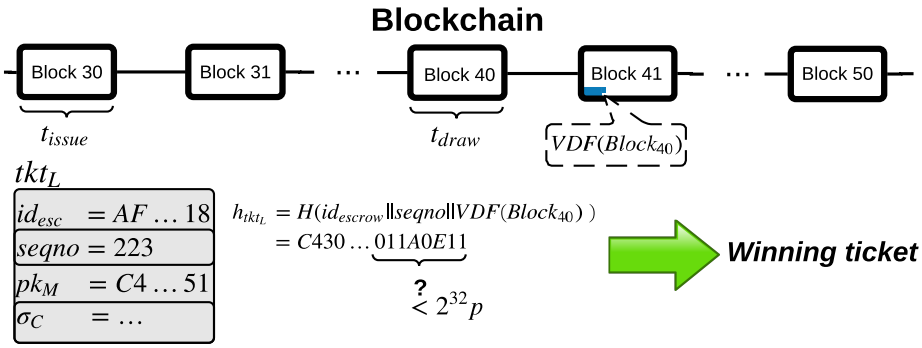


Figure 4.3: Lottery draw example ($d_{draw} = 10$, and $p = 0.01$).

means that if a customer issues a ticket with the same sequence number to multiple merchants (i.e., duplicate ticket issuance), all these tickets will win or lose together. If the tickets win, detecting cheating is trivial because both merchants will publish their winning tickets on the blockchain to redeem the tickets.

4.4.5 Claiming Winning Tickets

After the lottery draw, a merchant can collect currency from the customer's escrow by redeeming its winning tickets (if any). This is done by issuing a redeem transaction that has the winning ticket as input, and has β coins directed to the merchant's address as output.

To allow the miners to resolve tickets and release escrow funds back to the customer in a reasonable timeframe, MicroCash specifies a redeem period for each ticket. This is done by defining a system parameter called d_{redeem} that determines the number of rounds during which a ticket can be redeemed. After this period, a ticket expires, i.e., becomes invalid, which happens at time $t_{expire} = t_{issue} + d_{draw} + d_{redeem}$. Thus, d_{redeem} must span a period sufficient for merchants to redeem their winning tickets.

After receiving a redeem transaction, the miners process it as follows:

- Check that the format of the transaction complies with the system specifications.
- Verify the redeemed ticket as outlined in Section 4.4.3.
- Verify that the ticket is a winning one based on equation 4.2.
- Check that the ticket is not expired.

- Verify the merchant's signature over the redeem transaction using the public key that appears in the winning ticket. This is needed to prevent participants from redeeming tickets they do not own.
- Check that no other ticket with the same *tseq* and tied to the same escrow has already been redeemed. If it is, this is proof of duplicate ticket issuance and is used as a proof-of-cheating against the customer.

If all these checks pass, miners approve the redeem transaction and update the escrow state accordingly. Otherwise, they drop an invalid transaction and, if a proof-of-cheating is produced, revoke the customer's penalty deposit as described in the next section.

4.4.6 Processing Proof-of-cheating

A proof-of-cheating is a special transaction that can be presented to the miners by any party who witnesses a cheating incident. In MicroCash, this transaction can be issued when any of the following cheating behaviors is detected:

- Issuing more tickets than what an escrow can cover, i.e., exceed the maximum *tseq* an escrow may allow.
- Duplicate ticket issuance.
- Issuing tickets to a merchant that is not listed as one of the escrow beneficiaries.

A valid proof-of-cheating must provide an irrefutable, publicly verifiable cheating proof. Accordingly, a signed lottery ticket, or tickets in case of duplicate ticket issuance, that falls under any of the above violations is a publicly verifiable proof against the issuing customer.

Upon verifying a cheating incident, miners punish the customer by breaking the penalty escrow tied to its payment escrow referenced in the ticket. In case of duplicate ticket issuance, the miners first pay all duplicated winning tickets from the payment escrow, if it is sufficient, and from the penalty deposit thereafter. Then, they publish an escrow break transaction containing the proof-of-cheating on the blockchain, which for security reasons that are explained in Section 5.5, burns the revoked penalty

deposit. Respecting the lower bound of $B_{penalty}$, ensures that all the aforementioned cheating behaviors are unprofitable in expectation (i.e., they do not achieve any additional utility gain over acting in an honest way). Hence, it makes such behaviors unappealing to rational customers.

4.5 Economic Analysis for Escrows

In this section, we show how to compute the payment escrow balance B_{escrow} in a way that satisfies the $1-\epsilon$ coverage rule. We also compute a lower bound for the penalty deposit required to deter cheating under MicroCash setup.

4.5.1 Computing B_{escrow}

Computing B_{escrow} is done using a probabilistic analysis that relies on modeling the payment process in MicroCash. In what follows, we state and prove a formula to calculate this balance.

Theorem 1. *For an escrow with lifetime l_{esc} rounds, ticket issue rate tkt_{rate}, lottery winning probability p, winning ticket currency value β, and parameter ϵ, where $l_{esc}, tkt_{rate} \in \mathbb{N}$, $\beta \in \mathbb{R}^+$, and $0 \leq p, \epsilon \leq 1$, the value of B_{escrow} needed to cover all winning lottery tickets with probability at least $1-\epsilon$ under MicroCash setup is given by:*

$$B_{\text{escrow}} = \beta F^{-1}(p, l_{\text{esc}} tkt_{\text{rate}}, 1-\epsilon) \tag{4.3}$$

where F^{-1} is the inverse of the cumulative distribution function (CDF) of the binomial distribution parameterized by p and a number of trials $t = l_{\text{esc}} tkt_{\text{rate}}$ at the value $1-\epsilon$.

Proof. Given that we work in the random oracle model, and that we model the block hashes on the blockchain as a uniform distribution, lottery winning events are independent and can be modeled as Bernoulli trials. This means that the total number of winning tickets tied to an escrow with lifetime l_{esc} rounds, ticket issue rate tkt_{rate}, and lottery winning probability p, is a binomial random variable parameterized by p and a number of trials $t = l_{esc} tkt_{rate}$.

Requiring B_{escrow} to cover all winning tickets with probability $1 - \epsilon$ means that B_{escrow} must contain sufficient currency to pay a number of winning tickets ψ that hits the $(1 - \epsilon)$th percentile of the above binomial distribution, i.e., $B_{escrow} = \psi\beta$. This number can be computed as:

$$\psi = F^{-1}(p, l_{esc}tkt_{rate}, 1 - \epsilon)$$

where F^{-1} is the inverse of the cumulative distribution function (CDF) of the binomial distribution parameterized by p and a number of trials $t = l_{esc}tkt_{rate}$ at the value $1 - \epsilon$. Substituting this expression in $B_{escrow} = \psi\beta$ produces the formula stated in the theorem above, which completes the proof. □

4.5.2 Computing a Lower Bound for $B_{penalty}$

Computing a lower bound for $B_{penalty}$ is done using a game theoretic approach that quantifies the additional utility gain, or monetary profit, a malicious customer could accrue as compared to an honest one. By setting the penalty deposit to at least equal this additional utility, cheating becomes less profitable, in expectation, than acting honestly, and hence, unappealing to rational customers.

In what follows, we present this analysis including the addressed malicious strategies, the game setup, a definition for the utility gain function, and finally state and prove a lower bound for $B_{penalty}$.

Covered malicious strategies. In MicroCash, a penalty escrow is revoked upon the detection of two types of malicious events: issuing duplicated tickets and invalid payments (including those with out-of-range sequence numbers). The utility gain of any of these malicious strategies depends on the length of the cheating detection period, i.e., the time needed to detect a cheating incident. During this time the cheating customer is still perceived as an honest by merchants, meaning that this customer still can cheat and increase its utility gain. Consequently, the longer the detection period the larger the additional utility.

Given that merchants verify each ticket immediately when received, invalid payments are detected instantly. On the other hand, duplicated tickets are detected after they are redeemed (if they win the lottery), which may happen after several rounds in MicroCash. This means that the additional utility gain of ticket duplication is larger than the one obtained by issuing invalid payments. Therefore, setting the value of the penalty deposit to be at least equal to the additional utility gain of the former covers the latter as well. For this reason, we consider only ticket duplication strategy in our analysis.

Game setup. We have a single player game in which a malicious customer applies the ticket duplication strategy. This strategy is defined as duplicating a sequence number among two or more tickets, up to m tickets, where m is the number of beneficiary merchants that are tied to the same escrow. Based on the design of MicroCash's lottery protocol, these duplicated tickets will either all win the lottery or all lose because the lottery draw does not depend on the ticket recipient address (see equation 4.2 in Section 4.4.4). This means that duplication among fewer than m merchants does not reduce the cheating detection probability. Therefore, a rational customer who decides to duplicate a specific ticket will always duplicate it among all m merchants.

We define the utility gain function of any customer as the service value minus the payments made to merchants. We compute the expected value of this function for an honest customer and for a malicious one that applies the ticket duplication strategy. In order to deter cheating, we require the latter to be less than or equal to the former. This is achieved by setting the penalty deposit to be at least equal to the maximum additional expected utility gained by cheating.

As mentioned previously, MicroCash requires any customer to specify in advance the set of merchants who are beneficiaries of its escrow. This is needed to be able to bound the additional utility gain of malicious customers [70]. If this set is not specified, the additional utility cannot be bounded because we would not know the maximum number of duplicated tickets that could be issued.

Table 4.1: Notation.

Symbol	Meaning
$\mathcal{C}$	Honest customer.
$\hat{\mathcal{C}}$	Malicious customer.
$u(\cdot)$	Utility gain function.
τ	tkt_{rate} (number of tickets that can be issued per round), such that $\tau \in \mathbb{N}$.
d	d_{draw} (lottery draw period in rounds), such that $d \in \mathbb{N}$.
r	d_{redeem} (ticket redemption period in rounds), such that $r \in \mathbb{N}$.
y_i	Number of duplicated tickets in round i, such that $0 \leq y_i \leq \tau$.
m	Number of beneficiary merchants, such that $m \in \mathbb{N}$.
p	Lottery winning probability, such that $0 \leq p \leq 1$.
β	Currency value of a wining ticket, such that $\beta \in \mathbb{R}^+$.
υ	l_{esc} (escrow lifetime in rounds), such that $\upsilon \in \mathbb{N}$.

The cheating detection period of duplicated tickets in MicroCash is $d_{draw} + d_{redeem}$ rounds. That is, a malicious customer will be detected when any of the duplicated tickets is presented to the miners[4]. This happens if a duplicated ticket wins the lottery and then claimed by the merchants. Considering the worst case that this claim may take place in the last round of the redeem period, cheating will be detected after $d_{draw} + d_{redeem}$ rounds from a ticket issue time. At that time, the miners break the penalty escrow and the cheating customer leaves the system. Otherwise, if none of the duplicated tickets win, this customer stays and may continue cheating.

Table 4.1 summarizes the notations we use in this section, including shorter abbreviations than those used earlier in this chapter to simplify presentation.

Additional utility gain analysis. We now state and prove a lower bound for $B_{penalty}$ based on the above game setup.

Theorem 2. *For the game and escrow setup described above, issuing invalid or duplicated lottery tickets is less profitable, in expectation, than acting in an honest*

[4]We do not assume that merchants exchange any information between each others about tickets.

way if:

$$B_{\text{penalty}} > (m-1)p\beta\tau\left(\frac{1}{1-(1-p)^{\tau}} + d + r - 1\right) \tag{4.4}$$

Proof. During each round of an escrow lifetime, an honest customer can issue up to τ tickets with unique sequence numbers. Each ticket has an expected value of $p\beta$ coins, which corresponds to the service value a customer obtains for handing out this ticket. We use this service value in computing the utility gain function, and hence, deriving a lower bound for $B_{penalty}$.

In MicroCash, a customer can create an escrow with an υ round lifetime. All tickets issued in a round enter the lottery after d rounds, and all winning tickets will expire after r rounds from the lottery draw time. In other words, for each round $i \in \{1, \ldots, \upsilon\}$, all tickets issued in round i will enter the lottery at round $i + d$ and will expire at round $i + d + r$.

When applying the duplicated ticket issuance strategy, for each round $i \in \{1, \ldots, \upsilon\}$ a malicious customer would decide to duplicate y_i tickets, where $y_i \in \{1, \ldots, \tau\}$. If none of the duplicated tickets win, which happens with probability $(1-p)^{y_i}$, the customer stays in the system and obtains an additional utility gain of $(m-1)p\beta y_i$ over what an honest customer obtains. On the other hand, if any of these tickets wins the lottery at round $i+d$, which happens with probability $1-(1-p)^{y_i}$, the customer will be detected at round $i + d + r$ (the latest). This reduces its additional utility by $B_{penalty}$ as its penalty escrow will be revoked by the miners.

Note that when a duplicated ticket wins, i.e., cheating will be detected, the malicious customer still has r rounds to issue tickets from the time of learning that it will be caught. Therefore, as a rational behavior, this customer will choose to duplicate all tickets in these rounds because it will leave the system either way.

In order to compute the additional utility gain, we need to model the duplication decisions a malicious customer would make at each round of an escrow lifetime. We use a decision process diagram that captures a process evolution over time. Such a diagram contains states indicating the rounds, and transition probabilities between these states computed based on the decisions made at each state. This diagram also

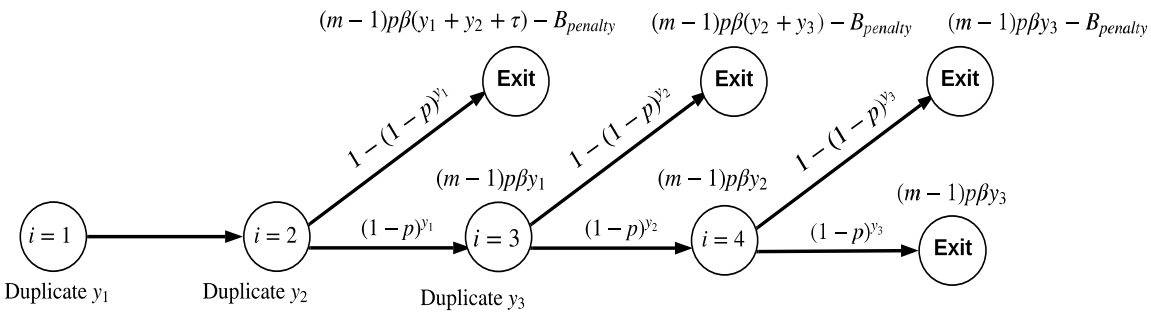


Figure 4.4: Decision process for a 3 round escrow with $d = 2$ rounds and $r = 1$ round. Arrows carry probabilities, decisions are found below the states, and the utility gain is found above the states.

shows the additional utility of being at each state, or round, in the system.

To clarify this concept, we consider a simple case where we have an escrow with 3 round lifetime, $d = 2$ rounds, and $r = 1$ round. The decision process for this setup is captured in Figure 4.4. As shown, a customer issues tickets for rounds 1 and 2 before any lottery draw takes place, where it duplicates y_1 and y_2 tickets, respectively. All tickets issued during round 1 enter the lottery at the beginning of round 3 (or immediately after the end of round 2 as depicted in the figure). If none of these tickets win, the malicious customer obtains an additional utility gain of $(m-1)p\beta y_1$ and proceeds to round 3. For this round, the customer decides to duplicate y_3 tickets. On the other hand, if any of the y_1 tickets wins, the customer knows that it will be detected at the end of round 3 (since $r = 1$). Hence, it decides to duplicate all tickets in round 3 (i.e., $y_3 = \tau$). The total additional utility it obtains in this case, which is displayed above the exit state as this customer will leave the system, is the sum of the utility gain of duplicating y_1, y_2, and y_3 tickets, where $y_3 = \tau$, minus the penalty deposit that will be revoked.

The same analogy is applied to the rest of the rounds, with the exception that at the very last rounds there are less than r rounds to be used at the exit state. In other words, the number of remaining rounds in the escrow lifetime could be less than r, and hence, a customer will duplicate fewer than $r\tau$ tickets, e.g., see the exit state after round $i = 3$ in Figure 4.4.

Instead of analyzing a decision process for an υ round escrow directly, we formulate

the expected utility gain of a malicious customer in a recursive way. That is, we use the expected utility gain in an $\upsilon - 1$ round escrow to compute the expected utility gain in an υ round escrow, and so on. Intuitively, during the first round of an υ round escrow, a malicious customer will decide to duplicate y_1 tickets. If any of these tickets wins at round $1 + d$, cheating will be detected. In this case, and as mentioned earlier, the customer will duplicate all tickets for the next r rounds and will pay the penalty $B_{penalty}$. This means that with probability $1 - (1 - p)^{y_1}$, the utility gain of this customer is $(m - 1)p\beta\big(\sum_{i=1}^{d} y_i + r\tau\big) - B_{penalty}$.

If none of the y_1 tickets wins the lottery, the customer stays in the system. In this case, round 2 was a fresh start for this customer in an $\upsilon - 1$ round escrow. That is, after collecting the utility gain of duplicating y_1 tickets, the customer started fresh in a one round shorter escrow. This means that with probability $(1-p)^{y_1}$, the utility gain of this customer will be $(m-1)p\beta y_1 + \mathbb{E}_{\upsilon-1}[u(\hat{\mathcal{C}})]$, where the second term denotes the expected utility gain of a malicious customer in a $\upsilon - 1$ round escrow.

Based on the above, we can express $\mathbb{E}_k[u(\hat{\mathcal{C}})]$, which is the quantity of interest, as follows:

$$\mathbb{E}_{\upsilon}[u(\hat{\mathcal{C}})] = \big(1 - (1-p)^{y_1}\big)\left((m-1)p\beta\sum_{i=1}^{d} y_i + (m-1)p\beta r\tau - B_{penalty}\right) + (1-p)^{y_1}\left((m-1)p\beta y_1 + \mathbb{E}_{\upsilon-1}[u(\hat{\mathcal{C}})]\right) \quad (4.5)$$

But we have $\mathbb{E}_{\upsilon-1}[u(\hat{\mathcal{C}})] \leq 0$ since the penalty for an $\upsilon - 1$ round escrow has been configured in a way that makes $\mathbb{E}_{\upsilon-1}[u(\hat{\mathcal{C}})] \leq 0$ in order to deter cheating. Hence, and by requiring $\mathbb{E}_{\upsilon}[u(\hat{\mathcal{C}})] \leq 0$ to make cheating unprofitable, we find that:

$$B_{penalty}(y_1, \ldots, y_d) \geq (m-1)p\beta\left(\frac{y_1}{1-(1-p)^{y_1}} + \sum_{i=2}^{d} y_i + r\tau\right) \quad (4.6)$$

For any d and r value, the above quantity is maximized when $y_i = \tau$ for $i \in \{1, \ldots, d\}$.[5] Substituting this in equation 4.6 produces the lower bound stated in the

[5]This is done by considering the terms in $\big(\frac{y_1}{1-(1-p)^{y_1}} + \sum_{i=2}^{d} y_i + r\tau\big)$. For the first term, we used a

theorem, which completes the proof. □

As an example, let's consider an escrow with a 200 round lifetime, $\tau = 1000$ tickets, $p = 0.01$, $\beta = 1$ coin, $m = 5$, $d = 6$, $r = 6$, and $\epsilon = 0.01$. Applying equation 4.3 produces $B_{escrow} = 2,104$ coins (note that the expected total payments, i.e., coverage with probability 0.5, is 2,000 coins). And applying equation 4.4 produces $B_{penalty} \geq 480$ coins.

4.6 Security Analysis

In this section, we analyze the resilience of MicroCash to the threats outlined in Section 4.3. While investigating all possible attack scenarios, using ABC's collusion matrices, we analyzed 126 threat cases to distill 11 cases that MicroCash must address to secure its operation. In defending against these threats, our scheme utilizes cryptographic and financial techniques based on the threat type to be addressed. This is discussed in the following paragraphs.

Escrow overdraft. This threat can be exploited using several strategies, including:

- A customer creates a payment escrow with a balance that does not obey the $1 - \epsilon$ rule, or a penalty escrow with a balance that cannot cover the financial punishment.
- A customer issues more tickets than tkt_{rate} as specified in its escrow setup.
- A customer performs a front running attack in which it withdraws the payment escrow before paying merchants, or withdraws the penalty escrow before paying the financial punishment in the case of cheating.

The first strategy, in which a customer creates payment and penalty escrows with insufficient balance, is neutralized by the escrow setup of MicroCash. When processing an escrow creation transaction, the miners check that the payment escrow

simple iterative program to compute the value of y_1 that maximizes this term for $p \in \{2^{-1}, \ldots, 2^{-10}\}$ and $\tau \leq 10^6$. We found that $y_1 = \tau$ for all these p values. The second term is maximized when y_i is set to its maximum value, which is τ, for all $i \in \{2, \ldots, d\}$.

balance covers all winning tickets with probability $1 - \epsilon$. In addition, they check that the penalty deposit meets the lower bound derived in the previous section. The miners will reject any escrow that do not satisfy these conditions.

The second strategy, i.e., issuing more tickets than can be covered by the escrow, is not possible because lottery tickets are tracked using their sequence number. That is, it is not possible for the client to issue more tickets than tkt_{rate} in any round because each merchant checks the sequence number when verifying any received ticket. Merchants will reject any ticket with a sequence number outside of the current round (modulo one round to deal with desynchronized views of the blockchain).

For the last strategy that covers front running attacks, such attacks are mitigated by the escrow spending mechanism and the lottery protocol implemented in MicroCash. A customer does not control any of the escrows it owns. Instead, fund release is triggered only by the receipt of a valid winning lottery ticket, in case of the payment escrow, or a valid proof-of-cheating, in case of the penalty escrow. Honest miners will enforce these rules in the system.

As mentioned previously, MicroCash burns penalty escrow funds when a proof-of-cheating is received, rather than providing them to another party, for the following reason. If the funds are provided to another party, that party may have colluded with the customer to receive those funds. This would reduce the effective penalty of cheating imposed on the customer. To mitigate this, miners burn the funds in a penalty escrow (this is after paying out the merchants in case of ticket duplication).

Duplicate Ticket Issuance. MicroCash addresses this attack financially through a detect-and-punish approach. Any party that detects two or more tickets carrying identical sequence numbers, and issued using the same escrow, can produce a proof-of-cheating against the issuing customer. Miners publish this proof on the blockchain, which burns the customer's penalty escrow as a punishment. Setting the penalty deposit as described in Section 4.5 makes cheating unprofitable, which deters rational customers from attempting this malicious strategy.

Invalid payments. To pursue this attack, a customer may issue tickets with invalid format or invalid field values knowing that these tickets cannot be claimed later if they win. An invalid ticket is a ticket that uses an invalid escrow (e.g., a broken one), or has an invalid $(t_{issue}, tseq)$ tuple, or destined to a merchant that is not one of the escrow beneficiaries. These events can be detected instantly because merchants validate all lottery tickets before accepting them. As mentioned previously, an invalid ticket is dropped unless it has an invalid $(t_{issue}, tseq)$ tuple or a merchant that is not a member of the set of beneficiary merchants. Such a ticket can be used as a proof-of-cheating that costs the customer its penalty escrow, which exceeds the profit of cheating as explained in the previous section. This deters rational customers from issuing invalid tickets.

Unused-escrow withholding. This threat is mitigated by the expiration rule of lottery tickets and MicroCash's escrow refund policy. Each ticket has a redemption period after which it expires. Hence, a merchant who tries to put an escrow on hold by delaying claiming its winning ticket claim will lose its payment. Furthermore, when all tickets tied to an escrow expire, i.e., when t_{refund} is approached, the miners will allow the owner-customer to spend the remaining funds in its escrow. This prevents locking unused escrow funds indefinitely on the blockchain.

DoS. As DoS is a large threat category, in this work we consider the cases that are different and unique to the design of MicroCash. These include preventing customers from creating escrows, preventing merchants from claiming their winning lottery tickets, or control relaying blocks and transactions based on their content. Such situations may happen when miners disregard specific transactions or blocks, or when an attacker controls the network links and drops specific transactions or blocks.

The case of miners disregarding specific transactions/blocks may take place when an attacker controls a substantial portion of the mining power. This may work even under the assumption that the majority of the mining power is honest. That is, an attacker with $< 50\%$ of the mining power may still be able to perform harmful

attacks, e.g., selfish mining [116]. To protect against selective relaying, techniques that allow propagating blocks and transactions without disclosing their content can be employed, e.g., BloXroute [18].

The case of controlling the network links, which represents attacks against the network availability, is a potential problem in any distributed system in general. Deploying mechanisms to enhance network connectivity, e.g., participants maintain connections with large number of miners, may reduce the effect of this attack. Such mechanisms are independent of the design of MicroCash, it is up to the parties themselves to adopt suitable solutions.

Lottery manipulation. This threat covers all strategies that could be used to manipulate the lottery draw, including:

- An attacker, who could be any party inside or outside the system, tries to influence the hash used in a lottery draw in order to make specific tickets win, or lose.
- A malicious customer may issue winning lottery tickets, to itself or to malicious colluding merchants, to drain the escrow before other merchants can claim their winning tickets.
- A malicious customer deliberately issues losing tickets to merchants to avoid paying them.
- An attacker, insider or outsider, tries to issue lottery tickets to herself or others.

In the first strategy, an attacker tries to influence the lottery by controlling the hash used in the lottery draw. This can be done by either manipulating the ticket fields that impact the lottery, where the issuing customer may tweak a ticket in order to produce a favorable hash. Or by controlling the hash of the block mined at time t_{draw}, where a miner may forgo any block that does not produce a favorable lottery outcome (i.e., a favorable VDF value). Or even this malicious miner may publish an incorrect VDF value in order to bias the lottery draw outcome.

All these malicious behaviors are mitigated by the lottery protocol design. The

ticket fields that impact the lottery draw include only the ones that appear in the escrow state, which cannot be tweaked by the issuing customer, or any other party. For discarding unfavorable blocks, recall that the lottery draw is based on the VDF value of the block at index t_{draw}. This value needs time to be computed. Therefore, a miner who chooses to perform this computation, and then announce a favorable block, will have a low chance of succeeding in publishing this block on the blockchain. This is because other miners will announce their blocks immediately once they mine them. The chances are higher for any of these blocks to be adopted by the network, and hence, to be used in the lottery draw (under the assumption that the majority of the mining power is honest). Furthermore, publishing an invalid VDF value will not succeed. This is because honest miners, who already computed this value during mining their blocks, will reject a newly received block with a VDF value that does not agree with the one they computed. Consequently, such a malicious miner will not only fail in biasing the lottery, but also will lose the mining rewards.

In the second strategy, a malicious customer tries to withdraw an escrow indirectly by issuing winning tickets to itself, and claim these tickets before merchants can claim their payments. To do so, a customer may wait until the block at index t_{draw} is mined and then print winning tickets to itself. This techniques is mitigated by the lottery protocol design. After observing the block at time t_{draw}, the customer cannot do anything to produce winning tickets other than checking which sequence numbers are winning (assuming it did not use these sequence numbers to pay any merchant earlier). This customer cannot manipulate the ticket fields to make a losing $tseq$ wins as mentioned previously. Also, it cannot issue losing tickets using these sequence numbers to merchants because it is too late (the issue time of these tickets was d_{draw} rounds earlier), and hence, merchants will not accept these losing tickets. On the other hand, a customer who saves some sequence numbers and use them to issue tickets to itself, if they win, will not affect the payments of other merchants. This is because these sequence numbers are covered by the payment escrow balance (i.e., they are legitimate tickets covered by the escrow).

The third strategy, where a customer deliberately hand merchants losing tickets,

succeeds with a very low probability. In order to determine which ticket will lose the lottery, this customer needs to predict the hash of the future block mined at time t_{draw} in order to compute the VDF value of this block, or even guess the VDF value itself. Since hashes are modeled as random oracles, predicting such values succeeds with negligible probability.

Lastly, the fourth strategy, in which an attacker tries to issue tickets tied to escrows she does not own, will not succeed due to the use of secure cryptographic signatures. MicroCash requires any customer to sign all lottery tickets it issues, which means that to issue a valid ticket an attacker needs to forge the customer's signature. By using a secure digital signature scheme, and unless an attacker steals the signing key, such an attack will fail with all but negligible probability.

4.7 Performance Evaluation

In order to understand the performance benefit of concurrent probabilistic micropayments, this section evaluates the computation, bandwidth, and payment setup costs of MicroCash. To do this, we conduct empirical experiments to answer the following questions:

- How fast can customers, merchants, and miners process lottery tickets?
- What is the bandwidth cost of exchanging these tickets?
- What is the size of escrows on the blockchain?
- How do the schemes compare using workload numbers derived from real world scenarios?

To put our results in context, we compare our scheme with MICROPAY [111]. The rest of this section describes our methodology and discusses the significance of the obtained results.

4.7.1 Methodology

To establish our benchmarks, we implemented the functions used for generating tickets, verifying these ticket, and performing a lottery draw. For MicroCash, we followed

the design introduced in this chapter. For MICROPAY, we tested its fully decentralized version, called MICROPAY1, with its non-interactive lottery protocol [111]. This protocol requires a merchant to publish the description of a verifiable unpredictable function to perform the lottery. For this function, we used the verifiable random function (VRF) construction introduced by Goldberg et al. [80] with its implementation over the NIST P-256 curve [39].

Two cryptographic primitives affect the implementation of both MicroCash and MICROPAY, namely, hash functions and digital signatures. For hashing, we used SHA256. For digital signatures, elliptic curve based constructions are usually used, e.g., ECDSA and EdDSA, with various options for the underlying elliptic curve. We chose to test the most common schemes, including ECDSA with secp256k1 curve (used in Bitcoin and most cryptocurrencies), ECDSA with P-256 curve (widely used and recommended by NIST), and EdDSA with Ed25519 curve [65] (received a great interest recently due to its security and efficiency, where several major cryptocurrencies are using this scheme [34, 42] or preparing to switch to it [25, 41]).

We computed the performance metrics of interest as follows. We measured the computation cost as the rate at which customers, merchants, and miners can process lottery tickets. And we measured the bandwidth overhead by reporting on the size of tickets (in bytes) when exchanged between customers and merchants, and the size of winning tickets claimed by merchants through the miners. To evaluate the effect of micropayment concurrency, we computed the number of escrows a customer would need to support the ticket issue rate in each of the tested schemes. Furthermore, we studied two real life applications and computed the overhead of processing micropayments using workload numbers derived from these applications.

Lastly, our experiments were implemented in C on an Intel Core i7-4600U CPU @ 2.1 GHz, with 4 MB cache and 8 GB RAM, where each of the payment processing functions was called 10^6 times.

Table 4.2: Ticket processing rate (ticket / sec).

Scheme	**ECDSA (secp256k1)**	**ECDSA (P-256)**	**EdDSA (Ed25519)**
MICROPAY			
Customer	1891	32606	20884
Merchant	1353	2530	2509
Miner	1365	2591	2565
MicroCash			
Customer	1890	32978	20879
Merchant	2266	10463	7825
Miner	2266	10463	7825

4.7.2 Microbenchmark Results

Lottery ticket processing rate

We start by quantifying the computation cost of processing micropayments in both schemes. This is done by measuring the rate at which a customer can generate lottery tickets, the rate at which a merchant can process these tickets, which involves both validating a ticket and running the lottery[6], and the rate at which miners validate claimed tickets. The obtained results are found in Table 4.2.

As the table shows, customers in both schemes generate tickets at comparable rates. This is because the operations performed when issuing a ticket are almost identical in MicroCash and MICROPAY, with the exception that a ticket size in MICROPAY is larger. The heaviest operation in this process is signing a ticket. For this reason, the generation rates improve by using an efficient digital signature scheme, where ECDSA over P-256 is the most efficient scheme. Such an improvement is significant, as reported in Table 4.2, where replacing ECDSA (secp256k1) with ECDSA (P-256) and EdDSA (Ed25519) boosts the performance by around 17x and 11x, respectively.

[6]Although a merchant in MicroCash runs the lottery several rounds after validating a ticket, we report the cost of these two operations together. This is because such cost is the overhead per ticket incurred on the merchant side.

The trend is different for the rates of merchants and miners. As shown in Table 4.2, these parties in MicroCash are 1.67x, 4.1x, and 3.1x faster than in MICROPAY for the three digital signature schemes. This is because miners and merchants run the lottery over the received ticket. In MicroCash, this process involves a single, lightweight hash operation. On the other hand, the lottery in MICROPAY requires evaluating a computationally-heavy VRF.

Furthermore, merchants and miners in MicroCash benefit more from the efficiency of the used digital signature scheme. This is because the heaviest operation these parties perform when processing a ticket in MicroCash is verifying a customer's signature. However, in MICROPAY the bottleneck is evaluating a VRF and producing a proof that its output is correct on the merchant side, and verifying this proof on the miner side. Thus, verifying the signature over a ticket comprise a small portion of the computation needed to verify a ticket in MICROPAY. For this reason, optimizing the digital signature scheme has a larger effect in MicroCash. As shown in the table, MICROPAY obtains only around 1.9x improvement when replacing ECDSA (secp256k1) with any of the other two schemes. In contrast, MicroCash achieves around 4.6x and 3.45x improvement when replacing ECDSA (secp256k1) with ECDSA (P-256) or EdDSA (Ed25519), respectively.

Key Takeaway: Compared to MICROPAY, MicroCash reduces the computational load on merchants and miners by a factor of 1.67-4.1x.

Lottery ticket bandwidth cost

In terms of bandwidth overhead, MicroCash incurs less overhead than MICROPAY because its lottery tickets are of smaller size. A ticket sent from a customer to a merchant is 142 bytes in MicroCash, while it is 274 bytes in MICROPAY[7]. A winning ticket sent from merchants to miners is also 142 bytes in MicroCash, while it is 355 bytes for MICROPAY becuase this ticket must be accompanied with an additional

[7]For MICROPAY each lottery ticket contains a sequence number, the public keys of the customer, the recipient merchant, and the escrow, beside a description of a VRF that includes the group order, generator, and the VRF public key.

field, a NIZK proof[8]. This means that MicroCash incurs only 52% of the bandwidth cost of MICROPAY between customers and merchants, and only 40% of the cost between merchants and miners.

To put these numbers in context, we report on the transaction sizes in Bitcoin. The average size is around 500 byte, where a transaction with one or two inputs is about 250 bytes [44]. Adding a winning ticket as one of these inputs produces a claim transaction with a size of 392 bytes in MicroCash, which is less than the average Bitcoin transaction size. While in MICROPAY, the size of a claim transaction will be 605 bytes, exceeding the average size.

Key Takeaway: The use of efficient lottery protocol reduces not only the computation cost in MicroCash, but also the bandwidth cost of exchanging lottery tickets and the amount of data to be logged on the blockchain.

Size of escrow data on the blockchain

One major difference between concurrent and sequential micropayment schemes is that concurrent schemes allow multiple winning tickets per escrow. This means that the size of escrow data on the blockchain will vary based upon the number of winning tickets. So, an escrow with 100 winning tickets in MicroCash will prevent 99 escrow transactions that would have occurred with MICROPAY.

Additionally, in sequential micropayments, as in MICROPAY, the ticket issuance rate also makes a difference in terms of the number of escrows needed. This is because a new ticket cannot be issued using the same escrow until it is confirmed that the prior ticket did not win, which requires the customer to wait for the merchant to announce the lottery result. To issue another ticket during this waiting time, a customer needs a different escrow. Hence, a customer might slow down just because it does not have enough escrows to allow this rate. Furthermore, even if the customer is willing to create larger number of escrows, this dramatically increases the payment setup cost. Each of these escrows requires an individual escrow creation transaction, which in

[8]All these sizes are few bytes less when EdDSA is used because the signature is shorter than what ECDSA produces.

turn requires paying a transaction fee and logging on the blockchain.

For example, to support the ticket issue rates reported in Table 4.2, a MICROPAY customer would need large number of escrows. The exact number of escrows needed depends on the network latency and a merchant's ticket processing rate. Using the average US RTT of 31 ms [6], and the processing time of the tickets reported in the table, in the best case an escrow in MICROPAY can be used to issue 32 tickets per second (this is in case none of these ticket win or only the last one wins). Therefore, a customer in MICROPAY would need 60, 1019, or 653 escrows to support the generation rates for signature schemes ECDSA (secp256k1), ECDSA (P-256), or EdDSA (Ed25519), respectively, as found in Table 4.2. On the other hand, a customer in MicroCash would need only *one* escrow with proper balance to pay at any given ticket rate.

Key Takeaway: Supporting micropayment concurrency dramatically reduces the amount of escrow data on the blockchain.

4.7.3 Micropayments in Real World Applications

To ground our results in real world numbers, we examined two applications; online gaming and peer-assisted content delivery networks (CDNs). We computed the overhead of processing micropyaments with parameter values derived based on the service price and workload in these applications. This is done for three cases: Bitcoin without employing any micropayment scheme, Bitcoin with MICROPAY, and Bitcoin with MicroCash.

Setup

We begin with the online gaming. For this application we used data from the popular game Minecraft [33] as an example. The average mid tier cost of playing this game for 8 players is around $12 per month [26]. We considered 1000 players distributed among 125 servers. This means that the service price is $0.034722 per minute (or $0.000579 per second) for the 1000 players.

For the peer-assisted CDN application, we considered a content publisher who hires peers as caches to distribute the content for its clients. Suppose that a publisher wants to serve content at roughly 1Gpbs. Such a service costs around $17,312 monthly in the US [5], and hence, on average, the service cost per second is $0.006679. The publisher will provide a lottery ticket to a cache for each 1MB data chunk it serves. Thus, to support a rate of 1Gbps, the publisher will issue 128 tickets per second.

With these combined values, it is possible to compute the lottery winning probability p and the currency value of a winning ticket β. The former is done by determining the total transaction fee to be paid for the miners (per second), and then compute p in a way that makes the fees paid when claiming winning tickets (per second) does not exceed this value. We consider the fee to be 2% of the service cost paid per second [7] (at a minimum)[9]. For the fee of redeeming a winning ticket, we use the median transaction fee in Bitcoin, which is around $0.068 as of late January 2019 [10].

Based on that, the fees per second are equal to the expected number of winning tickets per second multiplied by the transaction fee that miners charge. So in Minecraft, p can be computed as $p = \frac{(0.02)(0.000579)}{(16.67)(0.068)} = 0.00001$, where 16.67 is the number of tickets issued by all players per second (the 1000 players issue 1000 tickets per minute). Similarly, a publisher in our CDN example can compute p as $p = \frac{(0.02)(0.006679)}{(128)(0.068)} = 0.000015$.

As for computing β, it can be estimated by dividing the service cost by the number of winning tickets (both per second). In Minecraft, this produces $\beta = \$3.472$, and it is $3.4 in the CDN application.

For the escrow setup, in Minecraft, we assume that each player creates one escrow per month. For the CDN application, we consider a publisher creating one escrow per day. In addition to that, in MICROPAY a new escrow must be created after each winning ticket.

It should be noted that in both the gaming and CDN examples, we account for

[9]In both example, we assume that the players and the publisher will pay at the same price offered by a gaming or CDN company.

Table 4.3: Micropayment overhead in online gaming (a round is 10 min).

Metric	**Bitcoin**	**MICROPAY**	**MicroCash**
Winning tickets / sec	N/A	0.000167	0.000167
Escrows / sec	N/A	0.000552	0.000386
Transactions /sec	16.67	0.000719	0.000552
Transaction fees / round	$680	$0.02934	$0.02254
Bandwidth between players and miners	3,333 bps	1.105 bps	1.0093 bps
Bandwidth between players and servers	N/A	36,533 bps	18,933 bps
Bandwidth between servers and miners	N/A	0.807 bps	0.523 bps
Delta blockchain size / round	2.38 MB	0.000137 MB	0.00011 MB

the cost of operating the service only. For example, there is no money given to fund Minecraft's development team (which was presumably supported by the initial purchase of the game) or to produce the video content that was served by the CDN. In practice, operational costs are minimal compared to the development cost which can run in the hundreds of millions of dollars [47].

We used this setup to compute overhead of micropayment processing for both applications as shown in what follows.

Online Gaming Application

We used the configuration parameters outlined earlier to compute the cost of micropyament processing in this application. The results are found in Table 4.3.

We start with the number of transactions the miners process. In Bitcoin, all micropayments will be processed as individual transactions. In contrast, with MICROPAY and MicroCash, only winning tickets and escrow creation transactions will be sent to the miners. Given that MICROPAY is a sequential scheme, where every time a ticket wins the escrow breaks, the number of escrows per round equals to the expected number of winning tickets per round. While in MicroCash, a player may use one escrow for the duration of their subscription (a month, in our case). Conse-

quently, and as the table shows, MicroCash generates 25% fewer transactions (both escrow and winning ticket redemption).

The number of transactions affects the amount of fees miner charge for processing. As the table shows, processing micropayments individually is expensive, which is $680 per round (a transaction costs around $0.068 as mentioned previously). However, in MicroCash and MICROPAY, such a fee is paid only when claiming winning tickets or creating escrows. Furthermore, due to the reduced number of escrows, MicroCash incurs the least fees, which is around 75% of MICROPAY's fees.

For the bandwidth cost, in Bitcoin players will send all micropayments directly to the miners. They do not send anything to the servers nor servers send anything to the miners. While in MICROPAY and MicroCash, all lottery tickets are exchanged locally between players and servers. Only escrows and winning tickets will be sent to the miners. Based on the average size of a Bitcoin transaction (around 250 bytes as mentioned earlier), the size of an escrow transaction in MICROPAY is around 250 bytes, while it is around 327 bytes in MicroCash. This is because our scheme adds additional fields to store the payment setup parameters as described in Section 4.4.2. For the size of a claim transaction, beside the transaction average size, we add the size of a winning ticket (142 bytes in case of MicroCash and 355 bytes in case of MICROPAY). As shown in Table 4.3, the bandwidth cost between players/servers and the miners in Bitcoin is more than 3000x the cost incurred in MICROPAY or MicroCash. This shows the great benefit of processing payments locally using a micropayment scheme.

The bandwidth cost of the miners can be used to quantify the increase in the blockchain size per round, where all transactions sent to the miners are logged on the blockchain. As the table shows, logging all micropayments is prohibitive, it requires more than 2 MB per round. In Bitcoin, only one block of size 1MB can be published per round, meaning that paying at this relatively slow rate cannot be supported. On the other hand, this overhead is reduced to less than 0.00015 MB in MICROPAY and MicroCash.

Table 4.4: Micropayment overhead in Peer-assisted CDNs (a round is 10 min).

Metric	**Bitcoin**	**MICROPAY**	**MicroCash**
Winning tickets / sec	N/A	0.001964	0.001964
Escrows / sec	N/A	0.001976	0.00001157
Transactions /sec	128	0.00394	0.001976
Transaction fees / round	$5,222	$0.160769	$0.08062
Bandwidth between publisher and miners	256,000 bps	3.95 bps	0.165 bps
Bandwidth between publisher and caches	N/A	280,576 bps	145,408 bps
Bandwidth between caches and miners	N/A	9.508 bps	6.16 bps
Delta blockchain size / round	18.31 MB	0.000963 MB	0.000452 MB

Peer-assisted CDN Application

The results for this application are found in Table 4.4. There is a more dramatic benefit to employing a micropayment scheme for serving CDN traffic than the gaming application because the workload is larger.

As the table shows, in plain Bitcoin, 128 transactions per second are processed by the miners, which is the number of data chunks caches serve per second. Whereas this number drops to fractions in both MICROPAY and MicroCash, with 50% reduction in MicroCash as compared to MICROPAY. This is reflected on both the transaction fees and the blockchain size. Processing micropayments individually costs more than $5,000 per round, while these fees drop to cents when a micropayment scheme is employed. It also requires logging more than 18 MB per round on the blockchain. On the other hand, this overhead is reduced to around 0.001 MB in MICROPAY and 0.0005 in MicroCash. This shows the great advantage of employing a micropayment scheme for heavy loaded applications, and the benefit of supporting micropayment concurrency (where MicroCash reduced the additional blockchain size and the total fees by around 50%).

For the bandwidth cost between participants, in plain Bitcoin the miners have, at least, 19,000x the cost when a micropayment scheme is employed. Moreover,

MicroCash incurs almost no bandwidth cost between the publisher and the miners. This is despite the fact that in this application, an escrow creation transaction in MicroCash is larger than the one needed in MICROPAY (such a transaction is around 1,783 byte in MicroCash, where we consider 45 beneficiary caches to support the rate of 1Gbps[10]). Such a minimal cost is due to payment concurrency, where MicroCash allows creating a one long-lifetime escrow instead of large number of escrows as in MICROPAY. Even for ticket exchange, MicroCash incurs a lower cost although both schemes have the same number of tickets. This is because a ticket (on-chain or off-chain) in MicroCash is substantially smaller than in MICROPAY.

Key Takeaways: Micropayments are absolutely critical to be able to process small transactions in modern applications. MicroCash is cost efficient enough even to be used in gaming and CDN apps where content production is not part of the cost passed along to users. Concurrent use of a single escrow decreases the total data added to the blockchain by roughly half.

4.8 Conclusion

In this chapter, we introduced MicroCash, the first decentralized probabilistic framework that supports concurrent micropayments. MicroCash enables processing small monetary transactions with low overhead. This is done by introducing an escrow setup and ticket tracking mechanism that permit a customer to rapidly issue tickets in parallel using only one escrow. Moreover, MicroCash is cost effective, where it features a non-interactive lottery protocol for micropayment aggregation that is based solely on hashing. When compared to the sequential scheme MICROPAY, MicroCash has a significant performance increase while substantially decreasing the bandwidth cost and the amount of data stored on the blockchain.

In the next chapter, we present the main system, CacheCash, that builds upon all modules introduced in this and previous chapters to provide a secure and efficient fully distributed CDN service.

[10]We use the average upload speed in the US [8], which is 22.79 Mbps. Hence, to serve 1Gbps, a publisher needs 45 caches.

Chapter 5

Putting it Together: Tapping New Security and Efficiency Strategies to Build CacheCash

This chapter is based on joint work with Allison Bishop and Justin Cappos [55].

5.1 Overview

In this chapter, we present the core part of this dissertation, CacheCash, the first decentralized CDN system designed to address many of the limitations of existing solutions discussed in Section 1.2. We show how CacheCash combines the modules introduced in the previous chapters toward supporting a secure, efficient, and cost effective content delivery service.

As mentioned previously, CacheCash creates a distributed bandwidth market that enables publishers to hire caches on an as-needed basis. This market allows end users to organically set up new caches and serve content, without introducing any trust assumptions or pre-authentication requirements, and uses a cryptocurrency system to reward these caches in a distributed way. Thus, it permits caches to dynamically come and go as demand shifts, without constraining either caches or publishers with long-term business commitments.

In order to support such a flexible service model, CacheCash introduces a novel service-payment exchange protocol that reduces the risks of dealing with distrusted parties in P2P networks. This protocol utilizes gradual content disclosure and partial payment collection to encourage the honest collaborative work between peers, and to stabilize the work relation between a cache and a publisher. In detail, CacheCash devises a unique way to reward caches by modifying MicroCash to support two ticket

types instead of one, and to allow variable winning ticket currency value instead of a fixed payment amount. Furthermore, CacheCash utilizes CAPnet's accountability puzzle in finalizing a cache payment, where the full service reward cannot be collected unless the client solves a puzzle correctly over the requested content. All these design decisions are guided by a thorough threat model built using the ABC framework.

For serving each data chunk of the requested content, a cache receives two lottery tickets. It collects the first ticket after delivering a double encrypted chunk to the client, in exchange for the outer layer decryption key. While this cache collects the second ticket after the client solves a cache accountability puzzle over the retrieved chunks proving the correctness of the data, and confirming that caches performed the promised work. CacheCash ties the currency value of both tickets together, where collecting more winning tickets of the second type increases the currency value of a winning ticket of the first type. This motivates caches to work faithfully and collect both tickets. It also encourages any cache , to continue working with the same publisher to increase the accumulative number of winning tickets from the second type, and thus, increasing its profits.

The security of CacheCash relies on employing both cryptographic approaches and rational financial incentives. The latter is based on a detailed game theoretic analysis that determines how to make honesty the most profitable option for all entities in the system. Furthermore, this analysis shows how to configure the payment function used to compute the currency value of winning tickets in a way that captures the aforementioned goals.

Equally important for its use in practical applications, CacheCash does not sacrifice efficiency for enhanced security. Its tools are built on computationally-light primitives and operations, and it utilizes several techniques to boost performance. Publishers achieve a high degree of efficiency by processing clients' requests in batches, and by offloading, as one response, all the information a client needs to complete a service request. Similarly, caches and clients implement a lightweight data service protocol. Hence, caches are able to serve large number of clients concurrently, and clients are able to retrieve large amounts of data with low resource consumption.

In evaluating the performance of the system, our benchmark results show that a modest client machine can retrieve content at rate of around 122 Mbps, which is enough to watch dozens 1080p quality videos simultaneously. A low-end cache can serve clients at a bitrate of more than 20 Gbps (capped by the upload bandwidth speed such a cache has). Even a low-end publisher machine can serve content requests at a rate sufficient for 315,780 simultaneous views of the same 1080p video. All of these rates are supported with very low bandwidth overhead, less than 0.1%.

5.2 Threat Model

As we have stated previously, working in a monetary-incentivized environment that allows anyone to join creates several security risks. Participants will likely try to maximize their monetary gains, even by using a malicious strategy. This section presents a threat model for CacheCash that guided its design.

We used ABC (Chapter 2) to build a threat model for CacheCash[1], under the same set of assumptions found in Section 4.3. In this process, we identified the assets to be protected in cryptocurrency-based distributed CDNs. These assets include the service and the service rewards or payments[2]. By analyzing the security requirements of these assets, we produced a set of broad threat categories, which includes the following:

- **Service corruption:** This threat takes place when a client, who faithfully obtains data chunks, retrieves the correct amount of data, but this data (or part of it) has been altered in a way that results in retrieving corrupted content.
- **Service theft:** In this attack, honest caches that serve a client correctly do not get their payments. The attacker could be a publisher (possibly colluding with a client) who may manipulate payments to avoid paying caches or a client that pretends it does not obtain correct service.

[1]A detailed version of the threat model is available online [31]

[2]Note that assets from the micropayment scheme are secured under the design of MicroCash, and those for the currency exchange medium are part of Bitcoin threat model as we assume the use of a modified version of Bitcoin network protocol.

- **Cache accounting attacks:** In this attack, caches receive payments without doing all the required work. It may happen when a client (who is not interested in the data) colludes with caches to help them collect currency without delivering any service.
- **Denial of service (DoS):** This is a large threat category that includes all attack forms that interrupt system operation and make it unavailable to legitimate users. We focus on attacks related to the content distribution service, such as monitoring the network and dropping all service requests and payments, ignoring all content requests initiated by a specific client, or sending a huge number of requests to publishers or caches to waste their resources.

As mentioned in Section 1.3, CacheCash does not address preserving digital rights of publishers nor client privacy. Such an issues are part of our future work.

5.3 CacheCash Design

The design of CacheCash can be described as a series of interrelated processes, which collectively create an efficient and cost-effective CDN service that achieves the goal outlined in Section 1.3. This design combines cache accountability puzzles, decentralized probabilistic micropayments, and rational financial incentives to secure the service-payment exchange process. In addition, it taps various performance optimization techniques to reduce system overhead. This section presents an overview of the system followed by a detailed description of its individual operations.

5.3.1 Work Environment Model

CacheCash's network, as depicted in Figure 5.1, is composed of four types of participants: content publishers, their clients, caches, and miners. The roles and relations between publisher, clients, and caches are the same as defined in CAPnet's network model (see Section 3.3.1). The only difference is that publishers pay caches cryptocurrency tokens in exchange for the content distribution service. The miners perform the tasks needed in any cryptocurrency system (i.e., mining, transaction processing,

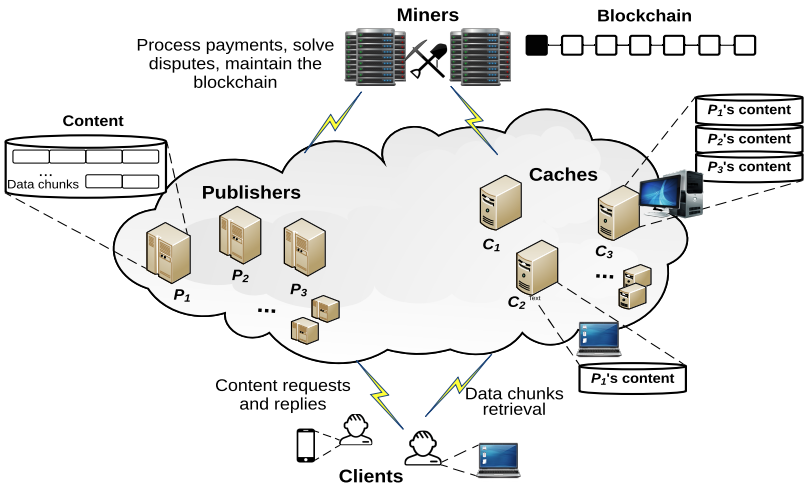


Figure 5.1: CacheCash network model.

and maintaining the blockchain). In addition, these miners implement the probabilistic micropayment scheme CacheCash adopts, and punish cheaters financially based on the protocol design. In implementing the underlying cryptucrrency system, we assume the use of a modified version of Bitcoin's network protocol. Such a version includes all the additional transaction types and processing logic that CacheCash requires.

5.3.2 CacheCash in a Nutshell

At a basic level, CacheCash operates as a set of three sequential actions: service setup, content distribution, and payment for completed service. Below, we offer a walk-through of these operations.

The service setup in CacheCash begins with the assembling of a set of caches. A publisher advertises for caches to handle distribution of a particular content. Any cache that agrees to participate retrieves a copy of the content from the publisher and verifies its correctness. Furthermore, and similar to CAPnet setup (see Section 3.3.3), this cache agrees with the publisher on a shared secret that is used to generate a unique key for each session this cache will handle.

When the network reaches a feasible size, the publisher creates an escrow on the blockchain to secure funds to pay the caches. This is done in a similar way to Micro-

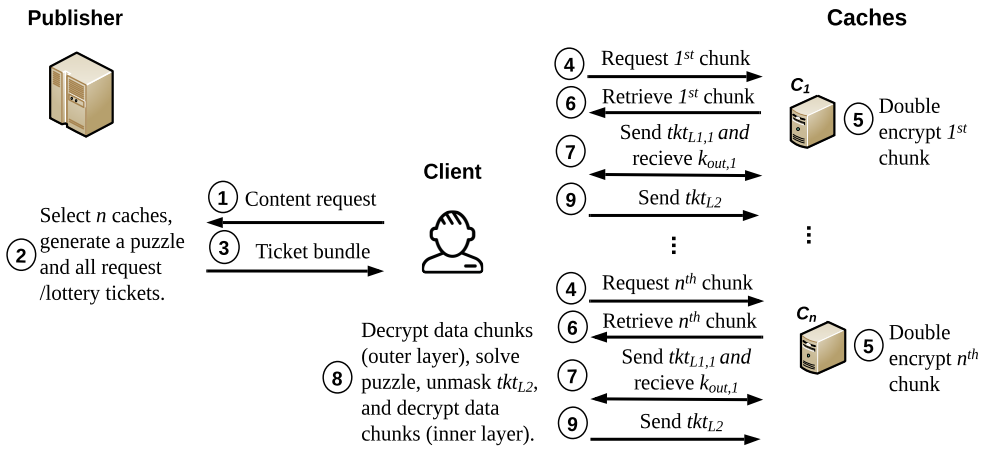


Figure 5.2: Content retrieval process ($tkt_{L1,j}$ is the j^{th} first lottery ticket, tkt_{L2} is the second lottery ticket, and $k_{out,j}$ is the j^{th} outer layer encryption key).

Cash (see Section 4.4.2). However, CacheCash adds more parameters to the escrow creation transaction, e.g., payment function configuration. Furthermore, it uses different bounds for the payment and the penalty escrow balances to accommodate this function and the two lottery ticket types that CacheCash adopts for rewarding caches.

Distribution of content, which is the second action in CacheCash's design, can begin as soon as the escrow is confirmed on the blockchain. This action proceeds in service sessions, as illustrated in Figure 5.2. A session starts with a client sending a content request to the publisher (**Step 1**), after which the publisher selects a set of n caches, each of which will deliver a specific data chunk of the content. In order to allow the client to proceed on its own, the publisher replies to the content request with a ticket bundle (**Step 2**). This bundle contains request tickets that a client will use to contact caches, a cache accountability puzzle, and a set of lottery tickets to pay these caches. The publisher sends the bundle back to the client (**Step 3**).

CacheCash reduces the monetary losses that can be caused by malicious actors in the service-payment exchange process by dividing a session payment into two parts. In detail, for any service request, the publisher generates two lottery tickets for each cache, denoted as tkt_{L1} and tkt_{L2}. A cache receives tkt_{L1} upon delivery of the data chunk to the client, but does not receive tkt_{L2} until the delivered chunk is proved valid. By having the publisher mask tkt_{L2} with the accountability puzzle solution,

caches cannot obtain this ticket unless the client solves the puzzle correctly. This confirms to the publisher that caches have earned their payments.

Retrieving the content starts by sending each request ticket to its designated cache (**Step 4**). Before sending the requested chunk, a cache double encrypts it by adding both inner and outer layers of encryption (**Step 5**). The inner encryption layer is required by the design of CAPnet (see Chapter 3). That is, encrypting a chunk using a unique session key causes the ciphertext to be different. This in turn makes the puzzle unique even if the raw content is the same. The outer layer encryption, with the encryption key known only to the cache, serves two purposes. First, it forces a client to retrieve the whole chunk before being able to start solving the puzzle (same role of the completion mask introduced in CAPnet). And second, it forces this client to pay a cache tkt_{L1} in exchange for the encryption key (**Steps 6** and **7**).

After decrypting the outer layer of all chunks (**Step 8**), the client can start solving the accountability puzzle. It uses the solution to unmask tkt_{L2} and reveal the inner layer encryption keys. With these keys, the client can recover the raw content, after which it forwards tkt_{L2} to all caches (**step 9**), which completes the service session.

The last action in CacheCash's design is paying the caches out of the publisher's escrow. This follows the basic operation of MicroCash (Section 4.4), where a cache uses its winning tickets to claim payments. However, CacheCash changes how the currency value of a winning ticket is computed. To motivate caches to work faithfully in a service session, and hence, collect both ticket types, the currency value of a winning tkt_{L1} is tied to the accumulated number of winning tkt_{L2} a cache owns (such that all these tickets are tied to the same escrow). This payment relation also encourages a cache to continue working with the same publisher for the long run, and thus increasing the counter value of winning tkt_{L2}, instead of switching frequently between publishers. In turn, by reducing the amount of turnover, publishers can gain a stability in the service they obtain.

As in MicroCash, once the escrow lifetime is over, no more tickets can be issued to the caches. At that time, the owner-publisher can spend the remaining funds (if any). This publisher needs to create a new escrow if it wants to continue using the

CDN service.

5.3.3 Service Setup

The service setup action defines how a cache or a publisher can join the system in order to start offering/receiving the CDN service. While the setup process for caches is simple, involving retrieving a copy of the content and establishing a shared secret with the publisher, it is a bit more involved on the publisher's end. It requires advertising the need for caches, and then creating and managing an escrow that determines how and when caches can be paid. For this reason, this section focuses on the publisher's service setup, which includes service advertising, and escrow creation and management.

Advertising to Recruit Caches

A publisher with content to distribute needs a set of caches to serve its customers. Assembling that network typically starts with a recruitment initiative. The publisher announces the set of service specifications it is looking for, such as bandwidth speed, storage capacity, geographic location, etc., possibly using the publisher website or any other suitable media. Any cache that meets the announced requirements can contact the publisher to join its network.

Escrow Creation

Once a reasonable number of caches join its network, the publisher needs to create two escrows, payment and penalty, before any service session can begin. Similar to before, this is done using one escrow creation transaction that specifies the amount of funds to be locked under B_{escrow} and $B_{penalty}$, and the set of parameters that determine the service price paid to caches. The publisher sets the values of these parameters, possibly after negotiating with caches, which include the following:

- The set of beneficiary caches that can be paid using the escrow, where the size of this set is denoted as N.

- The size of the cache set, i.e., n, that would be selected to fulfill any content request.
- A signed hash of the content to allow a cache to verify its correctness[3].
- The lottery winning probability p.
- The ticket issue rate of both tkt_{L1} and tkt_{L2}, denoted as tkt_{rate1} and tkt_{rate2}, respectively, such that $tkt_{rate1} = ntkt_{rate2}$. These are the maximum number of tickets a publisher is allowed to hand out per round, which determine the ticket sequence numbers to be used within each ticket issuing round.
- The escrow lifetime l_{esc}, which is the total number of ticket issuance rounds. Therefore, the total number of tkt_{L1} and tkt_{L2} tickets a publisher may issue equals to $l_{esc}tkt_{rate1}$ and $l_{esc}tkt_{rate2}$, respectively.

The values of B_{escrow} and $B_{penalty}$ must meet the requirements outlined in MicroCash. That is, B_{escrow} must obey the $1 - \epsilon$ coverage rule (for some small system parameter ϵ), and $B_{penalty}$ must respect the lower bound needed to discourage cheating. We derive bounds for these balances, under the two ticket model, in Section 5.4. Verifying the correctness of the publisher's payment setup is performed by the miners in a similar way as described in Chapter 4.

Escrow Management

To track the locked funds in the escrows, miners maintain a state for each escrow in the system that includes the following:

- The ID of the escrow.
- The balances of both the payment and penalty escrows.
- The public key of the owner-publisher.
- The values of p, l_{esc}, tkt_{rate1} and tkt_{rate2}.
- The value of n and the set of beneficiary caches (including both the public key of each cache and a corresponding index).

[3]This hash is the root of the Merkle tree that is computed over all data chunks comprising the content. Beside publishing the root on the blockchain, the publisher makes the full tree public off-chain.

- A counter for each beneficiary cache to track the number of winning tkt_{L2} it owns under this escrow. This counter is used to compute the currency value of a winning tkt_{L1} claimed by a cache as will be shown later.
- A refund time for the escrow t_{refund} at which the publisher who owns the escrow can spend the remaining funds. The value of t_{refund} is equal to the expiry time of the tickets issued in the last round of an escrow lifetime.

As outlined in Chapter 4, lottery ticket issuance must follow a schedule that specifies which ticket sequence number range can be used for each round. CacheCash modifies this schedule to cover the sequence number range for the two ticket types it introduces. Furthermore, this schedule specifies for each n ticket of type tkt_{L1} used in a service session, what sequence number tkt_{L2} must be used in that session. For example, let $n = 4$, $tkt_{rate1} = 100$, and $tkt_{rate2} = 25$. In the first round after the escrow is confirmed, the sequence numbers $\{0, \ldots, 99\}$ and $\{0, \ldots, 24\}$ can be used for issuing tkt_{L1} and tkt_{L2}, respectively. Moreover, the first n tickets of type tkt_{L1}, i.e., sequence numbers 0-3 in our example, can be used in a ticket bundle with the first tkt_{L2} in the assigned range, i.e., sequence number 0. This assignment continues until the last round of the escrow lifetime.

The miners update the escrow state based on the escrow related transactions they process. For example, redeeming a winning tkt_{L1} reduces B_{escrow} by the currency value of this ticket. Announcing a winning tkt_{L2} increases the counter of the owner-cache by one. And receiving a valid proof-of-cheating against the publisher causes the funds in $B_{penalty}$ to be burned and the escrow to be broken. The miners stop tracking an escrow and discard its state once all tickets tied to this escrow expire. This happens at time t_{refund}, or when an escrow is broken after receiving a valid proof-of-cheating (discussed in Section 5.3.6). At that time, the publisher may spend any remaining funds.

5.3.4 Content Distribution

Content distribution in CacheCash proceeds in service sessions, with each session involving two actions: a publisher handling a content request from a client, and a client exchanging the payments issued by the publisher for data chunks from caches.

As mentioned previously, fair exchange between mutually-distrusted parties is impossible [76,108]. This introduces certain security threats for caches and publishers alike when it comes to exchanging service and payments. Paying a cache after it delivers the content is risky because a malicious publisher may not pay once its client has been served. Equally risky to the publisher is paying a cache in advance because a malicious cache may choose not to deliver any content after collecting a full payment. On top of all of this, publishers have to deal with cache accounting attacks in which a cache colludes with a client to collect payments without doing any actual work.

CacheCash introduces a novel service-payment exchange protocol embedded with several techniques to reduce the impact of these risks. Though this protocol follows the general design of the non-monetized CAPnet service model (Section 3.3.3), it adds several modifications. First, it uses micropayments to reward caches. This means that a session payment is a small value, which motivates each party to act honestly in the long run to increase its total profits. Second, it divides a session payment into two parts, tkt_{L1} and tkt_{L2}, to further reduce any potential monetary loss caused by malicious actors. A cheating cache that serves corrupted content will collect only tkt_{L1}, and a cheating publisher, possibly colluding with a client, can only save tkt_{L2} because a cache will not release the outer layer encryption key without receiving a valid tkt_{L1}. And third, a publisher masks tkt_{L2} with the cache accountability puzzle solution to prevent a colluding client and caches from collecting this ticket for free.

In what follows, we discuss the details of this exchange protocol as part of how CacheCash handles a service session.

Handling a Client Request by the Publisher

To retrieve a specific piece of content, a client sends a service request to the publisher asking for n data chunks[4]. After validating the request, by ensuring that the requested data chunks correspond to a content the publisher owns, the publisher accepts the request and prepares a response for the client in the form of a ticket bundle. This bundle contains all the information needed to retrieve the requested data chunks and pay the caches. The publisher prepares a ticket bundle as follows:

- Select a set of n caches to serve the client and assign each cache to serve a specific data chunk[5].
- Generate a cache accountability puzzle over these chunks (see Section 3.3.3). This involves generating all inner layer encryption keys for the current session, where the j^{th} key is denoted as $k_{in,j}$.
- Generate a data chunk request ticket, denoted as $tkt_{r,j}$, for each cache C_j in the selected set. This ticket has the following format (H is a hash function):

$$tkt_{r,j} = id_{escrow}||index_{C_j}||id_{D_j}||H(k_{in,j})||rseq \tag{5.1}$$

  where id_{esc} is the ID of the escrow that will pay for the service request, $index_{C_j}$ is the index of the cache as specified in the escrow state, id_{D_j} is the identity of the data chunk that this cache will serve, and $rseq$ is the request sequence number, which the same as tkt_{L2} sequence number that will be used in this session. The request ticket also contains the hash of $k_{in,j}$, which commits the publisher to the key it used when generating the puzzle. This is needed to prevent a malicious publisher from accusing a cache of sending a corrupted encrypted data chunk.
- Generate a set of n distinct tkt_{L1} tickets, each of which is denoted as $tkt_{L1,j}$. Each of these tickets has the following format, where $tseq$ is the sequence num-

[4]If the requested object is composed of more than n chunks, such as a movie, the client will send several requests covering the whole object.

[5]The publisher selects this set using any criteria of its choice, e.g., based on location, quality of service, reputation, etc. The details of this mechanism is outside the scope of CacheCash

ber of the ticket:

$$tkt_{L1,j} = id_{escrow}||index_{C_j}||tseq \tag{5.2}$$

- Generate one tkt_{L2} for all n caches. This ticket has the following format:

$$tkt_{L2} = id_{escrow}||index_{C_1}||\ldots||index_{C_n}||tseq \tag{5.3}$$

- Mask tkt_{L2}, and all inner layer encryption keys $k_{in,j}$, by encrypting them using the puzzle solution.
- Sign the ticket bundle.

A publisher signs the ticket bundle in two steps. It hashes each field individually, which involves hashing every single request ticket and lottery ticket to produce a set of hashes called h_{bundle}. Then, the publisher signs h_{bundle} to obtain a signature σ_{bundle}. Such an approach allows a cache to verify one signature over each individual ticket it obtains from the client, as will be explained in the next section, without having the publisher sign each ticket individually.

Upon receiving the ticket bundle, the client checks that its format compiles with the system specifications, and that it contains valid request and lottery tickets (these checks are the same as done by caches, which we introduce in the next subsection). Then, the client computes h_{bundle} as done by the publisher, after which it verifies σ_{bundle} over h_{bundle} using the publisher's public key appearing in the escrow transaction that is recorded on the blockchain.

Data Chunk Retrieval from Caches

With a valid ticket bundle in hand, the client can start retrieving data chunks from caches. This action, as depicted in Figure 5.3, involves multiple steps as follows.

Step 1: As shown in the figure, the client starts by forwarding each request ticket $tkt_{r,j}$ to its designated cache. This is accompanied by a copy of h_{bundle} and the publisher's signature σ_{bundle} so that the cache can verify the correctness of this ticket.

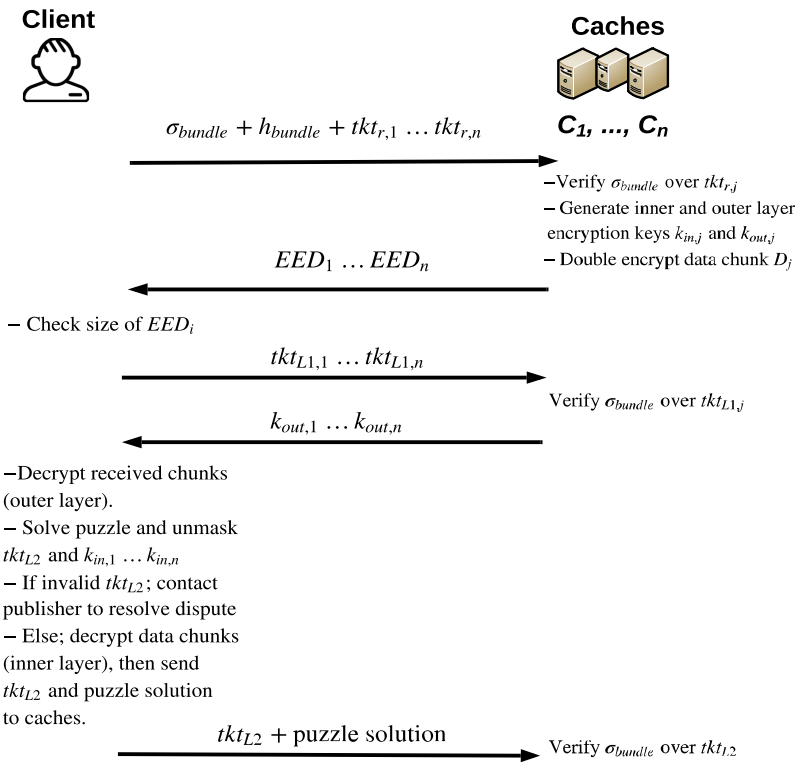


Figure 5.3: Data chunk retrieval from caches.

Step 2: Each cache C_j validates the received request ticket by checking that its format complies with the system specifications, ensuring the freshness of $rseq$, checking that the ticket has the correct cache index as listed in the escrow state, and generating $k_{in,i}$ (in the same way using a keyed PRF as described in Section 3.3.3) and verifying that the ticket contains the correct key digest. The cache also checks that the escrow referenced in the ticket is not broken, and, lastly, it verifies the publisher's signature over the ticket. The latter is done by, first, hashing $tkt_{r,i}$ and checking if an identical hash is found inside h_{bundle}. And second, verifying σ_{bundle} over h_{bundle} using the publisher's public key found in the escrow state.

After verifying the request ticket, cache C_j double encrypts the data chunk D_j it will deliver (the double encrypted chunk is denoted as EED_j in Figure 5.3). That is, this cache uses $k_{in,j}$ to encrypt the raw data chunk, and then it generates a fresh outer layer encryption key $k_{out,j}$ at random and use it to encrypt the produced ciphertext. At the end, each cache C_j sends EED_j to the client.

Step 3: Once the client receives EED_j, it sends each cache its lottery ticket $tkt_{L1,j}$ in order to receive the outer layer encryption key this cache used in the session.

Step 4: After receiving $tkt_{L1,j}$, cache C_i validates it as follows:

- Parse the ticket as $(id_{escrow}, index_{C_j}, tseq)$.
- Check that $index_{C_j}$ is identical to its own index listed in the escrow state (this also implies that a cache checks whether it is a beneficiary of the escrow).
- Check that the ticket sequence number $tseq$ is within the allowed sequence number range based on the ticket issuing schedule of the escrow.
- Verify σ_{bundle} over this ticket by only hashing the ticket and looking for an identical hash in h_{bundle}. Note that there is no need to verify σ_{bundle} again over h_{bundle}.

If all the above checks pass, the cache accepts the lottery ticket, and shares $k_{out,j}$ with the client as an acknowledgment. If any of these checks fails, the cache drops the client request unless it fails the second or third checks. A ticket with an out-of-range sequence number, or destined to a cache that is not a beneficiary of the escrow, can be used to issue a proof-of-cheating against the publisher.

Step 5: By using all $k_{out,j}$ from all caches, the client decrypts the outer encryption layer of all data chunks. It uses the produced inner-layer encrypted chunks to solve the puzzle challenge found in the ticket bundle. The puzzle solution allows the client to unmask tkt_{L2} and the inner layer encryption keys $k_{in,1}, \ldots, k_{in,n}$. The client will share a copy of tkt_{L2}, along with the puzzle solution, with each cache and use the keys to decrypt the data chunks. This recovers the requested content.

If the unmasking process of tkt_{L2} fails, it means that either a cache(s) has sent a corrupted data chunk, or the publisher has sent an incorrect puzzle or ticket, to avoid paying caches their tkt_{L2}. This dispute can be solved by having the client send the hashes of the inner-layer encrypted data chunks to the publisher to identify the

cheating cache (more about this in Section 5.3.6)[6]. A cheating publisher cannot be detected in this manner, but CacheCash can handle this case financially as will be shown later. Meanwhile, the publisher directs the client to contact other caches to retrieve valid copies of the corrupted data chunks.

Step 6: Each cache validates tkt_{L2} as in **Step 4** with two differences: First, it must mask this ticket using the puzzle solution before verifying the publisher's signature. And second, beside checking that $tseq$ is within the valid range according to the ticket issue schedule, the cache needs to check that $tseq$ of this ticket is the correct one that must be used with $tseq$ of tkt_{L1}. Also, this number must be identical to $rseq$ found in $tkt_{r,j}$ received earlier.

A service session is considered complete when the client successfully retrieves all the requested data chunks. If the client wishes to request more content, it will initiate another service session with a new content request.

5.3.5 Payment Processing

Any cache keeps each lottery ticket it receives until the lottery draw time of that ticket. It then determines if this is a winning ticket using the same lottery protocol of MicroCash (Section 4.4.4). A cache claims a winning ticket by issuing a redeem transaction to the miners containing the winning ticket and a copy of the ticket bundle signature (both h_{bundle} and σ_{bundle}). This claim need to be issued before a ticket expires, which happens in d_{redeem} rounds after the ticket lottery draw time.

The miners process the redeem transaction by checking its format and that the claimed ticket is indeed a valid winning, not-expired and non-duplicated one, i.e., no other winning ticket of the same type tied to the same escrow carries an identical $tseq$. Validity is checked using the same procedure found in **step 4** (for tkt_{L1}) or **step 6** (for tkt_{L2}) in the previous subsection. The only difference is that a miner needs to verify

[6]This requires a cache to sign the EED_j and $k_{out,j}$ before sending them to the client. We do not implement this in the current version of CacheCash.

σ_{bundle} over h_{bundle} if they have not seen this signature before. The miners also ensure that the cache owns the claimed ticket by verifying its signature over the redeem transaction using the cache's public key (found in the escrow state) that corresponds to the cache index found inside the winning ticket. If any of these checks does not pass, the miners drop the redeem transaction, while they issue a proof-of-cheating against the publisher if duplication is detected. Otherwise, the miners approve the redeem transaction.

How the escrow state is updated using a redeem transaction depends on the ticket type. Only a winning tkt_{L1} triggers currency transfer out of the publisher's escrow, while a winning tkt_{L2} increments the tkt_{L2} counter of the owner-cache by one. The currency value of a winning tkt_{L1} is computed as $f(z) = az$, where z is the cache's tkt_{L2} counter value under the same escrow that pays this winning tkt_{L1}. Therefore, the miners will first update the counter that a cache owns by counting all of its winning tkt_{L2} issued in a round. Then, they will compute the currency value of winning tkt_{L1} tickets that were issued in the same round.

The above payment function, and a bound for the constant a, are derived based on an economic analysis that aims to make honesty the most profitable option for publishers and caches (details are found in Section 5.6). This analysis also shows that tkt_{L2} counters need to be reset with each new escrow to avoid continued increase in service price over time.

5.3.6 Proof-of-cheating Processing

CacheCash adopts the same mechanism for detecting and processing proofs-of-cheating adopted in MicroCash (Section 4.4.6). The only difference is that due to payment process modifications CacheCash introduces, the lower bound for the penalty escrow is different (see equation 5.12 in Section 5.4).

In this section, we discuss other cheating behaviors that are specific to CacheCash and cannot be proven to a third party. These include a publisher that sends an invalid puzzle in a ticket bundle or invalid tkt_{L2}, or a cache that sends a corrupted data chunk and collects tkt_{L1} without revealing the outer layer encryption key. These behaviors

are addressed using rational financial incentives. By modeling the system operation as a repeated game, and having caches stop working with a cheating publisher or a publisher blacklisting a cheating cache, we show how these malicious behaviors can be made unprofitable in the long run. Hence, rational publishers, clients, and caches will choose to act honestly. A complete discussion can be found in Section 5.6.

5.4 Economic Analysis for Escrows

In this section, we show how to compute the payment escrow balance B_{escrow} in a way that satisfies the $1-\epsilon$ coverage rule under the two ticket type model CacheCash adopts. We also compute a lower bound for the penalty deposit required to deter cheating.

5.4.1 Computing B_{escrow}

Similar to MicroCash, we compute B_{escrow} using a probabilistic analysis that relies on modeling the payment process in CacheCash. In what follows, we state and prove a formula to calculate this balance.

Theorem 3. *For an escrow with lifetime l_{esc} rounds, ticket issue rates tkt_{rate1} and tkt_{rate2}, lottery winning probability p, number of caches per service session n, payment function $f(z) = az$ as defined earlier, and parameter ϵ, where $l_{esc}, tkt_{rate1}, tkt_{rate2}, n \in \mathbb{N}$, $a \in \mathbb{R}^+$, and $0 \leq p, \epsilon \leq 1$, the value of B_{escrow} needed to cover all winning lottery tickets with probability at least $1-\epsilon$ under CacheCash setup is given by:*

$$B_{\text{escrow}} = a\omega \sum_{i=1}^{i^*-1} F_i^{-1}(p, i\, tkt_{rate1}, 1 - \frac{\epsilon}{2l_{esc}}) + 0.5\, a\, \omega\, p\, tkt_{rate1}\big(l_{esc}(l_{esc}+1) - i^*(i^*-1)\big) \tag{5.4}$$

such that:

$$i^* = \frac{-3\ln(1-0.5\epsilon)}{p\,\epsilon^2 tkt_{rate1}} \tag{5.5}$$

and

$$\omega = F^{-1}(p, tkt_{rate2}, 1 - \frac{\epsilon}{2l_{esc}}) \tag{5.6}$$

F_i^{-1} is the inverse of the cumulative distribution function (CDF) of the binomial distribution parameterized by p and $t = i\, tkt_{rate1}$ at the value $1 - \frac{\epsilon}{2l_{esc}}$, and F^{-1} is the inverse of the CDF of the binomial distribution parameterized by p and $t = tkt_{rate2}$ at the value $1 - \frac{\epsilon}{2l_{esc}}$.

Proof. In CacheCash, the total number of winning tickets (of any type) can be modeled as a random variable that follows a binomial distribution (this is because the winning events of these tickets are independent). Our goal is to use these distributions to compute the number of tickets (of each type) that will win with probability at maximum $1 - \epsilon$ and use it to compute B_{escrow}.

Recall that the currency value of any winning tkt_{L1} in round i is tied to the total number of winning tkt_{L2} a cache owns accumulated over all rounds until round i. Therefore, we need to determine the number of tkt_{L2} each cache collects per round in order to compute this currency value. We consider an extreme case where the same n caches are selected for all service sessions paid by an escrow. This means that each cache receives $\frac{tkt_{rate1}}{n}$ tickets of type tkt_{L1}, and tkt_{rate2} tickets of type tkt_{L2} per round. Thus, the expected currency value of any winning tkt_{L1}, in a specific round, will be the same because all these n caches receive the same number of tkt_{L2}.

We use random variables $Z_1, \ldots, Z_{l_{esc}}$ to represent the number of winning tkt_{L2} in rounds $i \in \{1, \ldots, l_{esc}\}$, respectively. Hence, each Z_i follows a binomial distribution parameterized by p and a number of trials $t = tkt_{rate2}$. To obey the $1 - \epsilon$ coverage rule, we need to compute the the number of winning tkt_{L2} in round i that hits the $(1 - \frac{\epsilon}{2l_{esc}})$th percentile of Z_i.[7] This number, which is denoted as ω, can be computed as follows:

$$\omega = F^{-1}(p, tkt_{rate2}, 1 - \frac{\epsilon}{2l_{esc}}) \tag{5.7}$$

[7]We use $\frac{\epsilon}{2l_{esc}}$ to account for the union bound of all error terms in all rounds, for both ticket types, of the escrow lifetime.

where F^{-1} is the inverse of the cumulative distribution function (CDF) of the binomial distribution parameterized by p and $t = tkt_{rate2}$ at the value $1 - \frac{\epsilon}{2l_{esc}}$.

For tkt_{L1}, we apply a different method to make the bound of B_{escrow} tighter, and hence, reduce the amount of funds a publisher needs to lock in a payment escrow. This method is based on observing that the number of winning tkt_{L2} in the first round, i.e., ω, contributes in the currency value of all winning tkt_{L1} from all ticket issuing rounds. While the number of winning tkt_{L2} in the second round, which is also ω, contributes in the currency value of all winning tkt_{L1} in rounds $2, \ldots, l_{esc}$, and so on. This means that ω from the first round is multiplied by the total number of winning tkt_{L1} among $l_{esc}tkt_{rate1}$. And ω from the second round is multiplied by the total number of winning tkt_{L1} among $(l_{esc} - 1)tkt_{rate1}$, and so on. As such, and using the payment function $f(z) = az$ described earlier, B_{escrow} can be expressed as follows:

$$B_{escrow} = a \sum_{i=1}^{l_{esc}} \omega F_i^{-1}(p, i\, tkt_{rate1}, 1 - \frac{\epsilon}{2l_{esc}}) \tag{5.8}$$

where F_i^{-1} is the inverse of the CDF of the binomial distribution representing all winning tkt_{L1} in rounds $i, \ldots, l_{esc}$ (hence, it is parameterized by p and $t = i\, tkt_{rate1}$) at the value $1 - \frac{\epsilon}{2l_{esc}}$.

We can further simplify the above expression by using a Chernoff bound [102] to show that for large t, the $(1 - \frac{\epsilon}{2l_{esc}})$th percentile is not far from the mean. We first state the Chernoff bound and then show how to apply it.

Theorem 4 (Chernoff Bound [102]). *Let $X = \sum_{i=1}^{n} X_i$, where $X_i = 1$ with probability p_i and $X_i = 0$ with probability $1 - p_i$, and all X_i are independent. Let $\mu = \mathbb{E}(X) = \sum_{i=1}^{n} p_i$, then for any $0 < \lambda \leq 1$:*

$$\Pr(X \geq (1 + \lambda)\mu) \leq e^{-\frac{\mu\lambda^2}{3}} \tag{5.9}$$

In our analysis, we set $\lambda = \epsilon$ and we require that $\Pr(X \geq (1 + \lambda)\mu) \leq 1 - 0.5\epsilon$, where $\mu = p\, i^*\, tkt_{rate1}$. In other words, we want to define the value i^* such that for any round with index $i \geq i^*$ we can use the expectation to compute the number of

winning tkt_{L1} instead of computing the $(1-0.5\epsilon)$th percentile[8]. Accordingly, using the Chernoff bound we get $e^{-\frac{\mu\epsilon^2}{3}} \leq 1-0.5\epsilon$, which produces the following:

$$i^* \geq \frac{-3\ln(1-0.5\epsilon)}{p\,\epsilon^2 tkt_{rate1}} \tag{5.10}$$

For efficiency reasons, and to reduce the value of B_{escrow} as possible, we use the minimum value of i^* given by the above expression (this reduce the number of times the inverse of the binomial CDF needs to be computed).

Based on that, B_{escrow} can be expressed as follows:

$$\begin{aligned} B_{escrow} &= a\omega\Bigg(\sum_{i=1}^{i^*-1} F_i^{-1}(p, i\,tkt_{rate1}, 1-\frac{\epsilon}{2l_{esc}}) + \sum_{i=i^*}^{l_{esc}} p\,i\,tkt_{rate1}\Bigg) \\ &= a\omega\sum_{i=1}^{i^*-1} F_i^{-1}(p, i\,tkt_{rate1}, 1-\frac{\epsilon}{2l_{esc}}) \quad + \\ &\quad 0.5\,a\,\omega\,p\,tkt_{rate1}\big(l_{esc}(l_{esc}+1) - i^*(i^*-1)\big) \end{aligned} \tag{5.11}$$

Which completes the proof. □

The above equation is used by the publisher and the miners when computing the required payment escrow balance that satisfies the $1-\epsilon$ coverage rule.

5.4.2 Computing a Lower Bound for $B_{penalty}$

In this section, we compute a lower bound for the penalty deposit required to deter cheating in a similar way as done in Section 4.5.2, while accounting for the two ticket type model. We quantify the additional utility gain, or monetary profit, a malicious publisher could obtain as compared to an honest one. Then, we set the penalty deposit to at least equal this additional utility, which makes cheating unprofitable in expectation compared to acting honestly, and hence, unappealing to rational publishers.

[8]We consider the union bound for the error term $\frac{\epsilon}{2l_{esc}}$ over l_{esc} rounds as this is the period that is covered by computing the number of winning tkt_{L1}.

This analysis covers the same malicious strategies outlined in Section 4.5.2. Again, since ticket issuing duplication has a larger utility gain, we only consider this strategy in this section. In what follows, we present this analysis including the game setup, a definition for the utility gain function, and and finally state and prove a lower bound for $B_{penalty}$.

Game setup. We have a single player game in which a malicious publisher applies the ticket duplication strategy. In general, this strategy is defined as duplicating any sequence number among two or more tickets that are tied to the same escrow. In CacheCash this duplication is affected by the service setup. A publisher cannot duplicate all tkt_{L1} among all beneficiary caches of an escrow. This is because a cache knows that a publisher can afford at maximum $\frac{tkt_{rate1}}{n}$ service sessions per round[9], which equals to the maximum number of tkt_{L2} a publisher can issue per round. A rational cache will not participate in more than $\frac{tkt_{rate1}}{n}$ service sessions because it knows that any additional session will use invalid or duplicated lottery tickets.

Ticket duplication is also affected by whether a publisher is colluding with the client. This collusion allows giving duplicated tkt_{L1} to all n caches within the same service session. By doing so, the cheating detection probability becomes smaller, and hence, the utility gain of a cheating publisher becomes larger. To see this, assume that an escrow has $N = 4$ beneficiary caches, $n = 2$, and let caches $\{C_1, C_2\}$ and $\{C_3, C_4\}$ form two service sessions with different clients. Without colluding with the clients, a publisher who decides to duplicate tickets among these sessions would need two different tkt_{L1} sequence numbers. Each sequence number can be duplicated, at maximum, among two caches from different sessions. But with collusion, a publisher can hand all the four caches tkt_{L1} with the same sequence number. Given that the lottery draw does not depend on the ticket recipient address (see equation 4.2 in Section 4.4.4), all tickets with the same sequence number win, or lose, together. Thus,

[9]Recall that a service session involves a client and a set of n caches selected to fulfill a client's content request. Each of these caches receives one tkt_{L1} with a unique sequence number, and one tkt_{L2} that is destined to all these n caches. The sequence number of tkt_{L2} is determined based on the sequence numbers of tkt_{L1} tickets handed out in the session.

the use of two duplicated sequence numbers has higher cheating detection probability than having one duplicated sequence number. For this reason, we consider the case of a publisher colluding with the client because this collusion increases the utility gain.

Furthermore, a rational publisher that decides to duplicate a ticket, will duplicate it among all beneficiary caches. This is because duplication among fewer than N caches does not reduce the cheating detection probability. In addition, the utility gain of duplicating a ticket increases when the N caches are distributed among disjoint service sessions, i.e., among $\frac{N}{n}$ disjoint cache sets. Having overlapped sets prevents a publisher from duplicating any ticket among all N caches because common caches will detect cheating immediately.

Based on the above, ticket duplication in CacheCash involves duplicating a tkt_{L1} sequence number among all N caches, and duplicating the corresponding tkt_{L2} sequence number among all $\frac{N}{n}$ cache sets. In total, in any round a publisher can make up to $\frac{tkt_{rate1}}{n}$ duplication decisions. Since each duplication decision corresponds to two sequence numbers; one for duplicated tkt_{L1} and the other is for duplicated tkt_{L2}, we refer to each duplication decision as duplicating a ticket tuple. Also, as before, the cheating detection period of ticket duplication is $d_{draw} + d_{redeem}$ rounds[10].

Table 5.1 shows the set of notations used in this section. (Some of these symbols already appeared in Table 4.1, but we repeat them for completeness.)

Utility gain function definition. As before, we define the utility gain function of any publisher as the service value minus the payments made to caches. We compute the expected value of this function for an honest publisher and for a cheating one that applies the ticket duplication strategy. In order to make cheating unprofitable, we set $B_{penalty}$ to be at least equal to the maximum additional expected utility a malicious publisher obtains over what an honest publisher can achieve.

Additional utility gain analysis. We now state and prove a lower bound for $B_{penalty}$ based on the above game setup.

[10]We do not assume that caches exchange any information between each others about tickets.

Table 5.1: Notations I.

Symbol	Meaning
P	Honest publisher.
$\hat{P}$	Malicious publisher.
$u(\cdot)$	Utility gain function.
τ	Max number of duplicated ticket tuples a publisher can choose per round (this equals to tkt_{rate2}, or alternatively, $\frac{tkt_{rate1}}{n}$), such that $\tau \in \mathbb{N}$.
d	d_{draw} (lottery draw period in rounds), such that $d \in \mathbb{N}$.
r	d_{redeem} (ticket redemption period in rounds), such that $r \in \mathbb{N}$.
y_i	Number of duplicated ticket tuples in round i, such that $0 \leq y_i \leq \tau$.
N	Number of beneficiary caches of an escrow, such that $N \in \mathbb{N}$.
n	Number of caches per service session, such that $n \in \mathbb{N}$.
υ	l_{esc} (the escrow lifetime in rounds), such that $\upsilon \in \mathbb{N}$.
p	Lottery winning probability, such that $0 \leq p \leq 1$.
$f(\cdot)$	The payment function that computes the currency value of a winning tkt_{L1}.
a	A positive constant used to configure the payment function $f(z) = az$ where z is the value of winning tkt_{L2} counter, and $a \in \mathbb{R}^+$

Theorem 5. *For the game and escrow setup described above, issuing invalid or duplicated lottery tickets is less profitable, in expectation, than acting in an honest way if:*

$$B_{\text{penalty}} > ap^2\tau^2\left(\frac{\upsilon - d - r + N - 1}{1-(1-p)^{2\tau}} + (\upsilon - d - r)(d + r - 1) + \right.$$
$$\left. 0.5(N-1)(d+r)(d+r+1) - (N-1)\right) \quad (5.12)$$

Proof. In CacheCash, a publisher can create an escrow with υ round lifetime. All tickets issued in a round enter the lottery after d rounds, and all winning tickets will expire after r rounds from the lottery draw time.

During each round of the escrow lifetime, an honest publisher can issue up to $n\tau$ tickets of type tkt_{L1}, and up to τ tickets of type tkt_{L2} with unique sequence numbers.

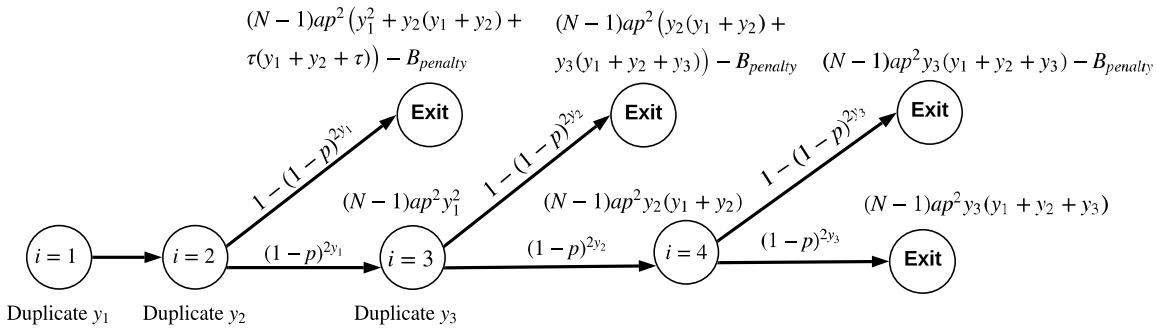


Figure 5.4: Decision process for a 3 round escrow with $d = 2$ rounds and $r = 1$ round. Arrows carry probabilities, decisions are found below the states, and the utility gain is found above the states.

In round i, each tkt_{L1} has an expected value of $pf(z_i)$ coins, where z_i is the expected number of winning tkt_{L2} the recipient cache owns in round i. This expected value is the service value a publisher obtains for handing out tkt_{L1} (and its corresponding tkt_{L2}), which may increase with time due to collecting more winning tkt_{L2}. We use this service value in computing the utility gain function for the publisher.

In contrast, a malicious publisher would decide to duplicate y_i ticket tuples in round i, where $i \in \{1, \ldots, \upsilon\}$ and $y_i \in \{1, \ldots, \tau\}$. Each tuple corresponds to two sequence numbers that are used to issue duplicated tkt_{L1} and tkt_{L2} for all N caches. If any of these tickets wins the lottery at round $i+d$, which happens with probability $1-(1-p)^{2y_i}$, the publisher will be detected at round $i+d+r$ (the latest). Once the lottery draw outcome is determined, this publisher still has r rounds to issue tickets before it is detected. Therefore, as a rational behavior, this publisher will choose to duplicate all τ ticket tuples in these rounds. On the other hand, if none of the duplicated tickets win, which happens with probability $(1-p)^{2y_i}$, the publisher stays in the system.

In order to compute the additional utility gain, we model the ticket duplication decisions a malicious publisher would make at each round using a decision process diagram. As an example, we consider a simple case where we have an escrow with 3 round lifetime, $d = 2$ rounds, and $r = 1$ round as illustrated in Figure 5.4.

As shown in Figure 5.4, a malicious publisher duplicates y_1 and y_2 ticket tuples in round 1 and 2, respectively. If none of these tickets win (it happens with probability

$(1-p)^{2y_1}$), the malicious publisher obtains an additional utility gain of $(N-1)ap^2y_1^2$ and proceeds to round 3. That is, each cache will have an expected counter value of py_1 (each cache received y_1 tickets of type tkt_{L2} in round 1). Hence, any winning tkt_{L1}, where a total of Ny_1 tickets of type tkt_{L1} has been issued in round 1, will have an expected value of ap^2y_1. So the total service value from all N caches is $Nap^2y_1^2$. Compared to an honest publisher that issues unique tickets to all these caches, the malicious publisher obtains additional service value from $(N-1)$ caches (which equals to $(N-1)ap^2y_1^2$ as shown in the figure).

On the other hand, if any of the duplicated tickets in round 1 wins (it happens with probability $1-(1-p)^{2y_1}$), the publisher knows that it will be detected at the end of round 3 (since $r=1$). Hence, it decides to duplicate all ticket tuples it has in round 3 (i.e., $y_3=\tau$). The total additional utility it obtains in this case is the sum of the utility gain of duplicating tickets in 3 rounds minus the penalty deposit that will be revoked. As shown in the figure, the counter value of winning tkt_{L2} increases over time. Therefore, a duplicated tkt_{L1} in later rounds has larger utility value than earlier rounds.

We use the same technique that we applied in Section 4.5.2,where we formulate the expected utility gain of a malicious publisher in a recursive way. The expected utility gain of an υ round escrow is expressed in terms of the expected utility gain of an $\upsilon-1$ round escrow, and so on. During the first round of an υ round escrow, all caches start with zero counters of winning tkt_{L2}. We refer to these counters as a vector $\mathbf{c}=(c_1,c_2,\ldots,c_N)$, where c_j is the counter value for cache C_j. By duplicating y_1 ticket tuples in this rounds, with probability $1-(1-p)^{2y_1}$, the additional utility gain of the malicious publisher is (where $c=\sum_{j=1}^{N}c_j$):

$$\phi_{\mathbf{c}} = ap\left(\sum_{i=1}^{d} y_i\left(c+(N-1)\sum_{j=1}^{i} py_j\right)\right)+ ap\left(\sum_{i=d+1}^{d+r}\tau\left(c+(N-1)\sum_{j=1}^{d} py_j+(N-1)(i-d)p\tau\right)\right)-B_{penalty} \quad (5.13)$$

That is, when the lottery draw outcome is determined for the y_1 duplicated tickets

(or the first round of the $d + r$ period), each cache C_j will have an expected counter value of $c_j + py_1$. Each of the duplicated tkt_{L1} has an expected service value of $p(c_j + py_1)$, where c_j is determined by the cache that owns the duplicated ticket. So the total service value obtained from all caches who received this duplicated tkt_{L1} is $p(c + Npy_1)$. The additional utility gain the publisher obtains, as compared to an honest one, will be this total minus the service value of one cache. To allow accounting for arbitrary initial counter value $\mathbf{c}$, we set this additional gain to be $p(c+(N-1)py_1)$, and hence, for all y_1 duplicated tkt_{L1} it will be $py_1(c+(N-1)py_1)$. This produces a more conservative bound than the actual one because it involves the sum of the counters for all N caches instead of $N-1$. The same analogy is applied for the duplicated tickets at later rounds, but with updating the initial counter value properly, as shown in equation 5.13.

If none of the duplicated tickets win the lottery, the publisher stays in the system. This means that round 2 is a fresh start for this publisher in an $\upsilon - 1$ round escrow. However, caches in this $\upsilon - 1$ round escrow have non-zero winning tkt_{L2} counters. We refer to these updated counters as a vector $\mathbf{c}'$. In this case, the utility gain of the cheating publisher will be $apy_1(c+(N-1)py_1)+\mathbb{E}_{\upsilon-1,\mathbf{c}'}[u(\hat{P})]$, where the second term denotes the expected utility gain in an $\upsilon - 1$ round escrow that has counter state $\mathbf{c}'$.

Based on the above, we can express $\mathbb{E}_{\upsilon,\mathbf{c}}[u(\hat{P})]$, which is the quantity of interest, as follows (with $\phi_{\mathbf{c}}$ as given above):

$$\mathbb{E}_{\upsilon,\mathbf{c}}[u(\hat{P})] = \big(1-(1-p)^{2y_1}\big)\phi_{\mathbf{c}} + (1-p)^{2y_1}\Big(apy_1(c+(N-1)py_1) + \mathbb{E}_{\upsilon-1,\mathbf{c}'}[u(\hat{P})]\Big) \quad (5.14)$$

Using the same argument we used earlier for MicroCash, i.e., we have $\mathbb{E}_{\upsilon-1,\mathbf{c}'}[\hat{P}] \leq$

0 and we require $\mathbb{E}_{\upsilon,\mathbf{c}}[u(\hat{P})] \leq 0$, we find that:

$$B_{penalty}(c, y_1, \ldots, y_d) \geq \frac{apy_1\big(c + py_1(N-1)\big)}{1-(1-p)^{2y_1}} + ap\sum_{i=2}^{d} y_i\left(c + (N-1)\sum_{j=1}^{i} py_j\right) + ap\sum_{i=d+1}^{d+z} \tau\left(c + (N-1)\sum_{j=1}^{d} py_j + (i-d)(N-1)p\tau\right) \quad (5.15)$$

In order to compute the maximum additional utility gain, we need to find the values of $c, y_1, \ldots, y_d$ that will maximize the above expression. Starting with c, the most expensive service period of an escrow lifetime is the last $d + r$ rounds, i.e. the last full cheating detection period. As such, $B_{penalty}(c, y_1, \ldots, y_d)$ is maximized by setting c to its maximum value after operating for $\upsilon - d - r$ rounds in the system. To be conservative, we consider a publisher that issues all tickets in all rounds (i.e., it operates at the maximum allowed service rate). This makes the expected value of c at the lottery draw time for the tickets issued in the first round of the last $d + r$ round period to be $p\tau(\upsilon - d - r)$ (recall that $\tau = tkt_{rate2}$).

For y_i, given the maximum c value, we find that for any d and r value, the above quantity is maximized when $y_i = \tau$ for $i \in \{1, \ldots, d\}$.[11] Substituting this in equation 5.15 produces the lower bound stated in the theorem, which completes the proof. □

5.5 Security Analysis

In this section, we analyze the resilience of CacheCash to the types of threats outlined earlier in its threat model. In investigating all possible attack scenarios, using ABC's collusion matrices, we analyzed 420 threat cases to distill 15 cases that CacheCash must defend against in order to secure the system operation[12].

[11] This is done in a similar way as described in Section 4.5 for the same set of p and τ values.

[12] As mentioned in Section 2.5 in Chapter 2, CacheCash's threat model have 525 cases that were reduced to 22 threat scenarios. These include the cases from Bitcoin's threat model. Excluding Bitcoin's related cases leaves us with 420 cases and 12 distilled threat scenarios as found in the full threat model [31]. However, in this section we group some scenarios together and divide other cases among multiple scenarios. Thus, we have 15 distilled cases instead of 12.

Several threat cases are addressed by the secure design CAPnet and MicroCash. CacheCash defends against the rest by using a combination of cryptographic and economic techniques. The economic defenses, which exploit rational financial incentives, will be noted here, but addressed in detail in the next section. The rest of this section discusses the 15 threat cases and the defense mechanisms that CacheCash employs against them.

Service corruption. In this threat, a malicious cache(s) delivers corrupted data chunks to the client. The goal is to collect payments while making the client construct different content than what was requested (this case is labeled as **T1**).

This threat is addressed financially. If it receives one or more corrupted data chunks, the client will fail to solve the accountability puzzle over these chunks. This costs caches their tkt_{L2} because the client will fail in unmasking this ticket. As will be shown in the next section, it is more profitable for a cache to collect both tkt_{L1} and tkt_{L2} in every service session it participates in than to collect only tkt_{L1}. This is because of the payment setup of CacheCash, where the number of winning tkt_{L2} controls the currency value of a winning tkt_{L1}. Hence, a rational cache will act honestly to maximize its utility gain.

Cache accounting attacks. In this type of attack, malicious caches may try to collect payments from the publisher without doing the promised work. This threat can be exploited using several strategies (**T4** *is addressed by CAPnet*):

(**T2**) An attacker issues itself, or other caches, lottery tickets tied to the publisher's escrow.

(**T3**) A client, who is not interested in the data, colludes with caches to hand them tkt_{L1} without retrieving any content.

(**T4**) A client, who again is not interested in the data, colludes with caches to solve the cache accountability puzzle and unmask tkt_{L2} without retrieving the full content.

Threat **T2** is neutralized by requiring the publisher to sign any ticket bundle it issues, with this signature covering all request and lottery tickets inside the bundle. Given that our system uses a secure digital signature scheme, any attacker will succeed in issuing valid tickets with negligible probability.

Threat **T3** is addressed financially as mentioned earlier. In order to increase the currency value of a winning tkt_{L1}, a cache is motivated to collect tkt_{L2}. In the long run, collecting only tkt_{L1}, or even collecting fewer tkt_{L2} than what an honest cache would do, is less profitable than acting in an honest way. This makes this threat case unappealing to rational caches.

Threat **T4** is addressed by the security of the CAPnet scheme. A publisher can configure the number of iterations in the puzzle in a way that forces any malicious puzzle solver to expend a predetermined bandwidth bound that is very close to what is required to retrieve the content.

Service theft. In this threat, honest caches that serve clients correctly may lose their payments, fully or partially, in various ways including (**T6 - T9** *are addressed by MicroCash*):

(**T5**) A publisher generates invalid lottery tickets.

(**T6**) A publisher performs a front running attack by withdrawing the payment escrow before caches can claim their winning tickets.

(**T7**) A publisher issues more tickets than what can be covered by the payment escrow. This includes issuing tickets with sequence numbers that exceed the allowed range or tickets with duplicated sequence numbers.

(**T8**) An attacker manipulates the lottery draw in order to make certain tickets win, or lose, based on which cache owns the tickets.

(**T9**) A publisher issues winning tickets to itself after observing the lottery draw outcome. The goal is to drain the escrow by claiming these tickets before caches can claim their winning tickets.

(**T10**) A publisher generates an incorrect cache accountability puzzle that a client cannot solve even over the correct data chunks. Therefore, this client will not

be able to unmask tkt_{L2} and caches will receive only tkt_{L1} for such a session.

(**T11**) A client, acting on its own or in collusion with the publisher, does not hand caches their tkt_{L2} after retrieving the full content.

Invalid lottery tickets (**T4**) are detected instantly by the client when verifying the correctness of the received ticket bundle. An honest client will reject any invalid bundle, and hence, will not contact the caches. The case of a malicious client that colludes with the publisher and sends caches invalid tkt_{L1} or tkt_{L2} is addressed both cryptographically and financially. Any cache that receives an invalid tkt_{L1} will not send the outer layer encryption key to the client. Therefore, the client, who is interested in retrieving the data, will get only double encrypted data chunks that do not allow access to the raw content. On the financial side, frequent loss of lottery tickets will make caches stop working with the cheating publisher, thus forcing it to pay for the service setup cost of a new network of caches. As will be shown in the next section, such a cost exceeds the profit of applying threat **T4**, making this strategy unappealing to rational publishers.

As mentioned above, threats **T6-T9** are addressed by the secure design of MicroCash. Publishers cannot withdraw their escrows before time t_{refund}, and issuing tickets with out-of-range or duplicated sequence numbers will cost the publisher its penalty deposit. Respecting the lower bound of the penalty deposit (as given by equation 5.12) makes these cheating behaviors unprofitable compared to acting in an honest way. Furthermore, MicroCash prevents lottery manipulations by requiring the publisher to issue tickets using a fixed ticket issue schedule with a future lottery draw time. Hence, a publisher cannot tell which ticket will win or lose in advance. MicroCash's lottery design also discourages malicious miners from performing selective mining in which they forgo a block that does not produce a favorable lottery. All these defenses ensure that caches get their fair chance in the lottery protocol of the adopted payment scheme.

Both Threats **T10** and **T11** are addressed financially in a similar way to Threat **T5**. Frequent loss of tkt_{L2} will cause caches to stop working with the publisher, forcing it to pay the cost of constructing a new network of caches. Thus, a rational

publisher will not follow these attack strategies. In the case of a client acting on its own in Threat **T11**, the publisher needs to employ mechanisms to monitor its clients. Frequent complaints from caches about losing their tkt_{L2} may make the publisher blacklist a malicious client, stopping any future service requests. The design of such service feedback techniques is outside the scope of the current work.

DoS. As DoS is a large threat category, we only consider here cases that are different and unique to the design of CacheCash. These include:

(**T12**) An attacker overwhelms publishers or caches by sending them a large number of forged content requests or old (replayed) ones.

(**T13**) Caches ignore all data chunk requests coming from a specific client.

(**T14**) A publisher avoids selecting specific caches to serve clients. Thus, these caches waste resources storing the content and waiting for client requests, without collecting any payments.

(**T15**) A publisher distributes corrupted content to caches during the service setup phase in order to accuse them later of behaving maliciously and serving clients incorrect data. The goal is to waste the resources of these caches and destroy their reputation in the system.

Threat **T12** is neutralized by requiring any client to sign all content requests it issues, any publisher to sign all request tickets it issues, and any issued request to include a fresh sequence number. Given that the system uses a secure digital signature scheme, an attacker will fail in issuing valid requests because it needs to forge the client/publisher signature over these requests. Furthermore, each publisher checks that a content request is fresh before answering it, and each cache checks that a request ticket is fresh before responding. This prevents request replay attacks in the system.

Threat **T13** is addressed financially. Any rational cache has the goal of maximizing its profits by serving as many clients as it can. Thus, a cache is discouraged from following such a strategy. However, and as explained in the next section, caches

prioritize serving clients based on the reputation of the publisher for honesty. This priority drops when a publisher does not pay caches their tkt_{L2}. Reducing this priority, or even ending a work relationship with a malicious publisher, is not considered a DoS attack against the publisher's clients.

Threat **T14** is also addressed financially. Service setup is costly. After paying this cost, which includes distributing content to caches and announcing them as escrow beneficiaries, a rational publisher will choose to work with these caches to serve its clients. Even if a malicious publisher is willing to pay this cost and then abandon caches, rational caches would not be willing to wait for so long. Instead, they will switch to work with a different publisher in order to collect more profits, thus taking away any incentive to pursue this strategy.

Finally, Threat **T15** is neutralized by the escrow setup and the design of the service-payment exchange protocol in CacheCash. A publisher is committed to the content because the escrow creation transaction includes a signed hash of this content. This hash is the root of the Merkle tree computed over all data chunks comprising the content. The escrow transaction is publicly published on the blockchain, and the full Merkle tree is also made public by the publisher. This allows any cache to verify the correctness of the content copy it retrieves before starting the service. Furthermore, given that the publisher is committed to the session keys used for encrypting the data chunks in any service session, which includes the hash of each key in the request ticket, the publisher cannot pretend that a cache served incorrect content. In case of a dispute, a cache can disclose this key, along with the encrypted chunk it delivered, which allows anyone to verify the correctness of the served chunk.

5.6 Economic Analysis for Financial Defense Techniques

In this section, we present a detailed economic analysis for the financial defense mechanisms mentioned in the previous section. More concretely, we hypothesize utility functions for rational participants and examine the extent to which honest behavior

maximizes utility. Then based on this utility, we configure the system parameters in a way that makes cheating less profitable, in expectation, than acting in an honest way.

5.6.1 Analysis Setup

Our analysis focuses on how to prevent two deviant behavior cases by making them unappealing to rational participants. First, we consider the case of a rational subset of caches, possibly colluding with a client, attempting to maximize payments while minimizing the resources they spend to provide the service. Second, we explore the case of a rational publisher, possibly colluding with clients, attempting to maximize the amount of service provided to its clients while minimizing the payments made to caches.

We approach these economic attacks using a game theoretic model. We consider a repeated game in which the same one-shot game, involving the same set of entities, is played repeatedly over several time periods. In our setup, a one-shot game is a round consisting of multiple service sessions, where each session involves a publisher, a client, and a set of n caches. We track the evolution of the participants' utility gain across the rounds based on the various strategies they may follow. In particular, we compare the utility gain of a malicious actor to an honest one, and show how to make honesty the most profitable option. The set of notations used in this section include those introduced earlier in Table 5.1 and these found in Table 5.2.

5.6.2 Case 1: Rational Cache Collusion

Any rational cache has the goal of maximizing its payments while minimizing the amount of work it has to do in the system. Therefore, we define the utility function for caches as the payments minus the monetary cost of the resources expended in acquiring these payments. This can be expressed as follows:

$$utility_{cache} := payments - resources_{cost} \tag{5.16}$$

Table 5.2: Notations II.

Symbol	**Meaning**
C	Honest cache.
$\hat{C}$	Malicious cache.
D	Data chunk.
D_{cost}	The monetary cost of the bandwidth amount expended when serving a data chunk D.
S_i^j	Number of service sessions assigned to cache C_j in round i.
$\mathbb{Z}$	Random variable that models the number of winning tkt_{L2} tickets.
z	The value of winning tkt_{L2} counter.

For each service session in each round of the repeated game, a cache decides which strategy to follow. The set of all cache strategies includes the following:

- Behave honestly.
- Collude with the client to collect tkt_{L1} without doing any work (**T3** listed in the previous section).
- Collude with the client to collect tkt_{L2} without delivering the full content (**T4** listed in the previous section).

Recall that the third strategy is addressed by the security of CAPnet **??**. Thus, we focus on making the second cheating strategy unprofitable when compared to acting honestly.

Let's suppose a rational cache has been behaving honestly so far and achieved a counter value of z_{i-1} at round $i-1$. Let $\mathbb{Z}_i^j$ be a random variable that represents the counter value for an honest cache C_j at round i. C_j will behave honestly in all S_i^j sessions in round i, which results in the following expected utility gain for this honest cache:

$$\mathbb{E}[u(C_j)] = pS_i^j\mathbb{E}[f(\mathbb{Z}_i^j)] - S_i^j D_{cost} \tag{5.17}$$

The first term represents the expected payments this cache would collect in round i, which is computed as the expected number of winning tkt_{L1} (i.e., pS_i^j) multiplied

by the expected currency value of each of these tickets (i.e., $\mathbb{E}[f(\mathbb{Z}_i^j)]$)[13]. While the second term represents the total monetary cost of the resources this cache expends to serve S_i^j data chunks.

On the other hand, a malicious cache $\hat{C}_j$ may apply the second strategy, i.e., collect only tkt_{L1} without delivering any content, in x sessions out of S_j. Hence, it collects S_i^j tickets of type tkt_{L1}, $S_i^j - x$ tickets of type tkt_{L2}, and saves the cost of serving x data chunks. Let $\hat{\mathbb{Z}}_i^j$ be a random variable that represents the counter value for this malicious cache at round i, such that $\hat{\mathbb{Z}}_i^j < \mathbb{Z}_i^j$. This results in the following utility gain for a malicious cache in round i:

$$\mathbb{E}[u(\hat{C}_j)] = pS_i^j \mathbb{E}[f(\hat{\mathbb{Z}}_i^j)] - (S_i^j - x)D_{cost} \tag{5.18}$$

In order to make the honest behavior more profitable than cheating, we require that $\mathbb{E}[u(C_j)] \geq \mathbb{E}[u(\hat{C}_j)]$, which produces the following:

$$\mathbb{E}[f(\mathbb{Z}_i^j)] - \mathbb{E}[f(\hat{\mathbb{Z}}_i^j)] \geq \frac{xD_{cost}}{pS_i^j} \tag{5.19}$$

Accordingly, the function $f(.)$ must be of an appropriate growth rate to satisfy the above condition. We can achieve this, for example, by choosing f to be an appropriate linear function $f(z) := az$ for some positive constant a. In what follows, we derive a lower bound for the constant a in order to have a fully defined payment function.

By linearity of expectation, we have:

$$\mathbb{E}[f(\mathbb{Z}_i^j)] = \mathbb{E}[a\mathbb{Z}_i^j] = a(z_{i-1} + pS_i^j)$$

$$\mathbb{E}[f(\hat{\mathbb{Z}}_i^j)] = \mathbb{E}[a\hat{\mathbb{Z}}_i^j] = a(z_{i-1} + p(S_i^j - x))$$

Intuitively, both an honest and a malicious cache start round i with a counter value of z_{i-1}. During round i, an honest cache collects S_i^j tickets of type tkt_{L2}, among which pS_i^j tickets are expected to win. A malicious cache, on the other hand,

[13]As discussed in Section 5.4.1, the winning events of each ticket type are independent, and can be modeled as random variables each follows a binomial distribution.

collects $S_i^j - x$ tickets of type tkt_{L2}, and hence, its counter is expected to increase by $p(S_i^j - x)$.

By substituting the above quantities in equation 5.19, we obtain the following bound for a:

$$a \geq \max_{\substack{j \in \{1,\ldots,N\} \\ i \in \{1,\ldots,l_{esc}\}}} \frac{D_{cost}}{p^2 S_i^j} \tag{5.20}$$

That is, the lower bound for a is the one computed based on the least number of assigned sessions to any cache in all rounds of an escrow lifetime (such that $S_i^j > 0$).

By using $f(\cdot)$ with a constant a the respects the above bound, the expected utility gain in future rounds cannot be improved by lower counter values from early rounds. Thus, there is never an incentive for a cache to deviate from honest behavior and collect only tkt_{L1} in some service sessions. This will motivate a rational cache to work faithfully and collect both ticket types in every service session in which it participates.

Computing the currency value of a winning tkt_{L1} based on an accumulative counter for winning tkt_{L2} means that the service a cache provides becomes more expensive with time. This may motivate a rational publisher to leave the system when the caches' counters reach large values, and then restart again with a new set of caches with zero counters. For this reason, we suggest resetting these counters for each new escrow a publisher creates.

5.6.3 Case 2: Rational Publisher Collusion

The goal of a rational publisher is to serve its clients correctly while paying as little to caches as possible. Therefore, we define the utility function for any publisher as follows:

$$utility_{publisher} := service_{cost} - payments \tag{5.21}$$

For each service session in each round of the repeated game defined earlier, a rational publisher chooses a strategy to follow, either on its own or in collusion with the client or even other caches. The strategy space for this publisher includes the following:

- Act honestly.
- Do not pay caches at all by issuing invalid tickets (**T5**), issuing tickets that will not be covered by the escrow, such as those with out-of-range or duplicated sequence numbers (**T7**), withdrawing the payment escrow before paying these caches (**T6**), or manipulating the lottery to make the issued tickets worth nothing (**T8** or **T9**).
- Do not pay caches tkt_{L2} (**T10** or **T11**).

As explained in the previous section, the second strategy class is addressed by the secure design of MicroCash. Consequently, we are left with the third strategy where it suffices to compare its utility gain to the honest case and show how to make it unprofitable.

Suppose that a rational publisher has been acting honestly until round $i-1$. Let $\mathbb{Z}_i^j$ be a random variable that represents the counter value for cache C_j at round i. By acting honestly in the next i^{th} round, the expected utility gain of this publisher can be expressed as follows:

$$\mathbb{E}[u(P)] = \sum_{j=1}^{N} \left(S_i^j D_{cost} - p S_i^j \mathbb{E}[f(\mathbb{Z}_i^j)] \right) \tag{5.22}$$

A cheating publisher, on the other hand, may not hand a cache C_j its tkt_{L2} in x_j sessions out of S_i^j, such that $x_j > 0$. This will reduce the payment value for this cache because the expected value of its winning tkt_{L2} counter will be lower than the honest case. Let $\hat{\mathbb{Z}}_i^j$ be a random variable that represents the counter value for cache C_j at round i when considering a malicious publisher, where $\hat{\mathbb{Z}}_i^j < \mathbb{Z}_i^j$. In this case, the expected utility gain of a malicious publisher can be computed as:

$$\mathbb{E}[u(\hat{P})] = \sum_{j=1}^{N} \left(S_i^j D_{cost} - p S_i^j \mathbb{E}[f(\hat{\mathbb{Z}}_i^j)] \right) \tag{5.23}$$

By comparing the two quantities in equations 5.22 and 5.23, we find that $\mathbb{E}[u(\hat{P})] > \mathbb{E}[u(P)]$. This means that if caches continue working with the malicious

publisher despite losing tkt_{L2}, applying the third strategy, i.e., not handing out tkt_{L2} to caches, is always more profitable than acting in an honest way.

Accordingly, the main defense taken by honest caches is to reduce the amount of service given to this publisher until they eventually leave its network. This will incur a loss on the publisher's side because it will have to pay the service setup cost for constructing a new network of caches. Furthermore, this publisher could lose its clients due to the reduced amount of service delivered by caches.

We need to characterize how a cache C_j will decide to leave or reduce service for any given publisher. This decision is affected by several factors including the current counter value of winning tkt_{L2}, the clients' request rate (i.e., load assigned by the publisher to a cache), and the probability that this publisher will act honestly in future service sessions, which determines the expected winning tkt_{L2} counter in the future.

Based on that, we suppose that each cache computes a priority for each publisher based on the expected profit it would collect by working with this publisher. A cache then answers requests coming from each publisher's clients with a probability proportional to the expected service profit.

With the assumption that service sessions are independent, the service profit of a publisher can be computed as follows. We start with computing the profit in a single round. A cache C_j has a current counter value z_i^j in round i, and it expects to participate in S_{i+1}^j sessions in the next round. It will collect S_{i+1}^j tickets of type tkt_{L1}, while the number of tkt_{L2} tickets could take any value in $\{0, 1, \ldots, S_{i+1}^j\}$, based on the honesty of the publisher. If a cache expects that a publisher will act honestly with probability w (i.e., it will get a tkt_{L2} in a session with probability w), then the number of tkt_{L2} it collects in S_{i+1}^j sessions is a random variable that follows a binomial distribution. Hence, the expected number of tkt_{L2} a cache may receive in the next round is wS_{i+1}^j, and the expected number of winning tkt_{L2} in that round will be pwS_{i+1}^j. This makes the counter value a cache owns to have an expected value of $z_i^j + pwS_{i+1}^j$.

Accordingly, the profit in round $i+1$ when working with the y^{th} publisher can be

computed as follows:

$$profit_{y,round} = pS^j_{i+1} f(z^j_i + pwS^j_i) = apS^j_{i+1}(z^j_i + pwS^j_{i+1}) \tag{5.24}$$

That is, a cache will get on average pS^j_{i+1} winning tkt_{L1}. Each of these tickets will have an expected currency value equals to $a(z^j_i + pwS^j_{i+1})$.

As mentioned above, equation 5.24, computes the profit for single round. A cache can repeat this process by computing the profit for the rest of the rounds in the publisher's escrow lifetime. This can be done by applying equation 5.24 after updating the initial counter z^j_i properly based on the prior rounds.

In order to estimate w, we suppose that a cache relies on the number of tkt_{L2} it receives from a publisher. In detail, a cache may adopt a weighted average policy to estimate this probability. During a specific time window, like several rounds in the system, a cache records the number of sessions it served for a publisher, denoted as S', and the number of tkt_{L2} it received, denoted as s'. Then it computes the new estimated probability as (where γ is some weighting factor such that $0 \leq \gamma \leq 1$):

$$w_{new} = \gamma \frac{s'}{S'} + (1 - \gamma) w_{old} \tag{5.25}$$

After computing the expected profit from the work with each publisher, a cache will give highest priority to client requests paid by the publisher with the highest profit value, and so on. If the work profit with some publisher is very low, while working with another, possibly new publisher is more profitable, a cache may leave this publisher and switch to a new one. Therefore, rational publishers will act honestly and pay caches their tkt_{L2} to avoid losing their caches, and possibly their clients if the service get disconnected for long periods.

5.7 Performance Evaluation

In order to understand the performance benefit of our system, this section evaluates the computation, bandwidth, and payment setup costs of CacheCash. To do this, we

conduct empirical experiments to answer the following questions:

- How fast can a publisher or a cache serve content to clients? And how fast a client can retrieve this content?
- What is the bandwidth overhead of serving and retrieving content?
- What is the size of the data logged on the blockchain?
- What do these numbers mean for a practical deployment?

The rest of this section describes our methodology and discusses the significance of the obtained results.

5.7.1 Methodology

To establish our microbenchmarks, we implemented the functions used by a publisher, a cache, and a client to serve/retrieve content. This includes all the routines used for issuing a content request, generating a ticket bundle, and performing the data chunk retrieval protocol between a client and a cache.

We computed the performance metrics of interest as follows. We measured the computation cost as the bitrate at which a publisher or a cache can serve content, and the bitrate at which a client can retrieve content. For the bandwidth overhead, we computed the total amount of data these parties exchange to deliver the service, other than the data chunks, such as content requests, ticket bundles, lottery tickets, etc. In addition, we report on the size of escrow transactions and lottery ticket redemption that publishers and caches record on the blockchain.

Our experiments were implemented in C, and conducted on a modest publisher/cache server with an AMD Ryzen 3 2200G processor and 16 GB of memory, and a low-end client machine with an Intel Core i7-4600U processor and 8 GB RAM. We used SHA256 for hashing, AES-CBC for the PRFs, AES-CTR for encryption, and EdDSA (over Ed25519 curve) for digital signatures. Each of the tested functions was called at least 10^6 times on the publisher/cache side, and 10^3 times on the client side.

Unless otherwise mentioned, all graphs consider the case of popular content where clients request similar content over close time intervals. They also use a chunk size of 1 MB, $n = 4$ caches, 5 puzzle rounds and piece size of 16 bytes for CAPnet cache accountability puzzle, and a batch signature covering 128 ticket bundles in each batch.

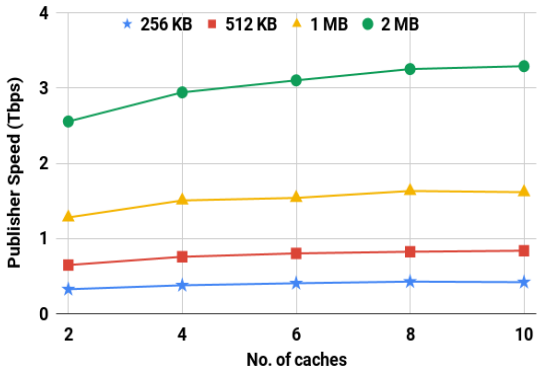
5.7.2 Results

Service Speed

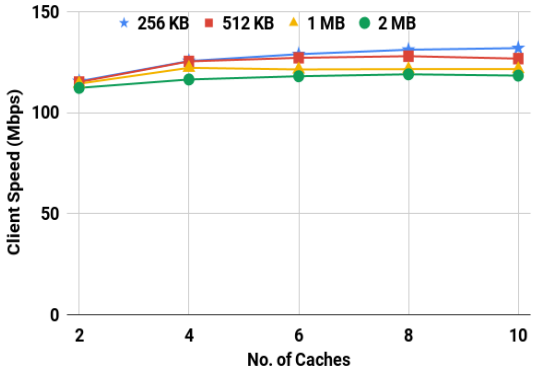
We start by quantifying the computation cost of content retrieval in CacheCash. We study the effect of several parameters including data chunk size, number of caches per request n, and employing batch signatures. We report the bitrate while assuming that all entities have unlimited bandwidth to communicate with each other.

How does data chunk size and n impact content bitrate? Figure 5.5 shows the bitrate for a publisher, cache, and a client when varying the number of caches in a service session and the chunk size. As shown, the chunk size has a large effect on the performance of a publisher and a cache, but minimal effect on a client's performance. There are several reasons for this difference. For a fixed number of caches, the publisher has almost a fixed cost for preparing a ticket bundle regardless of the chunk size. This is because it processes the same number of pieces when generating a puzzle, and generates the same number of request and lottery tickets. Consequently, a larger chunk size makes the amount of content retrieved per ticket bundle larger. On the other hand, for a fixed chunk size, increasing n reduces the number of ticket bundles a publisher can generate per second. Larger n increases the puzzle generation cost and the number of tickets to be generated per bundle. Nonetheless, multiplying the request processing rate with larger number of data chunks produces the same bitrate as the case for smaller n values.

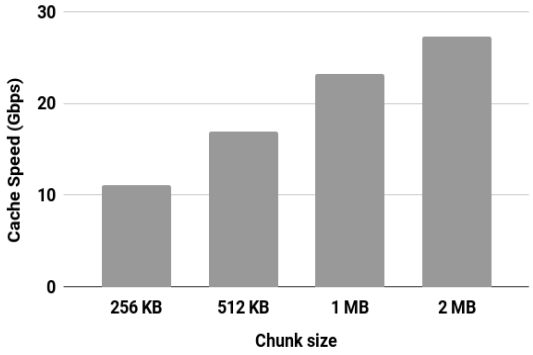
Alternatively, for a client (as depicted in Figure 5.5b), a larger chunk size for a fixed n, or larger n for a fixed chunk size, mean more computational cost to solve the puzzle, and larger amount of data to be decrypted. This allows the client to complete smaller number of content requests per second than when using small data chunk size



(a) Publisher speed (Tbps).



(b) Client speed (Mbps).



(c) Cache speed (Gbps).

Figure 5.5: Service speed for various data chunk sizes and n values.

or smaller n. Similar to the trend we saw in Section 3.5, when computing the bitrate for some n value, the low request completion rates are multiplied by large chunk size and vice versa. The same for the case of a fixed chunk size, larger n means more data is retrieved per request compared to smaller n values. For this reason, the client bitrate is somewhat similar for all chunk sizes and n values.

A cache speed, as depicted in Figure 5.5c, increases as the chunk size increases[14], where serving 2 MB chunks allows a bitrate of around 2.4x the one when 256 KB chunks are used. Although a cache becomes slower in terms of request rate when a chunks size increases, because a larger amount of data need to be double encrypted, the larger chunk size makes the amount of content served using these requests larger.

How does employing batch signing affect content bitrate? In what follows, we quantify the impact of one of CacheCash performance optimization techniques,

[14]Recall that n does not affect a cache performance, a cache always serves one data chunk per session regardless of n.

Table 5.3: Effect of batch signing on service speed. The number that comes after the label batching indicate the batch size.

Signing Scheme	**Publisher (Tbps)**	**Cache (Gbps)**	**Client (Mbps)**
Individual Tickets	0.064	11.34	121.92
Individual Bundles	0.46	23.55	122.56
Batching (64)	1.43	23.5	121.92
Batching (128)	1.51	23.24	122.24
Batching (256)	1.44	23.45	121.28
Batching (512)	1.48	23.35	120.64
Batching (1024)	1.45	23.47	121.28

namely, batch signing. As mentioned previously, and given that a cache needs to verify each ticket it receives, one option for the publisher is to sign each ticket individually inside a ticket bundle. The first optimization technique that CacheCash implements is to sign the whole bundle once while sending the individual ticket hashes to a cache to allow verifying this signature over each ticket (see Section 5.3.4). The second technique is to apply batch signing across bundles, where all bundles generated within a batch period are covered by one signature (see Appendix B).

In order to quantify the efficiency impact of batch signing, we measure the content bitrate for the parties in the system under various signing scenarios. This includes individual ticket signing, individual bundle signing, and batch signing with various batch sizes (i.e., number of bundles covered by a single batch signature). The results are found in Table 5.3.

As the table shows, signing ticket individually degrades the bitrate for both publishers and caches. This is because more signatures are generated and verified per content request, which is the heaviest computational operation these parties perform. A client, on the other hand, has a stable bitrate regardless of the signature scheme. This is due to the fact that puzzle solving is what drives the load at the client side. Optimizing the signature scheme optimizes a very small amount of this computational load, which produces an unnoticeable effect as the results demonstrate.

Reducing the number of signatures improves a publisher's bitrate. Signing a

bundle, instead of individual tickets, boosts the publisher performance by around 7.2x. Furthermore, signing a batch, instead of individual tickets or individual bundles, boosts the performance by around 22.7x and 3.2x, respectively. As shown, a larger batch size does not improve the speed further, where for all tested batch sizes the average bitrate is around 1.46 Tbps. This is because a larger batch size means larger Merkle tree to be constructed, and higher number of membership paths in this tree to be generated (each bundle needs a membership path). This additional cost balances the cost saving obtained by the reduced number of digital signatures for large batch sizes.

The trend is different for a cache, where an improvement by around 2.1x is gained when signing bundles instead of individual tickets. However, batch signing does not enhance the service speed. Although a publisher signs multiple bundles all at once, a cache has to verify a signature for each request it receives regardless of the batch size. For both bundle and batch signing, this corresponds to finding matching hashes of the tickets inside h_{bundle} and verifying one signature. For this reason, a cache has a very close bitrate for all batch sizes as the one obtained for individual bundle signing.

Contextualizing our results. To ground our results in real world numbers, we examined the customer demand from the popular content provider Netflix.com. Netflix serves 1080p video at a bitrate of approximately 5 Mbps [37]. As shown in Figure 5.5, and taking $n = 4$ caches and a chunk size of 1 MB, a publisher in our setup, using a single core machine, can process 49,341 requests per second, which translates to 197,364 data chunks per second. To understand this load, we look at a popular show, "House of Cards", where 5.4 million of its 83 million subscribers (as of 2015) watched at least one episode within a month of its release [29]. Since the report does not indicate how many of those views were concurrent, it is not possible to infer the exact peak load. However, our single core publisher supports concurrent viewing from 315,780 clients which is enough to support a simultaneous viewing peak from 1/17 of the 5.4 million customers at any point during the first month.

As for the client, the previous results showed that on average our low-end client is able to retrieve at least 122 Mbps using a single core (Figure 5.5b). This is more

than 24 times the rate required to retrieve the same popular 1080p video [37].

Key Takeaway: CacheCash has a low a computational cost that enables a modest, single core publisher or cache to serve content at a rate of several Tbps and several Gbps, respectively, allowing a client to watch dozens of 1080p videos concurrently.

Size of Data Logged on the Blockchain

In CacheCash, two types of transactions are logged on the blockchain, escrow creation and ticket redemption transactions. As mentioned before, CacheCash adopts a modified version of MicroCash, where it adds two types of tickets and more information to the escrow creation transaction (e.g., signed hash of the content). Furthermore, a signed lottery ticket has a copy of a batch signature that includes the full hash of the bundle that contains this lottery ticket, the membership path of the bundle in the batch Merkle tree, the tree root, and the signature over this root.

To reduce the size of the additional information to be logged on the blockchain, CacheCash introduces several optimizations. These include reducing the ticket sizes by including a cache index instead of its public key in a lottery ticket. Logging only one winning tkt_{L2} among the n copies that caches receive. And logging one batch signature on the blockchain for all winning lottery tickets that belong to the same bundle.

Based on that, we compute the delta blockchain size that CacheCash incurs, and compare it to MicroCash's overhead, for the same CDN example found in Section 4.7.3. We use the same setup for p, data chunk size, average transaction size, and number of beneficiary caches. We consider a batch size of 32 bundles (this is the number of content requests issued per second in that example). For MicroCash, we consider signatures and keys using EdDSA scheme. The results are shown in Table 5.4.

Starting with escrows, both MicroCash and CacheCash allow a publisher to create one escrow per day (as in our example). However, an escrow creation transaction in CacheCash is larger, which is around 1,835 byte as compared to 1,738 bytes in MicroCash. This is reflected on the slightly higher bandwidth cost between a publisher

Table 5.4: Size of data logged on the blockchain (a round is 10 minutes).

Metric	**MicroCash**	**CacheCash**
Escrows / sec	0.00001157	0.00001157
Tickets / sec	128	160
Winning tickets / sec	0.001964	0.002456
Transactions / sec	0.001976	0.00394
Bandwidth between publisher and miners	0.16093 bps	0.1699 bps
Bandwidth between caches and miners	6 bps	27.3 bps
Delta blockchain size / round	0.00044 MB	0.00123 MB

and the miners.

For the tickets issuance rate, a publisher in CacheCash issues more tickets, an additional 32 tkt_{L2} tickets. Each of these tickets is replicated among $n = 4$ caches, where in the worst case each cache will claim this ticket if it wins. Hence, the number of issued claim transactions in CacheCash is twice what will be produced in MicroCash. Also, each of these transactions will have a copy of the batch signature. For this reason, the bandwidth cost between caches and miners is significant (CacheCash incurs around 3.9x the cost in MicroCash).

However, not all these transactions are logged on the blockchain as mentioned earlier. Only one copy of a winning tkt_{L2} will be logged, and for all winning tickets that belong to the same bundle only one batch signature with be recorded on the blockchain. As such, a batch signature over a specific bundle will make to the blockchain is any of the lottery tickets inside the bundle wins, which happens with probability $1 - (1 - p)^n$. As Table 5.4, although CacheCash adds around 4.6x the amount of that MicroCash adds to the blockchain, the cost is still very low. It requires a space of around 0.13% of a block size in Bitcoin per round.

Key Takeaway: The two-ticket payment model in CacheCash adds more bandwidth cost than MicroCash, however, the overall cost is very low, less than 0.13% of a block size in Bitcoin.

Service Bandwidth Overhead

In terms of bandwidth overhead, we computed the total amount of data the parties exchange for the service other than the data chunks. This includes the content request messages, the ticket bundles, the request/lottery tickets and encryption key messages a client and a cache exchange, transactions with the miners, etc. Then, we computed the overhead ratio as overhead bytes divided by total bytes exchanged (which includes both overhead and data chunks) using the example setup used in the previous subsection. Our results shows that the overhead is less than 0.1%.

Key Takeaway: CacheCash incurs a very small bandwidth overhead, less than 0.1% of the total bandwidth needed for the service. This suggests that CacheCash would not degrade productivity of content distribution applications.

5.8 Conclusion

This chapter introduced CacheCash, a cryptocurrency-based, fully-decentralized content delivery service. CacheCash combines the flexibility of P2P networks with cryptocurrency payments to create a distributed bandwidth market. This market allows publishers to create dynamic CDNs at a low deployment cost, without sacrificing security nor efficiency. This is done by introducing a secure service-payment exchange protocol that combines cache accountability puzzles with a two-ticket micropayment model, to reduce the monetary losses caused by malicious actors.

The chapter also presented detailed security analysis demonstrating how malicious behaviors are neutralized using both cryptographic approaches and rational financial incentives. It also described a series of empirical experiments testing CacheCash's performance under various configurations. The results showed how our system can support content delivery at a high bitrate with a minimal bandwidth overhead. This indicates the potential of CacheCash as a practical solution to support an open access and fully distributed CDN service at large scale.

Chapter 6

Conclusion

6.1 Closing Remarks

The evolution of cryptocurrencies and blockchain technologies has provided potential new templates for reshaping large-scale distributed systems and services. Cryptocurrencies implement a decentralized virtual currency exchange medium that permits participants to be rewarded without any pre-authentication or identification requirements. Furthermore, they provide public verifiability and auditing without the need to place trust in any party. Such features can be exploited in P2P-based resource exchange paradigms to creating a market that rewards for the correct service without driving the underlying system toward centralization.

This work focused on one type of such services; online content delivery. The escalating demand for this service, which now accounts for more than half the Internet's bandwidth, drove content publishers to look for non-traditional solutions, such as peer-assisted models. The goal is to reduce costs, while enjoying higher flexibility than can be offered by traditional infrastructure-based CDNs. By providing monetary incentives, trustless peers, or caches, are encouraged to serve others and comply with protocol specifications. Such an approach is beneficial for both publishers and end users (or caches). Publishers obtain service at a cheaper price and can hire caches in areas that are not covered by CDN providers. In return, end users can exploit their idle resources to collect payments.

Though this idea has been around for a long time, even before the emergence of cryptocurrencies, all prior works and deployed solutions suffered from various drawbacks that restrained practicality. These include assuming the existence of a central-

ized party to control the system or placing trust in content publishers. In addition, these systems lack effective defenses against critical attacks and collusion cases, and the needed flexibility to meet specific delivery configurations, such as allowing publishers to sponsor content retrieval for their clients.

In this dissertation, we proposed CacheCash, the first fully decentralized, cryptocurrency-based, CDN system that addresses these problems. CacheCash permits caches to come and go as demand shifts, and enables publishers to hire these caches on "as needed" basis. This is done without constraining any of these parties with long term commitments. CacheCash builds upon various components and protocols that were developed to satisfy the security and efficiency goals of the system. Its design is guided by a thorough threat model built using our cryptocurrency-oriented ABC framework. CacheCash's service-payment exchange protocol utilizes CAPnet's accountability puzzles to defend against cache accounting attacks. This protocol also implements an efficient probabilistic micropayment scheme, a modified version of MicroCash, to reward caches. The system exploits various game theoretic and cryptographic approaches to secure its operation and encourage honest behavior based on an economic perspective.

In terms of efficiency, CacheCash builds a content distribution service that allows a publisher to serve content at a rate sufficient to support 315,780 concurrent clients watching the same 1080p video. It also enables a cache to serve data chunks at a bitrate of several Gbps (capped by the upload bandwidth speed such a cache has), and a client to retrieve content at a rate of 122 Mbps, enough to watch around 24 1080p videos simultaneously. This shows the potential of our system as a viable content delivery solution for large-scale applications.

6.2 Future Directions

While CacheCash resolves the security and efficiency issues elaborated at the beginning of this thesis, it would benefit further from few design optimizations that could promote its cost-effectiveness, and address issues such as user privacy. This

is in addition to providing a full implementation of the system. These optimization and deployment plans shape our future work directions, which we outline below.

Service price and collateral cost. In Chapter 5, we derived a payment function that is used to compute the currency value of a winning tkt_{L1}. Such a function, although it is linear in the winning tkt_{L2} counter value for a single ticket, is quadratic when used to compute the total payout a cache obtains for all its winning tkt_{L1} tickets. An interesting direction is to explore alternative payment function expressions that make the service price grow at a slower rate with the caches' load. In other words, we seek to find relations that are less sensitive to load distribution, and hence, lower service price inflation. This makes it more viable for publishers to provision service cost and create long lifetime escrows.

Another interesting aspect is to optimize the collateral cost of the service, i.e., balances of both payment and penalty escrows. One such approach is to devise an alternative lottery protocol that bounds the number of wining tickets per round. More concretely, we want to eliminate the chance that more tickets than expected will win. Beside developing such a lottery technique, an in-depth analysis is needed to derive the bounds for both B_{escrow} and $B_{penalty}$ based on the winning behavior of tickets under this new lottery protocol.

User privacy. The current version of our system does not preserve user privacy. Publishers can track their client activities and content watching habits. Similarly, caches can track what content each client is interested in. One of our future directions is to investigate how to enhance user privacy in CacheCash while maintaining its low overhead.

Fault tolerance and cache selection. Another future direction of our work is to investigate how faulty caches affect the service that clients receive. That is, what should a client/publisher do when some data chunks in a service session are not received (or corrupted)? Should a client just start a new session and

request all chunks again? We aim to find more optimized approaches that allow clients to request only missing chunks. This also involves developing probing and feedback, or reputation building, mechanisms, which can allow collecting performance information that help in discovering faulty, or poor performing caches early on. Such a mechanism would also enable publishers to set up criteria to select caches for content requests in a way that controls the quality of service.

System deployment. Until now, we have only built a prototype for the data service protocol of CacheCash. For the payment service, we intend to modify the Bitcoin's protocol by adding all the new transaction types, and processing logic, that our system introduces. Our next step is to assess this plan and either go with the Bitcoin protocol, adopt an alternative model, such as Ethereum [131], or even build our own network protocol, based on lessons learned from the operation of other cryptocurrencies. The latter is motivated by an initial vision of developing a new consensus protocol that ties mining to the CDN service a cache puts into the system. This promotes the notion of useful mining, rather than having the miners perform useless computations. It also offers a chance to build a decentralized mining network that discourages the formation of mining pools.

Bibliography

[1] *$257 Million: Filecoin Breaks All-Time Record for ICO Funding.* https://www.coindesk.com/257-million-filecoin-breaks-time-record-ico-funding/.

[2] *ABC Supplemental Material.* https://ssl.engineering.nyu.edu/papers/abc-material.zip.

[3] *Akamai.* https://www.akamai.com/.

[4] *Akamai NetSession.* https://www.akamai.com/us/en/products/media-delivery/netsession-interface-overview.jsp.

[5] *Amazon CloudFront Pricing.* https://aws.amazon.com/cloudfront/pricing/.

[6] *At&t Network Averages.* https://ipnetwork.bgtmo.ip.att.net/pws/averages.html.

[7] *Average Credit Card Processing Fees.* https://www.cardfellow.com/blog/average-fees-for-credit-card-processing/.

[8] *Bandwidth upload speed in the US.* https://www.recode.net/2017/9/7/16264430/fastest-broadband-speeds-ookla-city-internet-service-provider.

[9] *Bitcoin mining pools.* https://blockchain.info/pools.

[10] *Bitcoin transaction fees.* https://bitinfocharts.com/bitcoin/.

[11] *BitcoinCash.* https://www.bitcoincash.org/.

[12] *BitcoinGold.* https://bitcoingold.org/.

[13] Bitcoinj. https://bitcoinj.github.io/working-with-micropayments.

[14] *The Bitfinex Bitcoin Hack: What We Know (And Don't Know).* https://www.coindesk.com/bitfinex-bitcoin-hack-know-dont-know/.

[15] *Bitfloor Hacked, $250,000 Missing.* https://bitcoinmagazine.com/articles/bitfloor-hacked-250000-missing-1346821046/.

[16] *BitInfoCharts, Bitcoin avg. transaction fee.* https://bitinfocharts.com/comparison/bitcoin-transactionfees.html.

[17] *Bitstamp Claims $5 Million Lost in Hot Wallet Hack.* https://www.coindesk.com/bitstamp-claims-roughly-19000-btc-lost-hot-wallet-hack/.

[18] *BloXroute: A Scalable Trustless Blockchain Distribution Network.* https://bloxroute.com/wp-content/uploads/2018/03/bloXroute-whitepaper.pdf.

[19] *Board of Governers of the Federal Reserve System, press release June 2011.* https://www.federalreserve.gov/newsevents/pressreleases/bcreg20110629a.htm.

[20] *Board of Governers of the Federal Reserve System, Regulation II.* https://www.federalreserve.gov/paymentsystems/regii-about.htm.

[21] *Box plot diagrams.* https://www.itl.nist.gov/div898/handbook/eda/section3/boxplot.htm.

[22] *Cisco Visual Networking Index: Forecast and Methodology, 2016-2021.* https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html.

[23] *CloudFront.* https://aws.amazon.com/cloudfront/.

[24] *CryptoCurrency Market Capitalizations.* https://coinmarketcap.com/.

[25] *EIP 665: Add precompiled contract for Ed25519 signature verification.* https://eips.ethereum.org/EIPS/eip-665.

[26] *GameServers: Minecraft.* https://www.gameservers.com/game_servers/minecraft.php.

[27] *Golem.* https://golem.network/.

[28] *Hackers Stole $32 Million in Ethereum*. https://thehackernews.com/2017/07/ethereum-cryptocurrency-hacking.html.

[29] *House of Cards Viewers Stats*. http://fortune.com/2016/03/05/house-of-cards-viewership/.

[30] "lightning network will be highly centralized". https://cointelegraph.com/news/lightning-network-will-be-highly-centralized-gavin-andresen.

[31] *MicroCash and CacheCash threat models*. https://drive.google.com/drive/folders/1dHcHpMFq1BNolKLM4rR0qzhw7RSv6Hhr?usp=sharing.

[32] *Microsoft Threat Modeling Tool 2016 User Guide*. https://www.microsoft.com/en-us/download/details.aspx?id=49168.

[33] *Minecraft*. https://minecraft.net/en-us/.

[34] *Monero: Privacy in the blockchain*. https://eprint.iacr.org/2018/535.pdf.

[35] *Mysterium*. https://mysterium.network/.

[36] *Netflix*. https://www.netflix.com/.

[37] *Netflix Internet Connection Speed Recommendations*. https://help.netflix.com/en/node/306.

[38] *NOIA*. https://www.noia.network.

[39] *NSEC5-Crypto*. https://github.com/fcelda/nsec5-crypto.

[40] *Ripple*. https://ripple.com/.

[41] *Ripple to use Ed25519*. https://ripple.com/dev-blog/curves-with-a-twist/.

[42] *Signatures in Stellar*. https://www.stellar.org/developers/guides/concepts/multi-sig.html.

[43] *Steemit Hacked for '$85,000' as Users Complain of Weak Security*. https://news.bitcoin.com/steemit-hacked-weak-security/.

[44] *TradeBlock: Analysis of Bitcoin Transaction Size Trends.* https://tradeblock.com/blog/analysis-of-bitcoin-transaction-size-trends.

[45] *Understaning the DAO Attack.* https://www.coindesk.com/understanding-dao-hack-journalists/.

[46] *Veritaseum Founder Claims $8 Million in ICO Tokens Stolen.* https://www.coindesk.com/veritaseum-founder-claims-8-million-ico-token-stolen/.

[47] *Video Games Development Costs.* https://en.wikipedia.org/wiki/List_of_most_expensive_video_games_to_develop.

[48] *Youtube.* https://www.youtube.com/.

[49] *Filecoin: A Cryptocurrency Operated File Storage Network*, 2017. https://filecoin.io/filecoin.pdf.

[50] ... Capnet: A defense against cache accounting attacks on peer-assisted cdns. 2019.

[51] Paarijaat Aditya, Mingchen Zhao, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, and Bill Wishon. Reliable client accounting for p2p-infrastructure hybrids. In *NSDI'12*, 2012.

[52] Christopher J Alberts and Audrey Dorofee. *Managing information security risks: the OCTAVE approach*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[53] Ghada Almashaqbeh, Allison Bishop, and Justin Cappos. Abc: A cryptocurrency-focused threat modeling framework. In *IEEE CryBlock*, 2019.

[54] Ghada Almashaqbeh, Allison Bishop, and Justin Cappos. Abc: A cryptocurrency-focused threat modeling framework. *arXiv preprint arXiv:1903.03422*, 2019.

[55] Ghada Almashaqbeh, Allison Bishop, and Justin Cappos. Cachecash: A cryptocurrency-based decentralized content delivery network. In *Under submission*, 2019.

[56] Ghada Almashaqbeh, Allison Bishop, and Justin Cappos. Microcash: Practical concurrent processing of micropayments. In *Under submission*, 2019.

[57] Ghada Almashaqbeh, Kevin Kelley, Allison Bishop, and Justin Cappos. Capnet: A defense against cache accounting attacks on content distribution networks. In *IEEE CNS*, 2019.

[58] Elli Androulaki, Ghassan O Karame, Marc Roeschlin, Tobias Scherer, and Srdjan Capkun. Evaluating user privacy in bitcoin. In *FC*, 2013.

[59] Nasreen Anjum, Dmytro Karamshuk, Mohammad Shikh-Bahaei, and Nishanth Sastry. Survey on peer-assisted content delivery networks. *Computer Networks*, 116, 2017.

[60] Siddhartha Annapureddy, Saikat Guha, Christos Gkantsidis, Dinan Gunawardena, and Pablo Rodriguez Rodriguez. Is high-quality vod feasible using p2p swarming? In *Proceedings of the 16th international conference on World Wide Web*, pages 903–912. ACM, 2007.

[61] Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain*. O'Reilly Media, Inc., 2017.

[62] Christina Aperjis, Michael J Freedman, and Ramesh Johari. Peer-assisted content distribution with prices. In *Proceedings of the 2008 ACM CoNEXT Conference*, page 17. ACM, 2008.

[63] Michael Backes, Aniket Kate, Praveen Manoharan, Sebastian Meiser, and Esfandiar Mohammadi. Anoa: A framework for analyzing anonymous communication protocols. In *IEEE CSF*, 2013.

[64] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan. Analyzing the potential benefits of cdn augmentation strategies for internet video workloads. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 43–56. ACM, 2013.

[65] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2), 2012.

[66] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *Annual International Cryptology Conference*. Springer, 2018.

[67] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: Research perspectives and challenges for bitcoin and cryptocurrencies. In *IEEE S&P*, 2015.

[68] Rajkumar Buyya, Mukaddim Pathan, and Athena Vakali. *Content delivery networks*, volume 9. Springer Science & Business Media, 2008.

[69] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious ram. *Journal of Cryptology*, 30(1), 2017.

[70] Alessandro Chiesa, Matthew Green, Jingcheng Liu, Peihan Miao, Ian Miers, and Pratyush Mishra. Decentralized anonymous micropayments. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 609–642. Springer, 2017.

[71] William R Claycomb and Dongwan Shin. Threat modeling for virtual directory services. In *IEEE ICCST*, 2009.

[72] Christian Decker and Roger Wattenhofer. A fast and scalable payment network with bitcoin duplex micropayment channels. In *Symposium on Self-Stabilizing Systems*, pages 3–18. Springer, 2015.

[73] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering*, 16(1), 2011.

[74] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Crypto'15*, 2015.

[75] C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security*, 17(4), 2015.

[76] Shimon Even and Yacov Yacobi. Relations among public key signature systems. Technical report, Computer Science Department, Technion, 1980.

[77] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, 2015.

[78] Arthur Gervais, Hubert Ritzdorf, Ghassan O Karame, and Srdjan Capkun. Tampering with the delivery of blocks and transactions in bitcoin. In *ACM CCS*, 2015.

[79] Mainak Ghosh, B Ford, and M Richardson. A torpath to torcoin: Proofof-bandwidth altcoins for compensating relays. 2014.

[80] Sharon Goldberg, Moni Naor, Dimitrios Papadopoulos, and Leonid Reyzin. Nsec5 from elliptic curves: Provably preventing dnssec zone enumeration with shorter responses. *IACR Cryptology ePrint Archive*, 2016:83, 2016.

[81] Prateesh Goyal, Ravi Netravali, Mohammad Alizadeh, and Hari Balakrishnan. Secure incentivization for decentralized content delivery. *arXiv preprint arXiv:1808.00826*, 2018.

[82] Matthew Green and Ian Miers. Bolt: Anonymous payment channels for decentralized currencies. Technical report, Cryptology ePrint Archive, Report 2016/701, 2016.

[83] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review*, 41(6), 2007.

[84] Ragib Hasan, Suvda Myagmar, Adam J Lee, and William Yurcik. Toward a threat model for storage systems. In *ACM StorageSS*, 2005.

[85] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security*, 2015.

[86] Mike Heran and Jeremy Spilman. Bitcoin contracts. `https://en.bitcoin.it/wiki/Contract`, 2012.

[87] Matthias Hollick, Cristina Nita-Rotaru, Panagiotis Papadimitratos, Adrian Perrig, and Stefan Schmid. Toward a taxonomy and attacker model for secure routing protocols. *ACM SIGCOMM Computer Communication Review*, 47(1), 2017.

[88] Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[89] Cheng Huang, Jin Li, and Keith W Ross. Can internet video-on-demand be profitable? In *ACM SIGCOMM Computer Communication Review*, volume 37, pages 133–144. ACM, 2007.

[90] Yichao Jin, Yonggang Wen, and Kyle Guan. Toward cost-efficient content placement in media cloud: modeling and analysis. *IEEE Transactions on Multimedia*, 18(5), 2016.

[91] Ghassan O Karame, Elli Androulaki, and Srdjan Capkun. Double-spending fast payments in bitcoin. In *ACM CCS*, 2012.

[92] Dmytro Karamshuk, Nishanth Sastry, Andrew Secker, and Jigna Chandaria. Isp-friendly peer-assisted on-demand streaming of long duration content in bbc iplayer. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 289–297. IEEE, 2015.

[93] Debessay Fesehaye Kassa and Klara Nahrstedt. Hincent: Quick content distribution with priorities and high incentives. In *IEEE CCNC'13*, 2013.

[94] Joshua A Kroll, Ian C Davey, and Edward W Felten. The economics of bitcoin mining, or bitcoin in the presence of adversaries. In *WEIS*, 2013.

[95] Qiao Lian, Zheng Zhang, Mao Yang, Ben Y Zhao, Yafei Dai, and Xiaoming Li. An empirical study of collusion behavior in the maze p2p file-sharing system. In *ICDCS'07*, 2007.

[96] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *ACM CCS*, 2016.

[97] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *ACM CCS*, 2015.

[98] Giulio Malavolta, Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Srivatsan Ravi. Concurrency and privacy with payment-channel networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[99] Silvio Micali and Ronald L Rivest. Micropayments revisited. In *Cryptographers' Track at the RSA Conference*, pages 149–163. Springer, 2002.

[100] Nikolaos Michalakis, Robert Soulé, and Robert Grimm. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *NSDI'07*, 2007.

[101] Andrew Miller, Iddo Bentov, Ranjit Kumaresan, and Patrick McCorry. Sprites: Payment channels that go faster than lightning. *arXiv preprint arXiv:1702.05812*, 2017.

[102] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

[103] Pedro Moreno-Sanchez, Navin Modi, Raghuvir Songhela, Aniket Kate, and Sonia Fahmy. Mind your credit: Assessing the health of the ripple credit network. *arXiv preprint arXiv:1706.02358*, 2017.

[104] Pedro Moreno-Sanchez, Muhammad Bilal Zafar, and Aniket Kate. Listening to whispers of ripple: Linking wallets and deanonymizing transactions in the ripple network. *PETs*, (4), 2016.

[105] Suvda Myagmar, Adam J Lee, and William Yurcik. Threat modeling as a basis for security requirements. In *SREIS*, 2005.

[106] Srijith K Nair, Erik Zentveld, Bruno Crispo, and Andrew S Tanenbaum. Floodgate: A micropayment incentivized p2p content delivery network. In *IEEE ICCCN'08*, 2008.

[107] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[108] Henning Pagnia and Felix C Gärtner. On the impossibility of fair exchange without a trusted third party. Technical report, Technical Report TUD-BS-1999-02, Darmstadt University of Technology …, 1999.

[109] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *IEEE S&P*, 2013.

[110] Rafael Pass, Lior Seeman, and Abhi Shelat. Analysis of the blockchain protocol in asynchronous networks. In *EUROCRYPT*, 2017.

[111] Rafael Pass and Abhi Shelat. Micropayments for decentralized currencies. In *CCS*, pages 207–218. ACM, 2015.

[112] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. *Technical Report (draft)*, 2015.

[113] Michael K Reiter, Vyas Sekar, Chad Spensky, and Zhenghao Zhang. Making peer-assisted content distribution robust to collusion using bandwidth puzzles. In *ICISS'09*, 2009.

[114] Ronald L Rivest. Electronic lottery tickets as micropayments. In *International Conference on Financial Cryptography*, pages 307–314. Springer, 1997.

[115] Ronald L Rivest. Peppercoin micropayments. In *International Conference on Financial Cryptography*, pages 2–8. Springer, 2004.

[116] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. In *International Conference on Financial Cryptography and Data Security*. Springer, 2016.

[117] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *IEEE SP*, 2014.

[118] Riccardo Scandariato, Kim Wuyts, and Wouter Joosen. A descriptive study of microsoft's threat modeling technique. *Requirements Engineering*, 20(2), 2015.

[119] Michael Sirivianos, Jong Han Park, Rex Chen, and Xiaowei Yang. Free-riding in bittorrent networks with the large view exploit. In *IPTPS*, 2007.

[120] Michael Sirivianos, Xiaowei Yang, and Stanislaw Jarecki. Robust and efficient incentives for cooperative content distribution. *IEEE/ACM Transactions on Networking*, 17(6), 2009.

[121] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *FC*, 2015.

[122] Steven Tadelis. *Game theory: an introduction*. Princeton University Press, 2013.

[123] Peter Torr. Demystifying the threat modeling process. *IEEE Security & Privacy*, 3(5):66–70, 2005.

[124] Axel Van Lamsweerde. Elaborating security requirements by construction of intentional anti-models. In *IEEE ICSE*, 2004.

[125] David Vandervort, Dale Gaucas, and Robert St Jacques. Issues in designing a bitcoin-like community currency. In *FC*, 2015.

[126] Vivek Vishnumurthy, Sangeeth Chandrakumar, and Emin Gun Sirer. Karma: A secure economic framework for peer-to-peer resource sharing. In *Workshop on Economics of Peer-to-Peer Systems*, volume 35, 2003.

[127] Cong Wang, Kui Ren, Wenjing Lou, and Jin Li. Toward publicly auditable secure cloud data storage services. *IEEE network*, 24(4), 2010.

[128] Patrick Wendell and Michael J Freedman. Going viral: flash crowds in an open cdn. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 549–558. ACM, 2011.

[129] David Wheeler. Transactions using bets. In *International Workshop on Security Protocols*, pages 89–92. Springer, 1996.

[130] Shawn Wilkinson, Tome Boshevski, Josh Brandoff, and Vitalik Buterin. Storj a peer-to-peer cloud storage network. 2014.

[131] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. 2014.

[132] Mingchen Zhao, Paarijaat Aditya, Ang Chen, Yin Lin, Andreas Haeberlen, Peter Druschel, Bruce Maggs, Bill Wishon, and Miroslav Ponec. Peer-assisted content distribution in akamai netsession. In *IMC'13*, 2013.

Appendix A

Deriving ABC Threat Categories

In what follows, we show how the threat categories listed in Table 2.1 were derived. We apply the procedure outlined in Step 2 of the ABC framework (Section 2.3.2) to the assets of CompuCoin, which include the service (e.g., computation outsourcing in case of CompuCoin), service rewards or payments, blockchain, transactions, currency, and communication network. For each asset, we outline its security properties in order to identify any factors that might violate these properties, which are labeled as threat categories.

Starting with the **service asset**, our analysis outlines the following:

1. *Security properties*: A secure service can be defined as the action of serving clients correctly at anytime, while providing confidentiality and binding the servers to the service they provide. Hence, the security properties of a service asset may include integrity, availability, confidentiality, and non-repudiation.
2. *Threat categories:* By negating the above properties, we find that the service asset has the following threat categories:
    - Service tampering/corruption: An attacker provides clients with invalid service or corrupts the correct service delivered by others.
    - Information disclosure: An attacker reveals the contents of service-related messages, such as the service content/outcome, the service requests sent by clients, replies sent by servers, etc.
    - Repudiation: A server denies providing a specific service or a client denies receiving it.
    - DoS: An attacker makes the service unavailable to legitimate users.

Next, we analyze the **service payment** (or rewards) asset as follows:

1. *Security properties:* One may consider the service payment asset secure as long as: a) servers are rewarded properly for the service they provide, and b) servers earn the payments they collect.
2. *Threat categories:* Negating the above security requirements produce the following threat categories:
    - Service slacking: A server collects payments without performing all the promised work.
    - Service theft: A client obtains service for a lower payment than the agreed upon amount.

For the **blockchain asset**, our analysis produces the following:

1. *Security properties:* The blockchain security properties are tied to the security of the underlying consensus protocol. These properties have been thoroughly studied in the literature [67, 77, 110]. We adopt the ones introduced in [110] with slight modifications based on the work presented in [67], which include:
    - Consistency: At any point in time, honest miners hold copies of the blockchain that have a common prefix and may differ only in the last y blocks, where y is a block confirmation parameter. A block then is confirmed once it is buried under y blocks on the blockchain.
    - Future-self consistency: At any two points in time, t_1 and t_2, the blockchain maintained by an honest party may differ only in the last y blocks. Consistency and future-self consistency properties achieve blockchain persistence or immutability.
    - Fairness: Miners collect mining rewards in proportion to the resources they expend in the mining process.
    - Correctness: All the blocks within the longest branch in the blockchain are valid. (Note that correctness and fairness represent the chain quality property outlined in [77, 110].)
    - Growth: As long as the system is functional, new valid blocks will be added to the blockchain.

2. *Threat Categories:* By negating the above properties, we can distill the following threat categories for the blockchain asset:

    - Inconsistency: Honest miners do not agree on the prefix of the blockchain copies they hold beyond the unconfirmed blocks. This also covers the case of an honest miner who does not agree with itself on the blockchain prefix it holds over time, e.g., alternating between two branches that compete in being the longest.
    - Invalid block adoption: The longest chain contains corrupted blocks that either have invalid format or contain invalid transactions.
    - Biased mining: A miner pretends to expend the needed resources to be selected to extend the blockchain and collect the mining rewards.
    - Chain freezing: The blockchain does not grow at a regular rate, but instead freezes for several contiguous rounds. This threat category is a form of DoS attack, and hence, we cover it under DoS against the communication network asset.

Next we analyze the **transaction asset** as follows:

1. *Security properties:* Secure transactions can be characterized as correct, tamper-proof, and source-binding, i.e., cannot be denied by the originator. In addition, these transactions need to be accessible to the system users at any time so they can send/receive/view transactions as needed. Moreover, these transactions must not reveal any information about the source, destination, and amount of transferred funds. Accordingly, we outline the following security properties for the transaction asset: non-repudiation, integrity, validity, availability, and anonymity. Note that the validity property is already covered by the correctness aspect of the blockchain, where a valid blockchain contains only valid transactions. Furthermore, the availability property is covered under the communication network asset.
2. *Threat categories:* Based on the previous discussion, and again by negating the aforementioned security properties, the threat categories for the transaction

asset would be:

- Repudiation: An attacker denies issuing transactions.
- Tampering: An attacker manipulates the fields of a transaction.
- Deanonymization: An attacker violates users' privacy by exploiting the public nature of the blockchain to link transactions and payments, and use this knowledge to track the activity of these users in the system and, possibly, reveal their real identities.

Next we analyze the **currency asset** as follows:

1. *Security properties:* The security properties of the currency asset are intertwined with the properties of the transaction asset. This is because the currency takes the form of digital tokens, which are the transactions exchanged in the system. Thus, they inherit all the transaction security properties. What remains is to deal with the currency ownership, meaning that only the owner can spend these tokens.
2. Threat categories: Beside the categories outlined above for the transaction asset, we have the following threat category for the currency asset:

   - Currency theft: An attacker steals currency from others in the system. This includes all currency theft attacks that are not covered by other assets. For example, biased mining is currency theft, where a miner steals others rewards indirectly, but it is already covered by the blockchain. The same holds for the service payment related threats.

Finally, we analyze the **communication network asset** as follows:

1. *Security properties:* The communication network is the backbone of any cryptocurrency system, and one that is unreliable can lead to numerous problems. First, it can create delays in propagating newly mined blocks that could produce an inconsistent blockchain. Second, it can cause delays in relaying transactions, which could reduce the transaction throughput of the system and affect its availability aspect. Third, it can slow down setting up new miners who need a longer

time to discover other peers and download copies of the blockchain. Fourth, it opens the possibility of being controlled by external parties that could intercept the communication links and isolate nodes in the network. Consequently, a secure cryptocurrency system needs a reliable and robust communication network. We merge all these aspects into one security property, namely, availability.

2. *Threat categories:* The communication network asset has one threat category, which is DoS.

Table 2.1 summarizes all the threat categories derived in this appendix. As mentioned previously, this table is by no means comprehensive. Additional threats can be added based on the asset types of the system, or more refined definitions of the asset security properties. This detailed treatment was provided as a thorough example to clarify the application of Step 2 in the ABC framework. Nonetheless, we found this table sufficiently detailed when building threat models for the systems reported as use cases in Section 2.5, including Bitcoin, Filecoin, and CacheCash, as well as for the user study tutorial as reported in Section 2.4. For this reason, Table 2.1 can be viewed as a base threat list that can be extended, or even reduced, based on the system under design.

Appendix B

Batching Content Requests

This appendix introduces one of the performance optimization techniques utilized in CacheCash, namely, batching client requests. That is, instead of replying to each client instantly as explained earlier in the paper, the publisher responds to all requests received within a predefined period as one batch. The goal is to reduce the computation cost of signing ticket bundles, where instead of signing each bundle individually, the publisher will produce one signature covering all bundles within a batch. In what follows, we discuss how a batch signature is generated and verified.

As shown in Figure B.1, for each request the publisher prepares a ticket bundle in the same way as described in Section 5.3.4 without signing. After generating all ticket bundles for all requests received within a batch period, the publisher generates a batch signature. In this process, the publisher hashes each ticket bundle individually, i.e., compute its h_{bundle}. Then, it computes the Merkle hash tree of the hashes of all bundles within the batch and signs its root to produce the batch signature σ_{batch}. Each ticket bundle will have a batch signature structure containing a copy of σ_{batch}, the membership path of the bundle in the batch Merkle tree, and the root of this tree.

For the batch signature verification, clients and caches perform this process differently. This is because a client receives the full bundle all at once but each cache receives its tickets separately while serving the client. The client starts with computing h_{bundle} of the received bundle. Then, it verifies the membership path over h_{bundle} to make sure that this bundle is part of the batch Merkle tree. Lastly, it verifies σ_{batch} over the tree root.

Each cache, on the other hand, receives a copy of the batch signature structure, in addition to h_{bundle}, with the request ticket from the client. As before, this cache

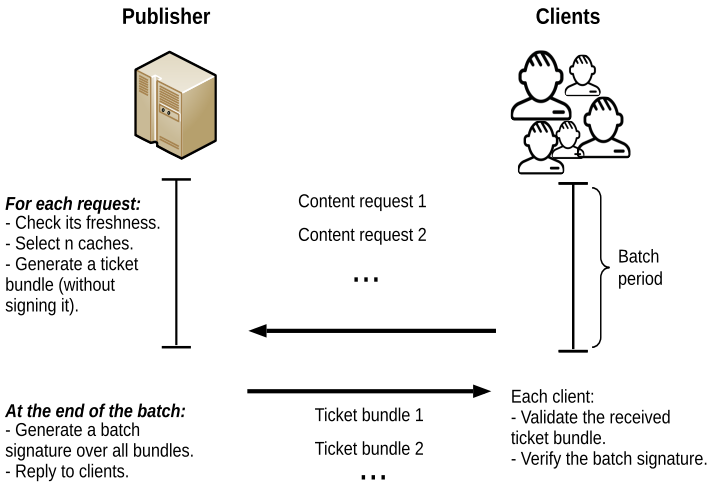


Figure B.1: Batching content requests.

hashes the request ticket and looks for an identical hash in h_{bundle}. If found, it proceeds with verifying the batch signature over h_{bundle} in the same way as performed by the client. This process is not repeated when receiving the lottery tickets. As mentioned previously, the cache only hashes a lottery ticket and looks for identical hash inside h_{bundle}.