Kitabın bu sürümü baskıdan önceki halidir. Kitabın açık erişimle paylaşılan son hali:
https://bilgiguvenligi.org.tr/BGD/Siber_Guvenlik_ve_%20Savunma_Kitap_Serisi_5_Blokzincir%20ve%20Kriptoloji.pdf

Çalışmayı referans vermek için:

Enis Karaarslan, Melih Birim, "Blokzincirde Güvenli ve Güvenilir Uygulama Geliştirme Temelleri", Siber Güvenlik ve Savunma Blokzinciri ve Kriptografi, p 1-48, Nobel Yayınevi, 2021

Blokzincirde Güvenli ve Güvenilir Uygulama Geliştirme Temelleri

Enis Karaarslan, Melih Birim

Özet:

Merkezi olmayan çözümlerin öneminin anlaşıldığı ve hayata geçirilmeye başlandığı bir çağdayız. Blokzincir teknolojisi ve akıllı sözleşme tabanlı merkezi olmayan uygulamalar birçok alanda iş yapma şeklimizi değiştirmeyi vadetmektedir. Merkezi olmayan uygulamalar birçok sistem için devrimsel değişiklikler yapma potansiyelini taşımaktadır ama bildiğimiz yazılım geliştirme süreçleri bu ortamlarda geliştirme yapmak için yeterli değildir. Blokzincirinde değiştirilemez kayıtlar kullanıldığı ve kodlar otonom çalıştığı için olası bir hatada değer kaybı yüksek olmaktadır. Bu kodların yazıldığı dil ve ortamlar henüz yeterince olgun değildir. Özellikle değer transferlerinde sorun yaşanmaması için güvenli ve güvenilir uygulamalar geliştirilmelidir. Bu tür kodların nasıl geliştirileceği ve test edileceği konusunda daha fazla örnek uygulama ve kılavuza ihtiyaç duyulmaktadır. Bu bölüm ile bu konularda temel altyapının verilmesi hedeflenmektedir. Bölüm blokzincir teknolojisinin ne olduğuna dair bir ön bilgi ile başlamaktadır. Ethereum, Quorum, Hyperledger, Corda, Avalanche ve Polygon gibi blokzincir platformlarına dair ön bilgi verilecektir. Blokzincir çözümlerinin nasıl geliştirileceği ele alınacaktır. DS4H blokzincir araştırma ağı hakkında bilgi verilecektir. Güvenli ve güvenilir akıllı sözleşme geliştirmeye dair uygulama örnekleri Ethereum Solidity ortamında verilecektir. Akıllı sözleşme geliştirilmesi, akıllı sözleşmelerin blokzincir sistemine yüklenmesi ve test edilmesi konusunda yapılmakta olan çalışmalardan ve geliştirilmekte olan araçlardan (Tubu-io, GoHammer) söz edilecektir. Akıllı sözleşmelere olan belli başlı saldırılar için çözüm önerileri verilecektir. Akıllı sözleşmeler için güvenlik denetim listesi ve öneriler sunulacaktır. Blokzincir ağ başarım, akıllı sözleşme testleri ve güvenlik kontrolleri için kullanılabilecek yöntemler tanıtılacaktır. Blokzincir ortamında yazılım geliştirme süreçlerinde yaşanan sıkıntılara ve fırsatlara değinilecektir. Merkezi olmayan sistemlerin güvenilir ve sürdürülebilir olması için yapılabilecek çalışmalara dair öngörüler paylaşılacaktır.

1.1 Giriş

İş süreçlerimizi güven (trust) içinde yapmak için güvenilir aracı kişiye veya kuruma (trusted third party) ihtiyaç duyarız. Bu süreçlere aracı olmaları ve işlemlerin kaydını güvenilir bir şekilde saklamaları için banka ve noter gibi birçok aracı (intermediary) kurumdan yararlanırız. Bu aracı kurumlar genellikle süreci yavaşlatır, işlemler ve komisyonlar çoğunlukla maliyetlidir ve işlemler sadece çalışma saatlerinde yapılabilir. Güvenilir işlemler için aracılar kullanılsa da, yine de işlemler çeşitli şekillerde manipüle edilebilir. Aracıları ortadan kaldıran otonom ve güvenilir (trusted) sistemler kurmak için bilgi teknolojilerinin kullanılması üzerine günümüzde denemeler yapılmaktadır. Merkezi olmayan P2P (peer to peer) ağlar üzerinden çalışan dosya sistemi BitTorrent bu konudaki ilk başarılı denemelerden biridir. Bu tür sistemler öncelikle dağıtıktır (distributed). Merkezi bir otoritenin/sunucunun kontrol etmediği ve aracıların olmadığı çözümlere ise merkezi olmayan (decentralized) çözümler denir. Satoshi 2009'daki yayında [1], aracılar olmadan taraflar arasında güvenilir bir şekilde değer transferinin gerçekleşmesini sağlayacak blokzincir sistemini ve bu süreçte kullanılabilecek Bitcoin kripto parasını (cryptocurrency)

--

tanıtmıştır. Blokzincir sistemi, 2009'dan beri aktif olan Bitcoin ile çalışabilirliğini kanıtlamıştır. Bu teknoloji, devrimsel değişiklikler vaat etmektedir. Blokzinciri üzerinde akıllı sözleşme (smart contract) adı verilen kodlarla merkezi olmayan çözümler geliştirilmesi mümkün olmaktadır. Akıllı sözleşmeler ile süreçler otonom hale gelmekte veya daha az insan katılımı ile gerçekleştirilebilmektedir. Bu da sözleşme işleme maliyetini önemli ölçüde azaltmaktadır. QYResearch'in raporuna [2] göre, küresel akıllı sözleşmeler pazar büyüklüğünün; 2019'da 106,7 Milyon ABD Dolarından 2026 yılına kadar 345,4 Milyon ABD Dolarına ulaşacağı tahmin edilmektedir. Tedarik zinciri, bankacılık, devlet, sigorta ve emlak sektörlerinde yaygın kullanım beklenmektedir. Blokzincir teknolojisinin artan popülaritesi de bu pazara olan talebi arttırmaktadır.

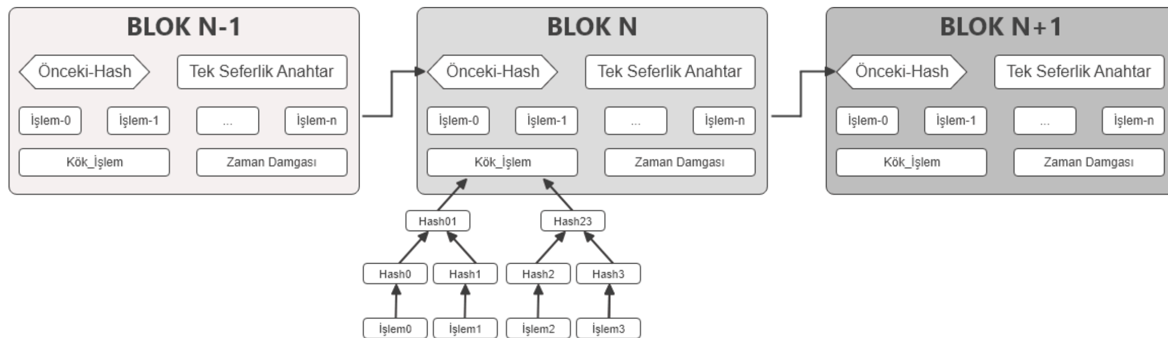
Akıllı sözleşmeler otonom çalıştıklarından ve blokzincirinde değiştirilmez kayıtlar kullanıldığından; olası bir hatada veya güvenlik zafiyetinde değer kaybı yüksek olacaktır. Akıllı sözleşmelerdeki hatalar sonucunda DAO (Decentralized organization - merkezi olmayan organizasyon) saldırıları, Ether Tahtının Kralı (King of the Ether Throne) ve çok ortaklı oyunlar (Multi-player Games) gibi çeşitli saldırılar yaşanmaktadır [3]. Bu kapsamda en büyük saldırılardan olan ve 2016 senesinde yapılan DAO saldırısında; o zaman için 50 milyon dolar değerinde olan 6 milyon ETH (Ethereum) çalınmıştır [4].

Bu saldırılar; blokzincirinin kullandığı kriptografik modelden ziyade, uygulama geliştirme süreçlerindeki sıkıntılardan kaynaklanmaktadır. Akıllı sözleşmelerin geliştirildiği dil ve ortamlar gelişmektedir ve henüz yeterince olgun değildir. Bu ortamlarda çözüm geliştirilirken nelere dikkat edilmesi gerektiği bu bölümde ele alınacaktır. Güvenli ve güvenilir yazılım geliştirmek için; blokzincir platformları, konsensüs protokolleri ve akıllı sözleşmelerin test edilmesi gereklidir. Yazılımcılar, blokzincir sistemleri üzerinde uygulama geliştirebilecekleri ve test süreçlerinde kolay kullanabilecekleri araçlara ihtiyaç duymaktadır. Bu kapsamda geliştirilmekte olan platform ve araçlar [5, 6] da bu bölümde sunulacaktır.

Bir sonraki bölümde blokzincir teknolojisinin ne olduğuna dair bir temel bilgi verilecektir. Üçüncü bölümde merkezi olmayan çözümler anlatılacaktır. Blokzincir platformlarına, merkezi olmayan çözümlere ve yönetilebilir ağ servislerine dair temel bilgiler verilecektir. Ethereum Solidity ortamında akıllı sözleşmelerde güvenli kod geliştirme dördüncü bölümde verilecektir. Beşinci bölümde akıllı sözleşmelere olan belli başlı saldırılar ele alınacaktır. Blokzinciri ağ başarım (performance) testleri ve akıllı sözleşme testleri altıncı bölümde ele alınacaktır. Yedinci bölümde güvenlik kontrol listesi verilecektir. Blokzinciri için yazılım geliştirmede sıkıntılar ve fırsatlar sekizinci bölümde tartışılacaktır. Bölüm sonuç ve değerlendirmeler ile bitecektir.

1.2 Blokzincir Teknolojisi Nedir?

Blokzinciri, merkezi olmayan bir şekilde işlemleri doğrulayan ve işlemlere dair kayıtların kaydını tutmak için kullanılan bir teknolojidir ve ilk kez Satoshi'nin Bitcoin'in nasıl çalıştığını anlattığı yayında [1] tanımlanmıştır. Blokzinciri, merkezi olmayan defter teknolojisi (DLT) olarak da adlandırılır. Blokzincirindeki blok yapısı Şekil 1.1'de verilmiştir.



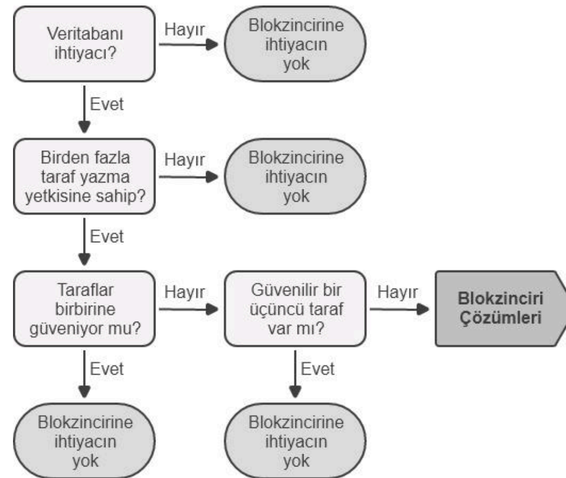
Şekil 1.1 Blok Yapısı [7]

Blokzincirinde kayıtlar, bir sistemin otonom olarak çalışmasına izin veren bir işlem (transaction) veya bir program kodu (akıllı sözleşme) hakkında bilgi içerir. İşlem bilgisine dair kaydın değişmediğini ve bütünlüğünü (integrity) göstermek için Keccak-256, SHA-3 gibi kriptografik hash fonksiyonları kullanılır [8]. İşlemlerin hash değerleri bir Merkle ağacı oluşturacak şekilde hesaplanır ve "Kök İşlem" (Tx_Root) değeri oluşturulur. Her blokta tek seferlik anahtar (nonce) değeri ve zaman damgası tutulur. Kayıtlar blok diye adlandırılan veri yapılarında toplanır. Bu bloklar bir önceki bloğun hash değerleriyle birbirine bağlanır ve bu yapıya blokzinciri denir. Bu kayıtların tümüne kayıt defteri (ledger) da denir [1]. Kayıt defteri, düğüm (node) adı verilen cihazlarda tutulur. Bu cihazlar farklı bilgisayar ağlarında çalışır ve P2P protokolleri ile birbirleriyle iletişim kurarlar. Bunlar aynı anda sunucu veya istemci görevi gerçekleştirir ve merkezi olmayan bir sistem oluştururlar. Düğümler farklı donanıma ve fonksiyonlara sahip olabilirler.

Sistemin tutarlı çalışabilmesi için bu cihazların fikir birliği/konsensüs (consensus) içinde çalışabilmesi gerekir. Bu cihazlar; sistemdeki işlemlerin geçerliliği, bir sonraki bloğu hangi cihazın yazacağı gibi ortak kararlar için konsensüs protokollerini kullanır [9]. Kullanılan konsensüs protokolüne göre blokzincir teknolojisinin çalışması için gereken minimum düğüm sayısı değişkenlik gösterir. Örneğin Raft protokolüne göre gereken minimum düğüm sayısı üç iken IBFT protokolüne göre bu sayı dörttür.

Merkezi olmayan çözümler sürekli gelişmektedir. Blokzincirinde en önemli milatlardan birisi, Ethereum platformu ile akıllı sözleşme/kontrat (smart contract) kavramının, yani bir adreste tanımlı otonom kodların kullanılabilir olmasıdır [10]. Ethereum'dan sonra Corda, Hyperledger gibi farklı blokzincir platformları da benzer otonom kod geliştirme ortamları sunmuşlardır. Ethereum günümüzde akıllı sözleşmeleri kodlamak ve işlemek için kullanılan en gelişmiş teknolojidir. En büyük akıllı sözleşme pazar payına (2020-2026 tahmin süresi boyunca) Etherum'un sahip olmaya devam etmesi beklenmektedir [2].

Blokzinciri, tüm yaşanan problemler için uygun bir çözüm değildir. Bu tür bir çözüme ne zaman ihtiyaç duyulabileceği, Wüst ve Gervais'in çalışmasında [11] anlatılmıştır ve Şekil 1.3'de özetlenmiştir.

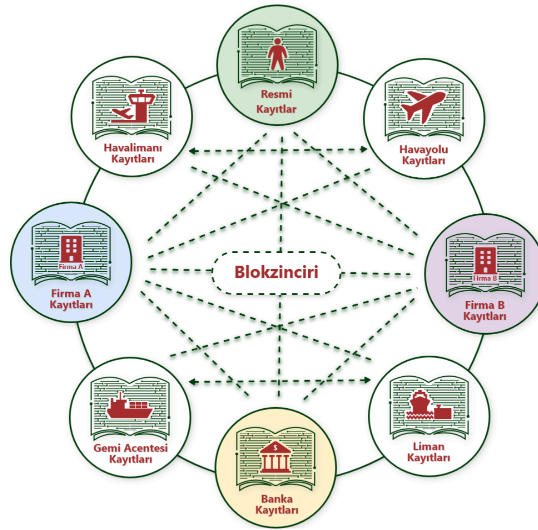


Şekil 1.3 Blokzincirine ihtiyacınız var mı? [11]

Bir alanda blokzinciri kullanımının anlamlı olabilmesi için öncelikle kayıtların saklanacağı bir veritabanı çözümüne ihtiyaç duyuluyor olmalıdır. Bu veritabanına farklı tarafların (kurum veya kişilerin) yazma ihtiyacının olması gerekir. Bu süreci gerçekleştirecek taraflar arasında güvenin düşük olması ve bu güveni sağlayacak güvenilir herhangi bir aracı kurumun (trusted third party) var olmaması gerekir. Tarafların bu şartlar altında da güvenilir işlem yapmak istemesi ve bu işlemlerin sonradan denetim için kayıt edilmesi ihtiyacının var olması durumunda blokzinciri kullanımı anlamlı olmaktadır [12]. Blokzincirin kullanımının potansiyeli olduğu başlıca alanlar olarak; finans ve değer aktarımı, tedarik zinciri (supply chain), dolandırıcılık tespiti (fraud detection), sağlık veri değişimi (health data exchange)

[7], müşterini tanı (know your customer), merkezi olmayan kimlik (decentralized identity), akıllı yönetim (smart governance) [13], ulusal ve siber güvenlikten söz edilebilir.

Blokzincir süreçlerinin uygulanabileceği alanlardan birisi tedarik zinciri uygulamalarıdır. Kurum tedarik zincirindeki tüm süreçleri izlemek ve kullanıcılarına bu süreçleri şeffaf hale getirmek isteyebilir. Örneğin Muğla'daki ES firmasından Osaka'daki Burokkuchēn firmasına bir lavanta ihracatında, Şekil 1.4'de gösterildiği gibi taşıma şirketleri, limanlar, resmi kurumlar ve bankalar gibi çok farklı kurumlar bu sürece dahil olacaktır. Tüm bu kurumların aralarında veri paylaşması ve işlem oluşturması gerekecektir. Blokzinciri, birbirine güvenmek zorunda olan birçok tarafın olduğu böyle bir veri paylaşımında anlamlı bir çözümdür[1]. TradeLens [14] blokzincir çözümü bu alana örnek verilebilir. 300'den fazla organizasyon, 10 transatlantik nakliye şirketi, 600'den fazla liman ve terminal bu sistemi kullanmaktadır[2]. Blokzinciri tabanlı sistemler akıllı şehir (smart city) uygulamalarında kullanılabilir. Merkezi olmayan kimlik yapıları aracılığı ile vatandaşların bilgilerine ulaşılabilir. Vatandaşlar kendi kişisel verilerinin mahremiyetini sıfır bilgi kanıt (zero knowledge proof) [15] protokolleri ile koruyabilir. Böyle bir sistem; pasaporttan sağlık ve tapu uygulamalarına kadar birçok alanda kullanılabilir [13].



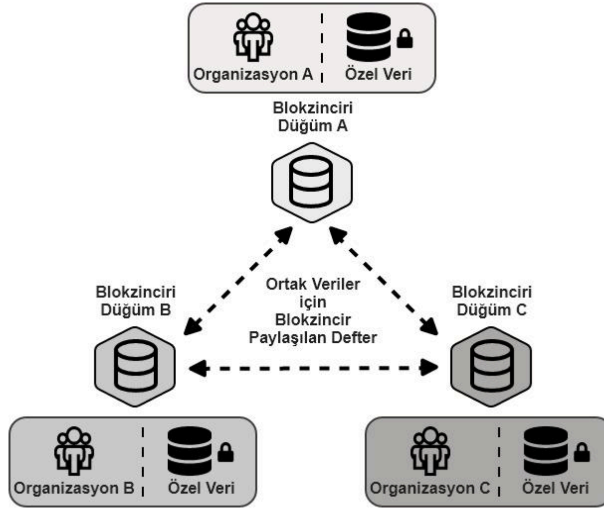
Şekil 1.4. Çok farklı kurum veri erişimi senaryosu [7]

Blokzinciri tabanlı sistemlerde verinin kaydedilmesi ve sonradan erişilmesinde başarım ölçütleri dikkate alınmalıdır. Sistemin sürdürülebilir olması için öncelikle verinin ölçeklendirilmesi ve veriye hızlı bir şekilde ulaşılabilmesi gerekir. Bu durumda karma (hybrid) çözümlerin kullanımı da söz konusudur. Böyle bir kullanım modeli önceki bir çalışmamızda [7] verilmişti ve Şekil 1.5'te gösterilmiştir.

---

[1] Mohan C. (2019). State of Permissionless and Permissioned Blockchains: Myths and Reality, BlueTalks @ Rio BNDES

[2] Hapag-lloyd And Ocean Network Express Complete Tradelens Integration, https://www.tradelens.com/post/hapag-lloyd-and-ocean-network-express-complete-tradelens-integration



Şekil 1.5 Paylaşılan veri senaryosu [7]

Blokzincir sistemi bu kullanım modelinde bir alternatif çözüm olmaktan daha çok, bütünleyici olarak yer alacaktır. Bu yapıda, kurumlar kendi özel verilerini kendi sistemlerinde tutmaya devam edeceklerdir. Ortak paylaşılan veriler ise blokzincirinde tutulacaktır [7]. Blokzincirinin veri bilimi açısından farklı uygulamaları Karaarslan ve Konacaklı'nın bir başka çalışmasında [12] ayrıntılı olarak ele alınmıştır.

1.3 Merkezi olmayan çözümler

Merkezi olmayan üç farklı çözüm türünden söz edilebilir. Bunlar; blokzincir platformları, merkezi olmayan uygulamalar ve yönetilebilir ağ servisleridir. Alt başlıklarda öncelikle blokzincir platformlarından, sonrasında da merkezi olmayan uygulamalardan söz edilecektir. Burada akıllı sözleşmelerin nasıl çalıştığı anlatılacak ve Ethereum ortamında kullanılan belli başlı uygulama geliştirme araçları kısaca tanıtılacaktır. En son olarak da uygulama geliştirme ve test sürecini kolaylaştırmak için yönetilebilir ağ servisleri anlatılacaktır.

1.3.1 Blokzincir Platformları

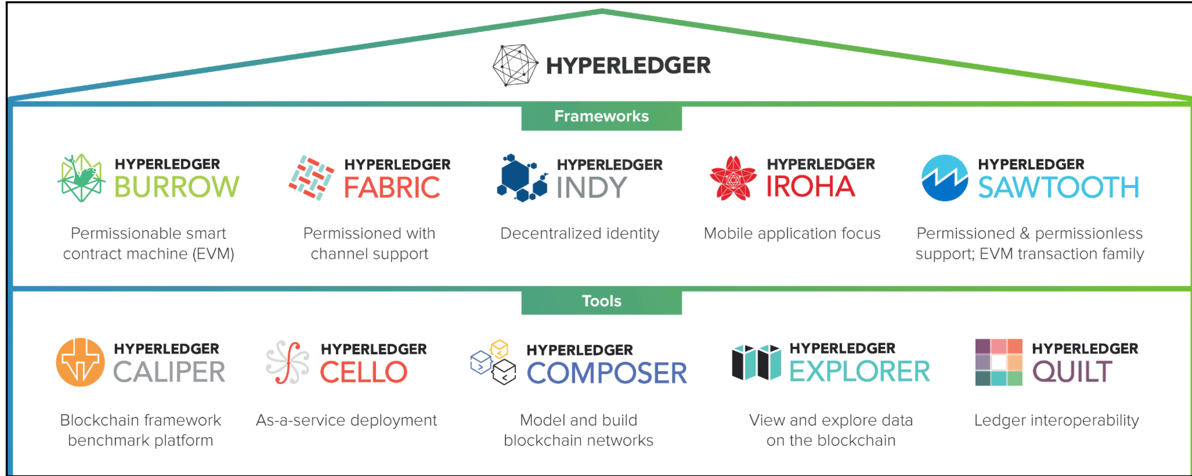
Blokzincir platformları (blockchain framework); blokzincir sistemine dair temel altyapıyı sunarlar. Aynı zamanda blokzincir tabanlı çözümlerin geliştirilebileceği ortamlar da sağlarlar. Bu platformlar blokzincir düğümleri üzerinde bir servis olarak çalıştırılır. Ethereum, Quorum, Hyperledger ve Corda günümüzdeki belli başlı blokzincir platformlarıdır.

Merkezi olmayan uygulamalar; Ethereum gibi genel (public) ağlar üzerinde çalıştırıldığında işlem ücreti ödenmesini gerektirecektir. Kurumsal uygulamalar için ayrıca platform kurulabilir. Bu platform, bir bulut servisi olarak alınabilir veya kurum altyapısına kurulur. Platformların lisansı izin veriyorsa; platform klonlanıp yeni bir sistem oluşturulur. Kurum isterse kendi platformunu sıfırdan kendisi geliştirir.

Blokzincir platformları farklı konularda uzmanlaşmaktadır. Örneğin, Corda ve Quorum finansal çözümler için geliştirilmiştir. Linux Foundation altında farklı gruplar tarafından geliştirilen Hyperledger, farklı platformları ve farklı araçları bünyesinde bulunduran bir çatıdır. Başlıca ürünler Şekil 1.7'de gösterilmiştir. Örneğin, Hyperledger Indy merkezi olmayan dijital kimlik uygulamalarında, Hyperledger Caliper başarım (performance) ölçümünde kullanılmaktadır. Bunun yanı sıra farklı platformlardan söz etmek mümkündür. Örneğin, Prof. Dr. Emin Gün Sirer'in kurduğu Avalanche platformu[3] da gelişmekte olan [16] ve geliştiricilerine çeşitli fırsatlar sunan sayılı ortamlardan biridir. Polygon[4] ise Ethereum

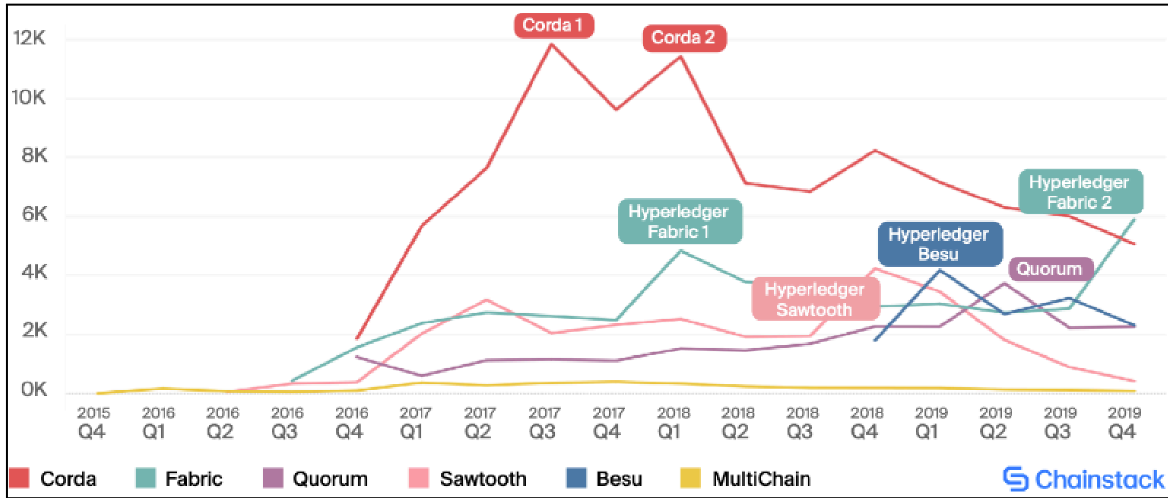
[3] Avalanche platformu, https://www.avalabs.org/
[4] Polygon protokol ve platformu, https://polygon.technology/

uyumlu blokzincir ağlarını oluşturmak ve birbirine bağlamak için bir protokol ve platformdur. Bu ortamda çoklu zincirli (multi-chain) bir ekosistem için ölçeklenebilir (scalable) çözümler geliştirilebilir.



Şekil 1. 7 Belli başlı Hyperledger çözümleri[5]

Blokzincir platformu seçerken geliştirici topluluklarının etkinliğine, topluluğun geliştirici sorunlarına cevap verme hızlarına ve projenin sürdürülebilirliğine dikkat edilmesi gereklidir. Platformlar üzerinde yapılan teknolojik geliştirmeler ve geliştirme topluluklarının etkinliği açısından Ethereum ve Hyperledger platformları sürdürülebilir projeler olarak gözükmektedir. Blokzincir platformlarının popülerliği zamana göre değişebilmektedir. Ethereum bu alanda en fazla kod yazılan ortam olarak liderliğini korumakla birlikte; Hyperledger, Quorum ve Corda'nın 2017-2019 seneleri arasında Github faaliyetlerinin değişimi Şekil 1.8'de gösterilmiştir.



Şekil 1.8 2017-2019 arasında Hyperledger - Quorum - Corda Platform Github faaliyeti [17]

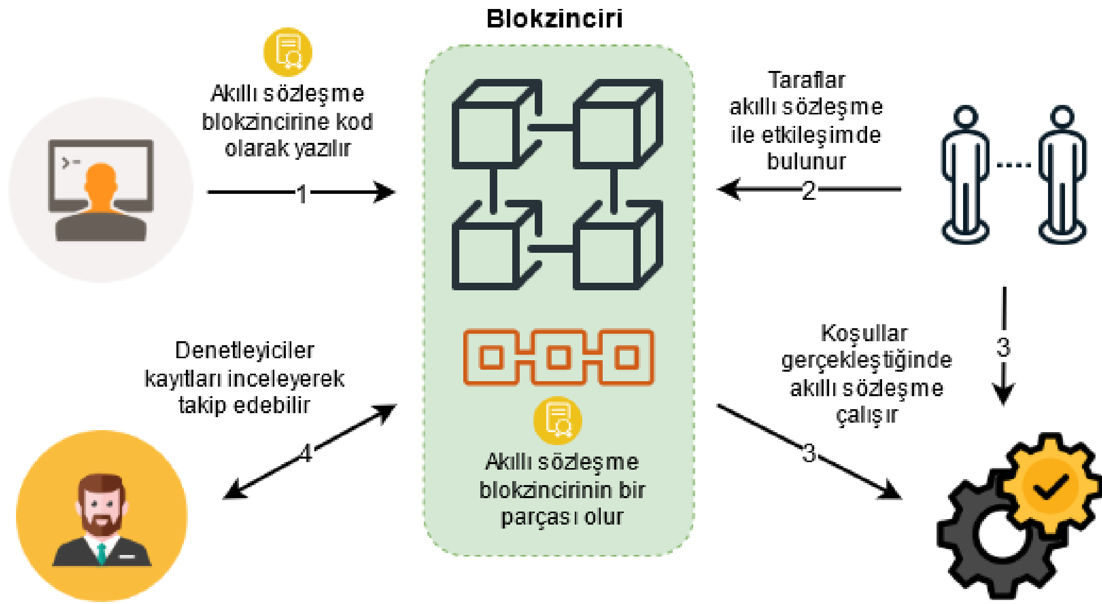
Bu şekildeki zaman diliminde; Corda'nın bir dönem ne kadar popüler olduğu ve sonra popülerliğini kaybettiği gözükmektedir. Bu zaman aralığında Hyperdleger Fabric'in popülerliği artmıştır. Zmudzinski'nin raporuna [17] göre; bu değişim Hyperledger Fabric'in 2019 kasımında kod yönetim aracı olarak Github'a geçmesinden sonra yaşanmıştır.

1.3.2 Merkezi Olmayan Uygulamalar ve Akıllı Sözleşmeler

[5] Hyperledger Solutions, https://events19.linuxfoundation.org/events/hyperledger-global-forum-2018/

Merkezi olmayan uygulamalar (decentralized applications - Dapp); blokzinciri üzerinden merkezi olmayan servisler verilmesini sağlayan uygulamalardır. Son kullanıcının bir mobil uygulama veya web tarayıcısı üzerinden blokzinciri ile iletişimini sağlayan ortamlardır. Bu uygulamalar, çoğunlukla bir API (Application Programming Interface - Uygulama Programlama Arayüzü) üzerinden blokzinciri ile iletişim kurarlar ve blokzincirindeki akıllı sözleşme olarak bilinen otonom kodu çalıştırırlar. Merkezi olmayan uygulamalar, üzerinde çalıştıkları blokzincir platformuna göre farklılıklar içermektedir. Ethereum ve türevlerinde otonom kodlara akıllı sözleşme/kontrat (smart contact) denir. Hyperledger dünyasında bu tür kodlara zincir kodu (chain code) denmektedir. Literatürde akıllı sözleşme kavramı daha sıklıkla kullanılmaktadır ve bu bölümde de bu şekilde kullanılacaktır.

Akıllı sözleşme yeni bir kavram değildir. Bu kavramı ilk olarak Nick Szabo 1997'de yayımladığı çalışmasında [18] ortaya atmıştır. Akıllı sözleşmeler, genel ağ (public network) üzerinden iletişim kuran taraflar arasındaki ilişkileri resmiyete döken (formalize) ve güvence altına alan (secure) programlardır. Akıllı sözleşmelerin uygulanabilir olması için determinizm gereklidir. Bu da blokzincir ağındaki tüm düğümlerin bu kodları çalıştırdığında ortaya çıkan son durumun (resulting state) ve işlemlerin (transaction) aynı olmasını gerektirir. Ethereum [10] ve benzeri merkezi olmayan platformlar rassallık (randomness) gibi deterministik olmayan durumlara (non-determinism) izin vermeyerek bunu sağlarlar [19]. Bir akıllı sözleşmenin (kontrat) çalışma aşamaları Şekil 1.2'de gösterilmiştir.



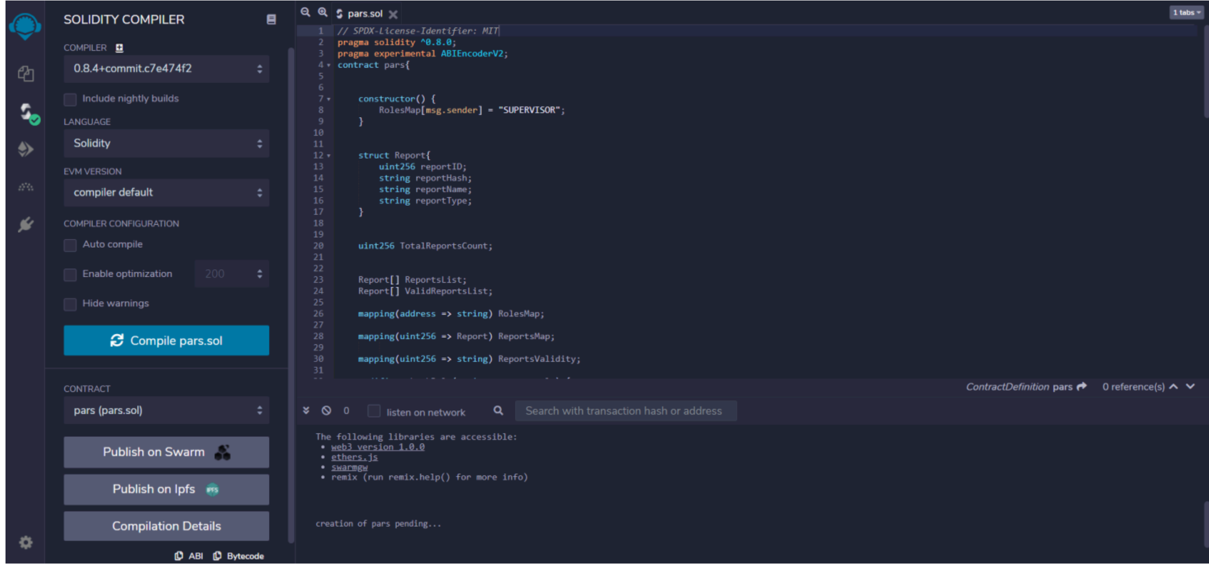
Şekil 1.2 Akıllı Sözleşmenin Çalışma Şekli

Geliştirici akıllı sözleşmeyi kodladıktan sonra blokzincirine kayıt olarak ekler. Bu kod Ethereum örneğinde genel (public) bir zincirde tutulabildiği gibi, farklı platformlarda özel (private) zincirlerde sadece belirli kurumlar arasında görünür bir şekilde de tutulabilir. Taraflar blokzincirindeki hesapları (account) üzerinden akıllı sözleşme ile etkileşimde bulunur. Akıllı sözleşmede belirlenmiş koşulların gerçekleşmesi durumunda kod kendini çalışır. İşlemin sonuçlarına dair kayıtlar blokzincirine yazılır. Böyle bir sistemde denetim süreçleri işlem kayıtlarının sonradan incelenmesi ile mümkün olur. Denetleyiciler (regulators), uygulamanın çalıştığı alana özel denetleme mekanizmalarıdır. Denetleyiciler blokzincir kayıtlarını inceleyerek kontratların çalışma durumunu takip edebilirler.

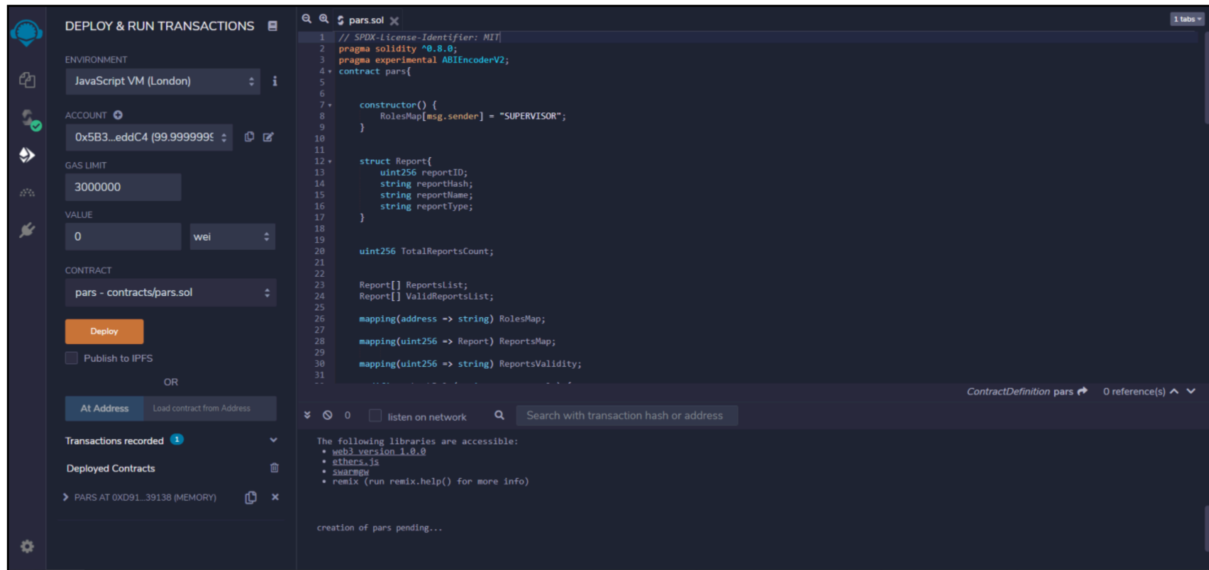
Ethereum'da bu kodlar (kontratlar) Solidity dilinde yazılır. Ethereum, şu ana kadar en fazla otonom kod geliştirilen ortamdır. Ethereum ortamında geliştirme sürecinde kullanılan belli başlı araçlar olarak Geth, Clef, Remix, Metamask, Truffle, Ganache ve web3.js örnek verilebilir. Geth, Ethereum'un Golang dilinde geliştirilmiş bir istemcisidir. Parity ve Besu gibi farklı özelliklere sahip Ethereum istemcileri de

bulunmaktadır. Clef, Ethereum da hesap yönetimi için kullanılan yeni bir sistemdir. Tek başına bir işlem olarak çalışır ve tüm hesap yönetimlerini Ethereum Geth istemcisi yerine kendisi sağlar.

Remix IDE (integrated development environment - tümleşik geliştirme ortamı), Ethereum akıllı sözleşmeleri geliştirip blokzincirine yüklemeye (deploy) imkan tanıyan, açık kaynaklı bir web ve masaüstü uygulamasıdır. Remix geliştirme ortamı Şekil 1.9'da gösterilmiştir. Bu arayüz üzerinden derleme (compile) ve blokzincirine yükleme süreçleri gerçekleştirilebilmektedir. Zengin bir eklenti setine sahiptir. Remix, tüm sözleşme geliştirme sürecinde kullanılabileceği gibi, Ethereum'u öğrenmek ve öğretmek için de kullanılabilir.



Şekil 1.9a Remix Geliştirme Ortamı - Derleme



Şekil 1.9b Remix Geliştirme Ortamı - Blokzincirine Yükleme

Truffle Suite; Truffle, Ganache, Drizzle gibi bir dizi programlama aracından oluşan bir geliştirme paketidir. Ortam arayüzü Şekil 1.10'da gösterilmiştir. Truffle popüler bir geliştirme ve test ortamı, aynı zamanda da bir varlık hattıdır (asset pipeline). Bu programlar geliştirme sürecini kolaylaştırır. Akıllı sözleşme yaşam döngüsü yönetimi, otomatik akıllı sözleşme testi, betiklerle (script) dağıtım ve taşımalar (mitigation), basit ağ yönetimi ve interaktif konsol gibi akıllı sözleşme geliştirme sürecini kolaylaştıran özelliklere sahiptir.

```
truffle(development)> accounts
[
  '0x23Db94fC1d9918966021B6a085aAE2ff3E6E9d66',
  '0x2Ba0463674C0D7DcE9B9A5B7e6794d6E3aA2e969',
  '0xB6F81934A6f6529D4f9dBC8E46b9aEB3c9d9916b',
  '0xF7c971DC9Af78ebD567df7Bb386201BfD82d2b0e',
  '0x43dE2AbF554Aa7Fd8833cA4E97EE611363a6bc50',
  '0x43AA5621F2eEb6aB66E3b5438A314BCf877AB9Bf',
  '0xdE5afA02efd9E8F7C7aAB122CA953B71153C1499',
  '0x0D8d2eeC60230D248e2f022bFC3d325C5D70be28',
  '0x40c731A5b1fe8A7376ad1C2181e1DEb482aE3f6A',
  '0xCdAb910144785D3b5367d16D78bC4CC328cE9E2f'
]
truffle(development)> contract.setUserRole(accounts[3], "LABORANT")
{
  tx: '0xc5cfe8c355ccdcaddd3cf9e2c859fa79c56dc7c74015a58f23d1814f90fe7d65',
  receipt: {
    transactionHash: '0xc5cfe8c355ccdcaddd3cf9e2c859fa79c56dc7c74015a58f23d1814f90fe7d65',
    transactionIndex: 0,
    blockHash: '0x5d3223b06a712356b7f9ffbdf0143d7e767fcc247a3b687741f9c2a0f5a00eb1',
    blockNumber: 11,
    from: '0x23db94fc1d9918966021b6a085aae2ff3e6e9d66',
    to: '0xd3c03426280e81908618de965f3566027394a142',
    gasUsed: 46456,
    cumulativeGasUsed: 46456,
    contractAddress: null,
    logs: [],
    status: true,
    logsBloom: '0x00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    rawLogs: []
  },
  logs: []
}
truffle(development)> contract.getUserRole(accounts[3])
'LABORANT'
truffle(development)> █
```

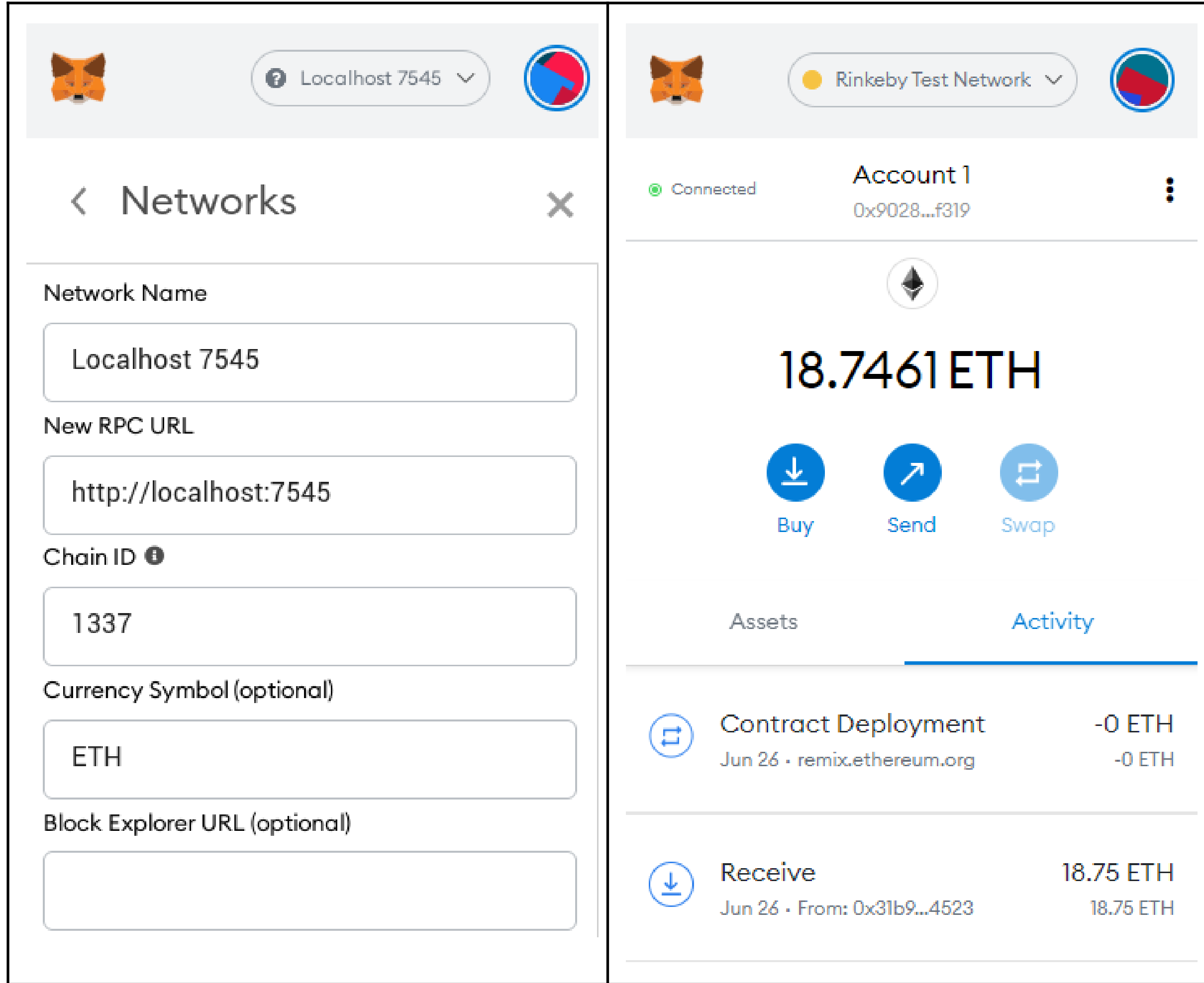
Şekil 1.10 Truffle Arayüzü

Truffle ile birlikte gelen Ganache ile yerel makineye bir Ethereum blokzinciri ağ simülatörü kurulur ve Solidity sözleşmeleri bu ortam üzerinde test edilir. Blokzinciri sistemi üzerindeki işlemlerin (transaction) ayrıntılarının takip edilebildiği Genanche arayüzü Şekil 1.11'de gösterilmiştir.

Şekil 1.11a Ganache Arayüzünde işlemlerin takibi

Şekil 1.11b Ganache ile bir işlemin detaylı gösterimi

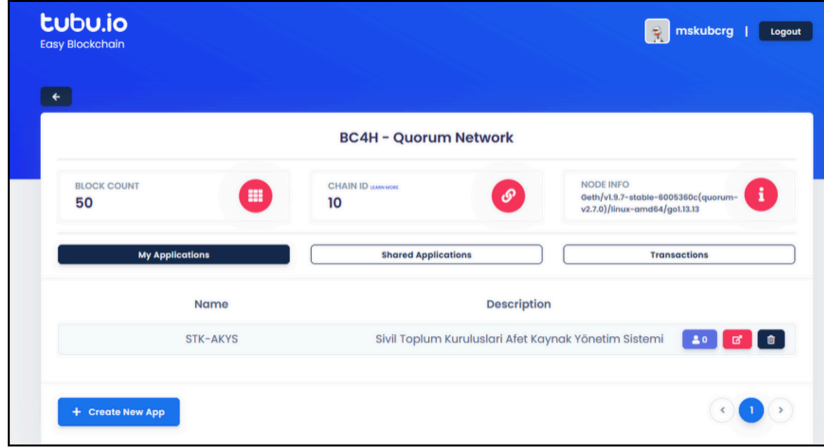
Metamask, blokzincir uygulamaları için bir kripto cüzdanıdır. Blokzinciri tabanlı uygulamalara güvenli bir bağlantı sağlamak için bir ağ geçidi olarak da çalışır. Uygulamalar geliştirilirken Truffle ile birlikte kullanılır. Arayüzü Şekil 1.12'de gösterilmiştir. Web arayüzü geliştirmek için kullanılabilecek web3.js, bir HTTP veya IPC bağlantısı kullanarak yerel veya uzak bir Ethereum ağı ile etkileşim kurulmasına izin veren bir kitaplık koleksiyonudur.



Şekil 1.12 (a) Metamask Network Ayarları (b) Metamask ile Rinkeby Test Ağına bağlı bir hesabın takibi

1.3.3 Yönetilebilir Ağ Servisleri

Geliştiricilerin canlı ortam testlerini gerçekleştirilmesi kolay süreçler değildir. Blokzincir teknolojileri her geçen gün gelişmektedir ve tüm gelişmelere hakim olmak ciddi zaman ve efor gerektirmektedir. Yönetilebilir ağ servisleri (managed network services); test ve ölçeklendirme süreçlerini kolaylaştırıp hızlandıran ağ servisleridir. Geliştiricilerin blokzincirinde kod çalıştırdıklarında sonuçları takip etmesi de kolay değildir. Bu tür servisler; geliştiricilerin özellikle state (durum) takibi yapmalarını kolaylaştıracaktır. Kod çalıştığında sistemde oluşturacağı değişikliğin takibi sağlanacaktır. Bu servisler; geliştirilen kodun değişik sürümlerinin (version) takibini de sağlayacaktır. Farklı konsensüs protokolleri seçiminde sistem performansın değişiminin gözlenmesi için de kullanılabilir. Örnek olarak; Simba (https://simbachain.com), Chainstack (https://chainstack.com) ve Tubu-io (https://www.tubu.io) verilebilir. Yönetilebilir ağ servisleri; geliştiricilerin bu süreçlerde sonuçları takip edebilecekleri, kullanımı kolay arayüzler de sağlamalıdır. Örneğin, Tubu-io blokzincir uygulama ortamı (workbench) (https://www.tubu.io), geliştiricilerin akıllı sözleşmeleri blokzincir ağlarına yüklemeleri ve bu sözleşmelerle kolayca etkileşim kurmaları için geliştirilmiştir. Tubu-io web arayüzü Şekil 1.13'de gösterilmiştir.



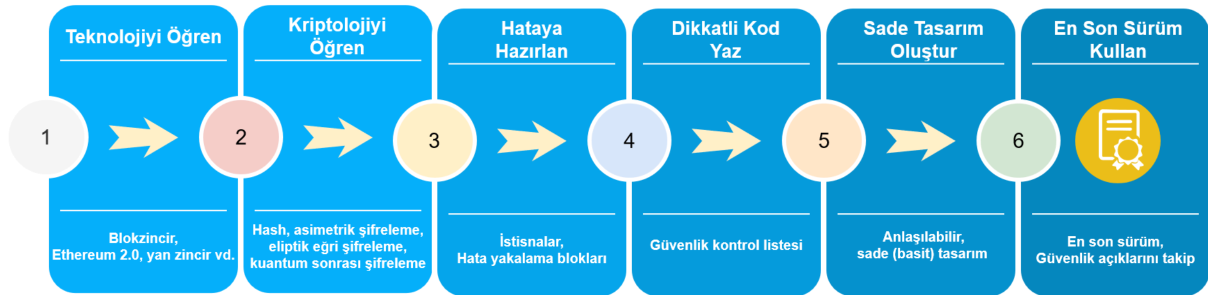
Şekil 1.13 Tubu-io Web Arayüzü [5]

Geliştiriciler Tubu-io web arayüzü üzerinden blokzincir ağları ile etkileşimlerini gerçekleştirebilmektedir. Bu uygulama geliştirme ortamı, merkezi olmayan uygulama programlamaya yeni başlayacak geliştiriciler için eğitim amaçlı da kullanılabilir. Muğla Sıtkı Koçman Üniversitesi'nde verilmekte olan CENG 3550 "Decentralized Systems and Applications" dersinde aktif olarak kullanılmaktadır. Bu ortamın kullanımının; merkezi olmayan uygulama projeleri geliştirme süresini ve maliyetlerini azaltmada etkisi olacaktır. Detaylı bilgi için bkz [5].

1.4 Akıllı Sözleşmelerde Güvenli Kod Geliştirme

Akıllı sözleşme programlama, alışık olduğunuzdan daha farklı bir mühendislik zihniyeti gerektirir. Uygulamada başarısızlığın maliyeti yüksektir ve sonrasında değişim zordur. Bu da Ethereum akıllı sözleşme geliştirmesini bu yönlerden web veya mobil geliştirmeye kıyasla donanım programlamaya yakınlaştırmaktadır. Bu nedenle, bilinen güvenlik açıklarına karşı savunma yapmak yeterli değildir. Bunun yerine, yeni bir geliştirme felsefesi öğrenmemiz ve bu açıkları önceden incelememize olanak sağlayacak yeni araçlar geliştirmemiz gerekecektir.

Akıllı sözleşmelerin üzerinde çalışacağı blokzinciri ve özellikle Ethereum çok yeni bir teknoloji ve yeni plaformlardır. Bu teknoloji her geçen gün gelişmeye ve iyileşmeye devam etmektedir. Alışılagelmiş programlama dillerinin kaynak kodları yıllarca incelenmiş ve geliştirilmiştir. Solidity dilinin bu şekilde bir süreçten geçmemiş olmaması; merkezi olmayan sistemler geliştirirken hatalara ve diğer olası güvenlik açıklarına hazır olmamızı gerektirmektedir. Bu nedenle, bu bölümde sunulan güvenlik uygulamalarını takip etmek, bir akıllı sözleşme geliştiricisi olarak yapmanız gereken güvenlik çalışmasının yalnızca başlangıcıdır. Ethereum tabanlı akıllı sözleşme geliştirirken unutmamanız gereken altı temel kural Şekil 1.14'te verilmiştir.



Şekil 1.14 Akıllı sözleşme geliştirmede altı temel kural

Bu şekilden de görüleceği üzere; blokzincir temellerini ve ayrıntılarını öğrenmeden, üzerinde geliştirme yapacağınız sistemi anlamanız ve ona göre kod yazmanız pek olası değildir. Blokzincir teknolojisinin güncel gelişmelerine de hakim olmak gerekir. Örneğin, Ethereum 2.0 ve yan zincir (side chain) gibi yapıları öğrenmek gereklidir. Bu sistemlerde kriptoloji çok önemlidir. Kriptoloji temelleri öğrenilmeli, anlaşılmalı ve sonra da unutulmamalıdır. Geliştirici bu kriptoloji esaslarının ne için kullanıldığının farkında olmalıdır. Sonraki aşamalar yazılım geliştirme süreçlerinde gerçekleşir. Hatalar her zaman olur, önemli olan hataya önceden hazırlanmaktır. Kod hataya zarif bir şekilde cevap vermelidir. İstisnalar (exception), hata yakalama blokları kullanılmalıdır. Dördüncü adım olarak; dikkatli kod yazılmalıdır. Dikkat, kod yazarken en önemli unsurdur. Dikkatli kod yazma stili geliştirmeye önem verilmelidir. Bu aşamada yapılması gerekenler güvenlik kontrol listesi bölümünde özetlenmiştir. Beşinci adım olarak; sade/basit (simple) tasarım oluşturmalıdır. Akıllı sözleşmeler basit tutulmalıdır. Akıllı sözleşmelerin, gelişmiş yazılım dilleri ve sistemleri gibi gelişmiş yapıları olmadığı için, yazılan sistemi basit tutmak oldukça önemlidir. Altıncı adım olarak; her zaman en son çıkan sürümler (version) kullanılmalıdır. Temel programlama mantığına aykırı olsa da, gelişmekte olan bu sistemlerin en son sürümlerini ve gelişmelerini takip etmek, güvenlik açıklarını daha önceden öğrenmek ve ona göre önlem almak açısından önemlidir.

1.4.1. Harici çağrılarda Dikkat Edilmesi Gerekenler

Ethereum blokzinciri kayıtlarında hali hazırda var olan, önceden blokzincirine yüklenmiş kontratlar; harici çağrı (external call) ile akıllı kontrat içerisinden kullanılabilir. Bu genelde adresi bilinen başka bir akıllı sözleşme (kontrat) üzerindeki bir fonksiyonu çağırma şeklinde olmaktadır. Harici çağrıların her zaman o sözleşmede veya bağlı olduğu herhangi bir başka sözleşmede kötü amaçlı bir kod olarak yürütülebileceğini unutmamanız gerekir. Ethereum ana ağında (Mainnet) uygulanacaksa, bu şekilde kullanılması çok yerinde olacaktır.

İlk olarak güvensiz kullanıma dair bir örnekle başlayalım. B kontratından A kontratındaki fonksiyonun çağrıldığı örnekte; IA bir interface olarak tanımlanmış ve B kontratı ile adresi verildikten sonra setNumber fonksiyonu içerisinden A kontratının fonksiyonu çağrılmıştır.

```
interface IA {
	function setNumber(uint n) external;
	function getNumber() external view returns (uint);
}
contract B {
	IA public a;
	function setContract(IA _a) public { a = _a; }
	function setNumber() public { a.setNumber(10); }
	function getNumber() public view returns (uint) {return
a.getNumber();}
}
```

Burada ilk aşamada unutulmamalıdır ki A kontratı blokzincirine yüklendiğinde, uygulama ayrıntısı bilinmediği için B kontratı içerisindeki fonksiyonun ne yapacağı bilinmemektedir. Harici çağrılar ile iletişim halinde olduğunda; diğer kontratı yazılan akıllı sözleşme içerisinde güvensiz olarak işaretlemek önemlidir. Harici çağrılar ile yapılacak etkileşim mümkün olduğunca kodun sonlarına alınmalıdır. Bu etkileşimin kod içerisindeki en son işlem olması, kontratı olası hatalara karşı daha kapalı hale getirecektir. Aşağıdaki kontratta daha güvenli bir kodlama örneği verilmiştir. “untrusted_a” tanımı her ne kadar doğru olsa da, kontrol mekanizmasının doğru tanımlanması gerekir. Bu da “setContract” fonksiyonunda durum (state) değişikliği yapmadan önce “require” veya “assert” kontrollerinin yapılması demektir.

```
contract BB{
    IA public a;
    IA public untrusted_a;

    function setContract(IA _a) public {
        IA trusted_a = IA(0xd9145CCE52D386f254917e481eB44e9943F39138);
        require(_a == trusted_a, "Address mismatch");
        a = _a;
    }
    function setUntrustedContract(IA _a) public { untrusted_a = _a; }
    function setNumber() public { a.setNumber(10); }
    function getNumber() public view returns (uint) {return a.getNumber();}
}
```

Harici çağrılarda meydana gelebilecek olası hataların; kendi yazdığınız kontratlarda kontrol edilmesi gerekir. Harici çağrılar, eğer bir adres üzerinde gidiyorsa, Solidity'nin alt düzey (low-level) çağrı özelliği ile "call", "callcode", "deletecall", "send" fonksiyonları üzerinden kullanılmalıdır. Bu alt düzey fonksiyonlar eğer bir hata var ise "false" döndürürler ve istisna fırlatmazlar. Bu yüzden "try-catch" blokları ile yakalamak yerine "false" kontrolü yapmak yeterli olacaktır. Aşağıdaki şekilde yapmayın;

```
bad_address.send(33);
bad_address.call.value(33)(""); // doğrudan diğer kontrat çağrıldı
```

Aşağıdaki şekilde yapın;

```
(bool success, ) = good_address.call.value(33)("");
if(!success) {
    // burada hata ile işlem yapabilirsiniz
}
```

Başka bir akıllı kontrat içerisinden, kontrat adresi ile çağrım yapılabilmektedir. "Address call" yapmak yerine, kontrat "import" ile kullanmak daha doğru olacaktır. Burada adresi bir interface ile kapatarak sadece belirli bir fonksiyonun çağrımı yapılabilecek hale getirilmiştir.

```
IA(trusted_address).setNumber(100)();
```

1.4.2 Kod Optimizasyonu

Ethereum'da akıllı sözleşmeler sistemde işlem (transaction) oluşturduklarında "gas" adı verilen ücreti ödemek durumundadır. Geliştiriciler; kodun sistemde harcayacağı masrafı (gas tüketim seviyelerini) kabul edilebilir bir seviyede tutmak için kodlarını iyileştirilmelidir. Çalışma (gas) maliyetlerini en aza indirirken, aynı zamanda kodun çalışma zamanını azaltmak hedeflenmelidir [20, 21].

Bu bölümde Solidity dilinde kod optimizasyonu için yapılabileceklere örnekler verilecektir. Öncelikle fonksiyon ve durum değişkenlerinin görünürlüklerinin (function visibility) düzgün ayarlanması gereklidir. Solidity dilinde fonksiyon ve durum değişkenlerinin (function visibility) "external", "public", "internal" ve "private" olmak üzere dört farklı görünürlüğü vardır. Fonksiyonlara ve değişkenlere herhangi bir görünürlük etiketi tanımlanmamışsa, varsayılan olarak "public" olarak kabul edilir. Değişkenler için otomatik olarak "set" ve "get" fonksiyonları oluşturulur. "external" fonksiyonların tanımlanması "public" olarak kullanılmasından daha doğrudur. Bu tip fonksiyonları "this" ile çağırmak mümkündür. "Public" olanlar ise doğrudan fonksiyon adı ile çağrılabilmektedir. "Internal" fonksiyonlar ve durum değişkenleri sadece içeride (mevcut ya da türetilmiş sözleşmeden) çağrılabilen fonksiyonlardır, bunlara dışarıdan erişim mümkün değildir. "Private" fonksiyonlar ve durum değişkenleri ise sadece bulunduğu kontrata

özel olmaktadır. Eğer bu kontrattan türetilen başka kontrat varsa bile “private” fonksiyonlara erişim mümkün değildir. Fonksiyon tanımlarının aşağıdaki şekilde yapılmaması gerekir.

```
uint x; // public otomatik olarak get/set tanımlanır

function auto_public() { // public

}
```

Kod optimizasyonu ve okunabilirlik açısından fonksiyonların ve durum değişkenlerinin kullanıma göre isim ve görünürlük ayarlanması yapılmalıdır. Saldırılara karşı fonksiyonların varsayılan görünürlüğü[6] ve durum değişkeni varsayılan görünürlüğü[7] ayarlanmalı ve fonksiyon isimleri ile bunlar belirtilmelidir.

```
uint private x;

function external_function() external {

    // this.external_function() ile çağrılır

}

function internal_function() internal {

    // doğrudan internal_funciton() olarak this kullanmadan çağrılır.

}
```

“abstract” ve “interface” kontrat tipleri bir seviyede yeniden kullanım sağlasa da, “interface” kontratların durum değişkenlerine erişemiyor olması, kod yazımı ve yönetimi bakımından önemlidir. “abstract” kontratlar, hem fonksiyon tanımı ve hem de o uygulama kodlarını içerdiğinden dolayı dikkat ve özel ilgi gerektirir. Yazılım mühendisliği uygulamaları gereği yeniden kullanım ve davranışsal kod geliştirmeye yönelik çalışmalar iyidir. Yine de bu tip kontratları kullanırken kod gözden geçirme stilleri tatbik edilmeli ve bilinmeyen/güvenilmeyen kaynaklardan kontratlar kullanılmamalıdır. Kodlanmış olan “abstract” kontratlar eğer adres tabanlı kullanım içeriyorsa “interface” üzerinden ulaşım sağlanacak şekilde değiştirilmelidir.

“fallback” (şalter) fonksiyonlar; her kontrat içerisinde sadece bir kere tanımlanabilen ve kontrat içerisinde eğer çağrılan isimde bir kontrat yoksa yerine çağrılan fonksiyonlardır. Bu fonksiyon, kontrata gönderilen tüm mesajlarda çalışır. Ayrıca kontrata doğrudan Ether gönderildiğinde de çalışırlar. Bu fonksiyona Ether göndermek hata atar çünkü ödenebilir (payable) bir fonksiyon değildir.

```
contract Test {

    uint x;

    fallback() external { x = 1; }

}
```

[6] SWC-100, Function Default Visibility, https://swcregistry.io/docs/SWC-100
[7] SWC-108, State Variable Default Visibility, https://swcregistry.io/docs/SWC-108

Fonksiyonlar “payable” olarak yazıldığında, kontrat artık Ether alabilir.

```
contract Test2 {

    uint x;

    fallback() external payable { x = 1; }

}
```

Fallback fonksiyonlarında yapılabilecek en doğru şey olay (event) tetiklemektir.

```
contract Test2 {

    uint x;

    event  LogFallbackFunction(address sender);

    fallback() external payable {

        emit LogFallbackFunction(msg.sender);

    }

}
```

Kontrat günlüğe alma (logging) için olay tetiklemesi kullanılmalıdır. Olaylar kontrat içerisinde kullanılan ve işlemler (transaction) sırasında olan değişimleri ve anlık durumları haber verebilecek günlük yapılarıdır. Bir kontrata atılan isteklerin durumları, o kontratın tüm işlem bilgileri o olay içerisinden bulunabilir.

Sayısal işlemlere de dikkat edilmelidir. Solidity, tüm tam sayı bölmelerini en yakın tam sayıya ve aşağıya doğru yuvarlar. Kesinliğe ihtiyacınız varsa, bir çarpan kullanmak veya hem payı hem de paydayı saklamak gerekir.

```
uint rounded_uint = 5 / 2 ;  // rounded_uint 2 olacaktır.
```

Solidity dilinde “float” ve ya “double” sayılar için ileri sürümlerde “fixed” anahtar kelimesi kullanılacaktır. Henüz en aktif versiyonda desteklenmemektedir. “fixed” in “float” ve “double” değişken tipleriyle farkı; tutulacak küsüratların, determinizim gereği belli olması gerekliliğidir. Bu tip bölme işlemlerinde, işlemlerin zincir dışında (off-chain) yapılması önerilmektedir. Bunun gerçekleşmemesi durumunda; ondalık hassasiyeti (precision) çarpan değeri ile belirlenmelidir.

```
uint carpan = 10;
```

```
uint bolum = (5*carpan) / 2; // bolum = 25 olacaktır.
```

Transfer fonksiyonu çağıran bir cüzdanın, bakiye kontrolünü ele alalım. Değer transferleri (asset transfer) ciddiye alınmalı ve assert(), require() doğru kullanılmalıdır. assert() ve require() fonksiyonları birbirlerine benzer olsalar da derlenmiş kod (opcode) bakımından kullanım alanları farklıdır. Her iki fonksiyon da kontrol için kullanılmaktadır ve eğer doğruluk sağlanmıyorsa hata fırlatacak yapıda tasarlanmışlardır. assert() fonksiyonu iç hatalar (internal errors) ve değişmeyen veriler için kullanılmalıdır.

```
assert(address(this).balance >= balanceBeforeTransfer);
```

Yukarıdaki kod parçacığı şu anda içerisinde yer aldığımız kontrat adresine gönderilen Ether miktarı ile transfer öncesindeki miktarı karşılaştırmaktadır. “require” ise sadece değerlerin (girdi değişkenleri, harici çağrıdan dönen değerler, akıllı sözleşme durum değişkenlerinin değerleri gibi) doğru olup olmadığını kontrol etmek için kullanılmalıdır.

```
require(balanceBeforeTransfer >= 0, “Balance değeri 0 dan büyük veya eşit
olmalı”);
```

Kontrol değeri olan “balanceBeforeTransfer” ile burada her zaman 0 ‘dan büyük veya 0 ‘a eşit olma durumu denetlenecektir. Eğer doğruluk sağlanmıyorsa, bir sonraki parametredeki mesaj ile hata atılacaktır.

```
    function transfer(uint _amount, address to) public {

        require(_amount > 0,"Transfer tutarı 0'dan büyük olmalıdır");

        require(balanceOf[msg.sender] >= _amount,

             "Transfer tutarı cari hesap bakiyesinden büyük");

        // Temel toplama ve çıkarma işlemleri güvenli değildir

        balanceOf [to] += _amount;

        balanceOf [msg.sender] -= _amount;

        // bakiyeler için iç kontrol, hesapların güncellenmesi

        assert(balanceOf [msg.sender] + _amount >= balanceOf[to]);

    }
```

Yukarıda görüldüğü gibi, transfer fonksiyonu önce değer kontrollerini ve bakiye (balance) kontrollerini yapmaktadır. Bu “require” fonksiyonu ile yapıldığında olası hataların önüne geçilmiş olacaktır. Sonrasında yapılacak işlemler, bir toplama çıkarma fonksiyonu olabileceği gibi, bir harici çağrı da olabilir. Burada olası iç hataların önüne geçilmesi için; fonksiyon bitiminde “assert” ile tekrar doğrulama yapılarak işlemlerin sağlaması yapılmalıdır.

```
    modifier hasBalance(uint _amount) {

        require(_amount > 0,"Transfer tutarı 0'dan büyük olmalıdır");"

        require(balanceOf[msg.sender] >= _amount,

               "Transfer tutarı cari hesap bakiyesinden büyük");

        _; }

       // hasBalance fnks ihtiyaç olan her yerde kullanılabilecek

    function transfer_with_modifier(uint _amount, address to)
hasBalance(_amount) public {

        balanceOf[to] += _amount;

        balanceOf[msg.sender] -= _amount;

        // bakiyeler için iç kontrol, hesapların güncellenmesi

        assert(balanceOf[msg.sender] + _amount >= balanceOf[to]);

   }
```

“Modifier” lar sadece kontrol amaçlı kullanılmalıdır. Solidity “modifier” anahtar kelimesi ile fonksiyonlar birer ön çağrı mekanizması haline dönüştürülmektedir. Buradaki amaç, “modifier” kullanan fonksiyon öncesinde başka bir kontrol fonksiyonunun çağrılması ve bu sonuca etki edecek olası değişikliklerin kontrol edilmesidir. “Modifier”lar, bir fonksiyonun iç yapısında durum değişkenlerini değiştirilebildiği için dikkatli olunmalıdır ve bu yapının önce kullanılması önemlidir. Başka akıllı sözleşmelerin içerisindeki modifier fonksiyonlardan varolan koda iç aktarım (import) yapılacaksa; kullanılmadan önce kod güvenliği açısından gözden geçirilmelidir. Modifier fonksiyonların bir diğer amacı da kod okunurluğunu arttırmaktır. Modifer kullanımı yerine “require” ve “assert” fonksiyonlarını kullanabileceği de unutulmamalıdır.

1.5 Akıllı Sözleşmelere Saldırılar

Akıllı sözleşmeler, günümüzde finansal değer transferinde yaygın olarak kullanılmaktadır. Bu sözleşmelerdeki olası zafiyetlerin (vulnerability) kötüye kullanımı ve elde edilebilecek finansal getiri bilgisayar korsanlarının ilgisini çekmektedir. Bu bölümde bu konular hakkında temel bilgiler verilecektir. Akıllı sözleşmelerin zayıflıklarının sınıflandırılması ve testi için “SWC Registry” iyi bir kaynaktır[8]. Bu konuda bilinen saldırılar Consensys’in Github sayfasındaki en iyi uygulamalar dökümanında[9] anlatılmıştır. Akıllı sözleşme bazlı saldırılar ve buna karşı yapılabilecek korumalar literatürde [20-23] de ele alınmıştır. Marchesi ve arkadaşlarının çalışmasında [20]; akıllı sözleşmelere olan saldırı süreci

[8] SWC Registry - Smart Contract Weakness Classification and Test Cases, https://swcregistry.io/
[9] Ethereum Smart Contract Best Practices, Known Attacks, https://consensys.github.io/smart-contract-best-practices/known_attacks/

ayrıntılı olarak ele alınmış ve önerilerde bulunulmuştur. Akıllı sözleşmelerin güvenliği hakkında yapılan akademik çalışmalar ve olası araştırma konuları güncel bir yayında [21] incelenmiştir.
Akıllı sözleşmelere saldırılardan başlıcaları Tablo 1.1'de karşılaştırılmıştır. Ethereum ağına özel akıllı sözleşme saldırıları Tablo 1.2'de verilmiştir. Çözüm süreçlerinde güvenli kodlama prensiplerinin uygulanması önceki bölümde ele alınmıştı. Yapılabilecek çözümlere dair bazı ek öneriler de tabloda verilmiştir.

Tablo 1.1 Akıllı Sözleşmelere olan temel saldırılar[22]

Saldırı Adı	**Tür**	**Çalışma Şekli**	**Etkisi**	**Çözüm Önerisi**
Yeniden Giriş	Özelliğin kötüye kullanılabilirliği	Ana işlev bir döngü ile özyinelemeli (recursive) bir geri aramasını yürütür	Sözleşmeyi tamamen yok edebilir veya değerli bilgileri çalabilir.	Güvenli kod geliştirme prensiplerinin uygulanması
Akıllı Sözleşme Taşması (overflow) ve Azalması (underflow)	Yetkisiz (unauthorized) girdilerin kabulü	Saldırganın maksimum değerden daha fazla değer göndermesi veya minimum değerden daha az değer göndermesi	Elinde bulunandan çok daha fazla jetonu (token) sistemden çekebilir.	Güvenli kod geliştirme prensiplerinin uygulanması
Delegatecall	Özelliğin kötüye kullanılabilirliği	Bir sözleşmeyi çağırmak için kullanıldığında; yürütülen kod dışında, msg.sender ve msg.value değişmez. Yeniden kullanılabilir kod oluşturularak ani kod yürütme şansını artırır.	Özel kitaplıklar oluştururken kusurlar ortaya çıkabilir, yeni güvenlik açıklarına yol açabilir	- Kitaplık ve çağrı sözleşmelerinde bir gecikme gözlemlendiğinde akıllı sözleşmenin çağrılması - Mümkün olduğunca durumsuz kitaplıkların geliştirilmesi
Varsayılan Görünürlükler	Varsayılan Ayarların kötüye kullanılması	Görünürlük belirtecinin, kullanıcıların türetilmiş sözleşmelerle harici işlevleri aramasına izin verirken kontrolü ele alması.	Saldırgan kendi amacı için kodu kötüye kullanabilir.	Kod yazarken görünürlük belirteci özel olarak ayarlanmalı

--

Tablo 1.2 Ethereum Ağına Özel Akıllı Sözleşme Saldırıları[22]

Saldırı Adı	**Tür**	**Çalışma Şekli**	**Etkisi**	**Çözüm Önerisi**
Kısa Adres Saldırısı	Girdi onaylama hatası	Ethereum Sanal Makinesi (EVM) zafiyetinde; saldırgan kasıtlı olarak adresin sonundaki “0” değerini göndermez.	Ciddi sorunlara yol açabilir	Girdi kontrolü yapılması
İşlem Sırasına Bağlılık (TOD)	Protokolün kötüye kullanılması	Bir işlemdeki akıllı sözleşme, bir diğer işleme bağlı çalışıyorsa; bloğu oluşturan madenci, bloktaki işlemlerin sırasını etkileyebilir.	Ethereum ağında bazı işlemler kasıtlı olarak çalıştırılmayabilir.	Otonom kodlarla kötüye kullanımın kontrolü
Zaman Damgası Bağımlılığı	Protokolün kötüye kullanılması	Akıllı sözleşme, madencilerin blok doğrulamasından sonraki 30 saniye içinde bir zaman damgası koymasına izin verdiğinden, bu değer fayda sağlamak için değiştirilebilir.	Ethereum ağı ve benzer çalışan ağlarda; madencilerin haksız kazanç elde etmesi.	Zaman damgası ile ilgili protokolün iyileştirilmesi

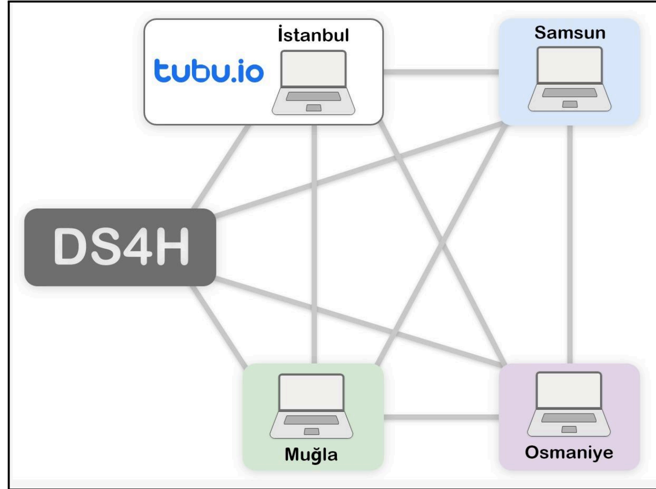
1.6 Blokzincir Testleri

Blokzincir ağlarının ve merkezi olmayan uygulamaların başarım ölçütleri yeterince test edilmemektedir. Bölümde öncelikle blokzincir test ağlarından söz edilecektir. Yapılabilecek testler; blokzincir ağının performans testleri ve akıllı sözleşme testleri olarak ele alınacaktır. Geliştiricilerin kendi akıllı sözleşme test araçlarını oluşturmaları için örnek verilecektir. Güvenlik analiz araçlarından söz edilecektir.

1.6.1 Blokzincir Test Ağları

Yazılımları gerçek blokzincir ağında denemeden önce, var olan deneme ağları (tesnet) üzerinde denenmesi gerekecektir. Her platform için çeşitli bulut tabanlı deneme ağları (testnet) bulunmaktadır. Ethereum dışındaki platformlara ait test ağlarında özel bir öğrenme süreci ve kullanım ücreti gerekmektedir. Blokzincir test ağları genel kayıt defterleri kullanır ve bu kayıtların görünür olması yüzünden birçok proje süreçlerinde tercih edilmez. Geliştirici sürecin gizli kalmasını istediğinde kendisine özel bir ağ kurması gerekecektir. Günümüzde bu tür ağlar docker gibi konteyner (container) teknolojileriyle çok kısa bir sürede kurulabilmektedir. Geliştirici kendi makinesinde veya bulut ortamında belirli sayıda düğümü çalıştırarak test işlemlerini gerçekleştirir.

Test ağları, ölçeklenebilirlik ve performans açısından esnek değildir ve başarım testlerinde gerçekçi sonuçlar vermezler. Gerçekçi bir test ortamı oluşturmak için fiziksel bir test ağına ihtiyaç duyulmaktadır. Merkezi olmayan teknolojilerin araştırılıp geliştirilebileceği sürdürülebilir ve ölçeklenebilir bir araştırma ortamı olarak DS4H (Decentralized solutions for humanity - İnsanlık için Merkezi Olmayan Çözümler) kurulmaktadır. Şekil 1.15'de ağın kurulan ilk düğümleri gösterilmiştir. ISOC (Internet Society)'dan alınan hibe desteği ile DS4H'ın ilk deneme düğümleri Türkiye'nin dört farklı şehrinde (İstanbul, Samsun, Muğla, Osmaniye) konumlandırılmıştır.



Şekil 1.15 DS4H Blokzincir Test Ağ Yapısı[5]

DS4H blokzincir araştırma ağının testleri ve yayın süreçleri sürmektedir [24]. Ağın otonom kodlarla yönetimi için çalışmalar devam etmektedir. Projenin bir sonraki aşamasında, araştırma kurumlarının ağa dahil olarak araştırmacılarını bu ağdan yararlandırması mümkün olacaktır. Bu araştırma ağına ait gelişmeler için DS4H proje sayfası[10] ve sosyal medyada ds4h etiketi (hashtag) takip edilebilir.

1.6.2 Blokzincir Ağının Başarım Testleri

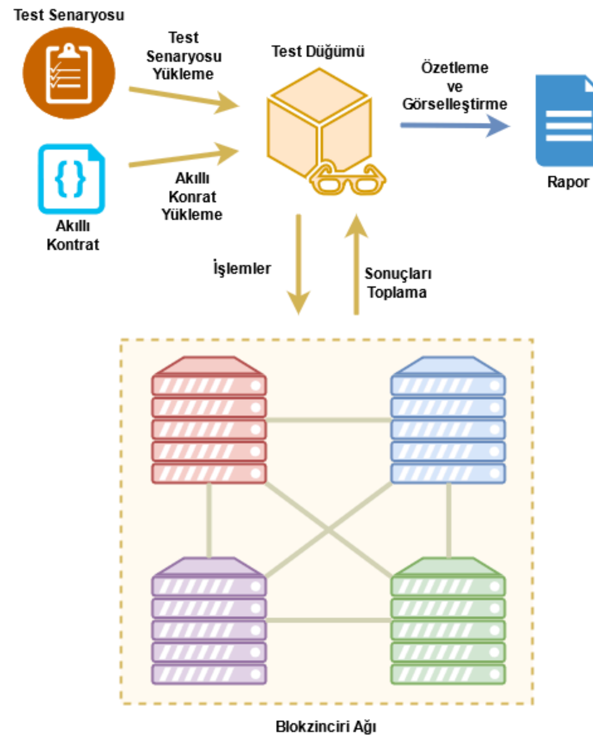
Blokzincir ağının başarımına (performance) dair bir fikir elde edebilmek için; gecikme (latency), birim zamanda üretilen iş (throughput), kaynak kullanımı ve başarısız/gecikmiş işlem değerleri ölçülmelidir. Gecikme değerleri olarak blokzincirinde bir işlemin gerçekleşmesindeki gecikme (transaction latency)

[10] Decentralized solutions for humanity (DS4H) blokzinciri araştırma ağı proje sayfası, http://wiki.netseclab.mu.edu.tr/index.php?title=DS4H

--

ve kayıtlardan okumadaki gecikme (read latency) ölçülür. Sistemde birim zamanda üretilen iş olarak da benzer süreçlerin hesaplamaları yapılır. Sistemdeki düğümlerin bu süreçler boyunca kaynak (işlemci, bellek, ağ, vb.) kullanımı da ölçülür [6]. Zaman aşımları nedeniyle oluşan başarısız/gecikmiş işlemlerin sayısı da takip edilmesi gereken önemli bir parametredir [25].

Başarım değerlendirmesi için kullanılabilecek örnek test süreci Şekil 1.16'da gösterilmiştir. Test edilecek blokzincir ağı ile iletişim kuracak en az bir test düğümü olmalıdır. Test senaryosu; test işlemlerinin kapsamını içeren bir konfigürasyon dosyasıdır. Sistem uygulamaya dair özel bir akıllı sözleşme ile test edilebileceği gibi bu süreçte varsayılan (default) bir akıllı sözleşme de kullanılabilir. Test düğümü; senaryoya göre sistemde yükü oluşturur ve ardından sonuçları gözlemler. Test sonuçları özetlendikten sonra görselleştirilerek rapor oluşturulur [5]. Düğümlerin kaynak kullanımı düğümlerin kendilerinde kurulacak sistemlerle veya SNMP (Simple Network Management Protocol) gibi protokoller aracılığıyla da takip edilebilir.



Şekil 1.16 Blokzincir Ağ Başarım Test Süreci [6]

Blokzincir ağlarında kullanılabilecek çeşitli blokzincir ağ başarım test araçları bulunmaktadır, ancak araçların çoğu belirli blokzincir platformları için oluşturulmuştur ve karmaşık konfigürasyon gerektirir. Günümüzde en kapsamlı başarım kıyaslama ortamı Hyperledger Kaliper[11]'dir. Quorum dışında birçok platformu desteklemektedir. Blokzinciri ağ başarım testlerine dair örnek çalışmalar [25-28]'da verilmiştir. Quorum ortamında Chainhammer[12] ve Quorum Profiling[13] gibi başarım test araçları bulunmaktadır. Bu araçların kullanımları kolay değildir, karmaşık kurulum adımlarına ihtiyaç duyulmaktadır. Chainhammer'ın bazı gerekli bağımlılıkları güncel değildir [6]. GoHammer aracı, Ethereum/Quorum blokzincir platformları için kullanımı kolay, esnek bir test aracı sağlamak üzere geliştirilmiştir. Saniyede işlem (TPS) değerleri ve çeşitli başarım ölçümlerinin test edilmesi için kullanılabilmektedir. GoHammer üzerindeki geliştirmeler devam etmektedir. Detaylı bilgi için bkz [6].

[11] Hyperledger Kaliper, https://hyperledger.github.io/caliper/
[12] Chainhammer, https://github.com/drandreaskrueger/chainhammer
[13] Quorum Profiling, https://docs.goquorum.consensys.net/en/stable/Concepts/Profiling/

--

1.6.3 Akıllı Sözleşme Test Araçları

Aktif olarak hem geliştirmesi hem de desteği süren çeşitli akıllı sözleşme test servisi ve araçları da mevcuttur. Bunlardan başlıcaları olarak OpenZeppelin[14], hardhat[15], waffle[16] projelerinden söz edilebilir. OpenZeppelin geliştiricilere akıllı kontratlar için bir çatı ve test ortamları için bir kontrol kütüphanesi sunmaktadır. OpenZeppelin kütüphanesinde yer alan temel kontratlar yardımıyla daha önce test edilmiş ve testleri geçmiş bilinen akıllı kontratları kullanmak, test kodlarını azaltacağı gibi basit tasarım yaklaşımımızla da örtüşmektedir. Hardhat; derleme, konuşlandırma, test ve hata ayıklama kısımlarında işe yarayan çok yetenekli bir javascript kütüphanesidir. Hardhat kütüphanesi içerisinde bir Ethereum ağıyla gelmekte ve yapılan tüm işlemler için onu kullanmaktadır. Bu kütüphanenin temel işlevselliği; yığın izleri, günlüğe alma (logging) ve işlemler başarısız olduğunda açık hata mesajları içeren Solidity hata ayıklamasıdır. Waffle ise daha çok javascript kütüphanesi olarak karşımıza çıkmaktadır. Hardhat ile uyumlu çalışması sayesinde, bir test aracı olarak akıllı kontratlar haricinde yapılabilecek testler için temel oluşturmaktadır.

1.6.4 Kendi Akıllı Sözleşme Test Kontratınızı Oluşturmak

Geliştiriciler kendi test kodlarını da yazabilirler. Böyle bir kontrat her türlü genel ve özel ağlarda test amaçlı kullanılabilmelidir. Bunun için ek bir araç ya da özel bir yazılıma ihtiyaç duyulmamalıdır. Bu bölümde buna bir örnek verilecektir. Aşağıdaki kod parçacığı; tüm Solidity örneklerinde kullanılan “Simple Storage” kontratıdır. Örneklerde bizlere gösterilen sadece kontratın kendisidir, Solidity dilini kullanarak bu tip basit bir kontratın testini yapmak ise oldukça karmaşık olabilir. Truffle gibi araçlarla farklı ortamlarda farklı şekillerde testler yapılabilse de, Solidity dilinde yazılmış kodların (en basit anlamıyla) testi için sadece “assert” yeterli olabilmektedir.

```
contract SimpleStorage {
  uint public storedData;

  constructor() public {
    storedData = 100;
  }

  function set(uint x) public {
    storedData = x;
  }

  function get() public view returns (uint retVal) {
    return storedData;
  }
}
```

Yukarıdaki kontrat, oluşturulduğunda içerisindeki bir değişkene 100 değeri atamaktadır. Bu değer daha sonra “set” fonksiyonu ile de değişebilmektedir. Ayrıca kayıt edilen bu değişkenin değerini bize gösteren fonksiyon olarak da ”get” kullanılmaktadır. Bu tip bir kontratı test edebilmek için başka bir test kontratı yazacağız. Bunun için o kontratı yazacağımız test kontratı içerisinde oluşturmamız gerekiyor. Böylelikle deterministik bir test kontratımız olacak. Aşağıda Remix IDE ile de kullanabileceğiniz “SimpleStorageTest” isimli test kontratı verilmiştir.

[14] OpenZeppelin, https://github.com/OpenZeppelin
[15] Hardhat, https://hardhat.org
[16] Waffle, https://getwaffle.io/

```
import "SimpleStorage.sol";
contract SimpleStorageTest {

  // Test edilecek kontrat değişken olarak tanımlanır
  SimpleStorage storage_test;
  // basit bir toggle değeri ile test fonksiyonlarının fail etmesi sağlanır
  uint i = 0;

  function beforeEach() public {
    // test edilecek kontrat burada oluşturulur
    storage_test = new SimpleStorage();
    if (i == 1) {
        storage_test.set(200);
    }
    i += 1;
  }

  function initialValueShouldBe100() public {  // ilk değerin 100 olması testi
        beforeEach();
        // değeri 100 olarak bekliyoruz, ilk seferde geçecektir.
        // 2. sefer de fail
        assert(storage_test.get() == 100);
  }
  // set fonksiyonun çağrılması ve değeri 200 yapma testi
  function valueShouldBe200() public {
      beforeEach();
      assert(storage_test.get() == 200);
  }
}
```

"beforeEach" fonksiyonu tüm alt test fonksiyonlara eklenmiş ve fonksiyon isimleri yapılacak testin içeriğini belirtecek şekilde ayarlanmıştır. Böylelikle hangi fonksiyonda neyi test ettiğimizi görme imkanı elde edilmektedir. Farklı bir yöntem olarak "constructor" oluşturularak diğer test fonksiyonları da oradan çağırılabilir. Gelişmiş programlama dillerinde de benzer "main" fonksiyonlarla test etme yöntemleri kullanılmaktadır. Hali hazırda yazılan akıllı sözleşmeleri çok kompleks tutmadan ilerleme tavsiyemiz ise, araçlara gerek kalmadan da bu testlerin yazılabileceğini göstermek amacıyla verilmiştir.

```
import "SimpleStorage.sol";
contract SimpleStorageTest {

  SimpleStorage storage_test;
  uint i = 0;

  constructor() public{
      storage_test = new SimpleStorage();
      initialValueShouldBe100();
      storage_test.set(200);
      valueShouldBe200();
  }
  function initialValueShouldBe100() private {
        assert(storage_test.get() == 100);
  }
  function valueShouldBe200() private {
      assert(storage_test.get() == 200);
  }
}
```


--

1.6.5 Güvenlik Analiz Araçları

Güvenlik analiz araçları üzerine ayrıntılı bir inceleme olarak; Durieux ve arkadaşların çalışması [32] bulunmaktadır. Bu çalışmada; belli başlı dokuz otomatik analiz aracının geçerliliği güvenlik açığı olan 69 akıllı sözleşme üzerinde denenmiştir. Sonrasında Etherscan'de bulunan 47,587 akıllı sözleşme üzerinde denetlenme gerçekleştirilmiştir. Bu deneylere göre; bu akıllı sözleşmelerinin %97'isinde zafiyet (vulnerability) bulunmuştur. Denenen araçların dört veya daha fazlasının aynı anda bulduğu zafiyetler ise azdır. Özetle, ancak birden fazla aracı çalıştırdığımızda anlamlı bir sonuç elde edilebilmektedir.

Ethereum'da akıllı sözleşme güvenlik analizi süreçlerinde kullanılabilecek belli başlı güvenlik aracı olarak Slither[17], Echidna[18], ve Manticore[19] den söz edilebilir. Slither, Solidity statik analiz aracıdır, sözleşme ayrıntıları hakkında görsel bilgiler yazdırır ve özel analizleri kolayca gerçekleştirmek için bir API sağlar. Slither, geliştiricilerin güvenlik açıklarını bulmasını ve özel analizleri hızlı bir şekilde oluşturabilmesini sağlar [29]. Slither, CI/CD (continuous integration - sürekli entegrasyon / continuous development - sürekli geliştirme) sisteminize, örneğin Github Actions'a entegre edilerek kullanılabilir.

Echidna, Ethereum akıllı sözleşmelerin bulandırma (fuzzing) ve özellik tabanlı testi (property-based testing) için tasarlanmış bir Haskell programıdır. Kullanıcı tanımlı tahminleri (predicates) veya Solidity iddialarını (assertions) yanıltmak (falsify) için bir ABI (Application Binary Interface - Uygulama İkili Arayüzü) sözleşmesine dayalı karmaşık dil bilgisi tabanlı bulandırma yöntemleri kullanır. Kodu sözde rasgele (pseudo random) olarak üretilen işlemlerle çalıştırır. Bulandırma aracı, belirli bir özelliği ihlal etmek için bir dizi işlem bulmaya çalışacaktır.

Manticore [31], akıllı sözleşmelerdeki hataları otomatik olarak bulmak için kullanılır. Sembolik çalıştırma (symbolic execution) ortamı sunmaktadır. Her çalıştırma yolunu (execution path) matematiksel bir formüle çeviren ve üzerinde üst (top) kısıtlamaların denetlenebileceği formal bir doğrulama tekniği kullanılabilmektedir. API aracılığıyla akıllı sözleşmeler manipüle edilebilir, kısıtlamalar eklenebilir.

Bu araçların ve kullandıkları yöntemlerin kıyaslaması Ethereum'un resmi sayfasında[20] verilmiştir. En yaygın kullanılan üç ana güvenlik test ve analiz tekniğinin kıyaslanması Tablo 1.3'de verilmiştir. Statik analiz yönteminde test süreci çok kısa (saniyeler) sürmektedir, kaçırılan hatalar (bug) ortalama miktardadır, yanlış alarm oranı düşüktür. Bulandırma yönteminde test süreci kısadır (dakikalar), kaçırılan hatalar (bug) düşük miktardadır, yanlış alarm oranı yoktur. Sembolik çalışma yönteminde test süreci saatler sürmektedir, kaçırılan hata ve yanlış alarm oranı yoktur.

Tablo 1.3 Üç Ana Güvenlik Test ve Analiz Tekniğinin Kıyaslanması

Teknik	Araç	Kullanımı	Çalışma Süresi	Kaçırılan hatalar	Yanlış Alarm Oranı
Statik Analiz	Slither	Komut satırı, betik	Saniyeler	Ortalama	Düşük
Bulandırma	Echidna	Solidity özellikleri	Dakikalar	Düşük	Yok
Sembolik Çalıştırma	Manticore	Solidity özellikleri, betik	Saatler	Yok	Yok

[17] Slither, https://github.com/crytic/slither
[18] Echidna, https://github.com/crytic/echidna
[19] Manticore, https://github.com/crytic/building-secure-contracts/tree/master/program-analysis/manticore
[20] A guide to Smart Contract Security Tools, https://ethereum.org/hr/developers/tutorials/guide-to-smart-contract-security-tools/

1.7 Güvenlik Kontrol Listesi

Güncel bir çalışmada [20]; güvenlik riskleri ele alınmış ve bir güvenlik denetim listesi önerilmiştir. Ethereum'un resmi sayfasında bulunan güvenlik listesinde[21] de pratiğe yönelik öneriler bulunmaktadır. Bu alt başlıkta aksi belirtilmedikçe bu kaynaklardaki içeriklerden yararlanılmıştır. Bu kaynaklardan ve deneyimlerimizden yararlanılarak kategoriler oluşturulmuş ve geliştiriciler için güncel bir güvenlik kontrol listesi Tablo 1.4'te sunulmuştur. Ana kontroller, kısıtlamalar, ek destekler, yazılım mühendisliği pratikleri, testler ve belgeleme olarak kategoriler tanımlanmıştır. Bu tablonun en güncel hali proje github sayfasından[22] erişilebilir olacaktır.

Güvenlik için öncelikle kontratta mantık kontrolü yapılmalıdır. Taşmalardan (overflow), yetersizliklerden (underflow) veya sonlu aritmetiğin diğer istenmeyen özelliklerinden korunmak için bir mantık oluşturulmalıdır. Koruma kontrolü (guard check) de gerçekleştirilmelidir. Akıllı sözleşmenin durumu ve fonksiyon girdilerine dair tüm gereksinimlerin karşılandığından emin olunulmalıdır.

Fonksiyon etkileşimleri kontrol edilmelidir. Bir sözleşmede bir fonksiyonu gerçekleştirirken öncelikle tüm ön koşullar kontrol edilmelidir. Sonrasında sözleşmenin durumuna etkiler (effect) uygulanmalıdır. En son aşamada diğer sözleşmelerle etkileşime geçilmelidir.

Mahremiyetin sağlanıp sağlanmadığı kontrol edilmelidir. Karaarslan ve Konacaklı'nın çalışmasında [7] da belirtildiği üzere, blokzincirinde kişisel veri tutulması tavsiye edilmemektedir. Sıfır bilgi kanıt (zero knowledge proof) [15] protokollerinin kullanımı yerinde olacaktır. Yine de kişisel verilerin tutulması gerektiğinde; bu tür veriler şifrelenerek kayıt defterinde saklanmalıdır. KVKK (kişisel verileri koruma kanunu) [33], GDPR (general data protection regulation - genel veri koruma yönetmeliği) [34] gibi yasal gereksinimler karşılanmalıdır. Bu gereksinimleri karşılarken coğrafi konumdan ziyade, hangi ülkelerin vatandaşının kişisel verilerinin tuttulduğu önemlidir. Örneğin Avrupa Birliği bir ülkenin vatandaşına ait kişisel veriler tutuluyorsa GDPR gereksinimleri karşılanmalıdır.

Yetkilendirmelerin gerçekleştirilmesi kontrol edilmelidir. Kritik metodlar ancak belirli kullanıcılar tarafından yürütülmelidir. Bu da adreslerin eşlenmesi kullanılarak gerçekleştirilir ve "modifier" kullanılarak kontrol edilir. Bunun için sahiplik de tanımlanmalıdır. Sözleşme yönetiminden sorumlu olan ve özel izinlere sahip olan sözleşme sahibi belirtilmelidir. Bu da muhtemelen kritik metodları çağırmaya yetkilendirilen o tek adres olacaktır.

Akıllı sözleşmenin sonlandırması denetlenmelidir. Bir akıllı sözleşmenin kullanım ömrü sona erdiğinde sonlandırılmalıdır. Bir sözleşmeyi feshetme yetkisi genellikle sözleşme sahibindedir. Sözleşmeye önceden tasarlanmış (ad-hoc) kod ekleyerek veya "selfdestruct" (kendi kendini yok etme) işlevini çağırarak bu gerçekleştirilir.

Sözleşmelerin özel durumları da göz önünde bulundurulmalıdır. Sözleşmelerin yükseltilebilir olup olmadığı kontrol edilmelidir. Sözleşmelerin ERC (Ethereum Request for Comments - Ethereum yorumlar için rica) 'lere uygunluğu denetlenmelidir. ERC'ler Ethereum kullanımında belirli bir standardı tanımlamak için yapılan EIP (Ethereum Improvement Proposal - Ethereum iyileştirme önerisi)'lere verilen etiketlerdir. ERC-20 jeton (token) standardı buna örnek verilebilir. Üçüncü taraf jetonlarıyla entegrasyon söz konusu ise; ilgili denetim listesine[23] göre davranmak yerinde olacaktır.

Kısıtlamaların gerçekleştirimi kontrol edilmelidir. Bakiye limiti kullanarak bir akıllı sözleşme içinde tutulan maksimum fon (fund) miktarı sınırlanmalıdır. Oran sınırlaması (rate limit) gerçekleştirilmelidir. Bir akıllı sözleşmeye gönderilen mesaj sayısını ve dolayısıyla hesaplama yükünü sınırlamak için; bir

---

[21] Smart Contract Development Checklist,
https://ethereum.org/hr/developers/tutorials/secure-development-workflow/

[22] MSKÜ Smart Contract Security Testing, https://github.com/MSKU-BcRG/SC-SecTesting

[23] Token integration checklist,
https://ethereum.org/hr/developers/tutorials/token-integration-checklist/

görevin belirli bir süre içinde ne sıklıkta yürütülebileceği düzenlenmelidir. Hassas görevlerde hız kısıtlaması (speed bump) uygulayarak süreç yavaşlatılır. Kötü niyetli eylemlerin gerçekleşmesi durumunda hasar sınırlı olacak ve buna karşı önlem almak için daha fazla zamanınız olacaktır. Zaman kısıtlaması yaparak eyleme ne zaman izin verileceği de belirtilebilir. Bu da işlemi tutan bloktaki kayıtlı zamana dayanacaktır. Zaman kısıtlaması; hız kısıtlaması ve oran sınırlaması yapılarıyla da birlikte kullanılabilir.

Vekil (proxy) ve kahin (oracle) gibi ek desteklerin kullanımı da mümkündür. Vekil temsilcileri veya vekil kalıpları (proxy patterns); akıllı sözleşmelerin sürüm yükseltilmesini kolaylaştırmak için kullanılabilecek bir dizi akıllı sözleşmedir. Vekil, aslında adresi değiştirilebilen bir başka akıllı sözleşmedir. Vekil kullanımında yeni akıllı sözleşmeyi gösterecektir. Bu da blokzincir kaynaklarının idareli kullanılmasını sağlayarak gas tasarrufu sağlamaktadır. Kahin, blokzinciri dışından veri sağlayan bir akıllı sözleşmedir. Kahin güvenilir bir kaynak tarafından veri ile güncellenmektedir. Ters (reverse) kahin olarak kullanımında; akıllı sözleşmenin zincir dışı (off-chain) bileşenlere veri sağlaması söz konusudur.

Yazılım mühendisliği pratikleri kontrol edilmelidir. CI/CD (sürekli entegrasyon ve sürekli geliştirme) süreçlere dahil edilmelidir. Geliştirilen uygulama ya da yazılım eğer genel ağlarda çalışacaksa, anahtar bilgileri paylaşımı doğru olmayacağı için CI/CD süreçleri için çevrimiçi (online) servis kullanımı doğru olmaz. Bu süreçleri kontrol etmek için olay (event) yapısının takip edilmesi için bir yapı kurulması tavsiye edilir. Bunun yanı sıra; yeniden kullanılabilirlik yapılarından ve mutex'den yararlanılması iyi bir pratiktir. Çoklu oluşumlar (multiple instance) için sözleşme kitaplıkları ve şablonları kullanılır. Mutex, paylaşılan bir kaynağa eşzamanlı erişimi kısıtlamak için kullanılan bir mekanizmadır. Harici bir çağrının (external call); onu çağıran fonksiyona (caller function) yeniden girmesini engellemek için kullanılmalıdır.

Bir önceki bölümde ele alınan güvenlik analiz araçları ile güvenlik testleri gerçekleştirilmelidir. Bilinen güvenlik sorunlarını kontrol etmek için Slither, Echidna, Manticore araçları ve benzerleri ile sözleşmeler sürekli olarak gözden geçirilmelidir. Otomatik araçların her türlü sorunu bulamayacağının farkında olmak gerekir. Özellikle mahremiyet, önden giden (front running) işlemler, kriptografik işlemler ve harici DeFi (decentralized finance - merkezi olmayan finans) bileşenleriyle riskli etkileşimlere dikkat edilmelidir. Yazılımı oluşturan parçaların gerektiği gibi çalıştığından emin olmak için birim testleri (unit test) de gerçekleştirilmelidir.

Belgeleme en önemli aşamalardandır. Kodun kritik güvenlik özellikleri öncelikle görsel olarak incelenmelidir. Bu süreçte Slither'ın "inheritance-graph" (miras grafik) oluşturucusu kullanılabilir. Fonksiyon görünürlüğünü ve erişim kontrollerini raporlamak için Slither'ın işlev özeti (function-summary) oluşturucusundan yararlanılabilir. Durum değişkenleri üzerindeki erişim kontrollerini raporlamak için Slither'ın değişken ve yetkilendirmeli (vars-and-auth) oluşturucusu kullanılabilir. Kodun kritik güvenlik özellikleri belgelenmelidir ve bunları değerlendirmek için otomatik test oluşturucularından yaralanılabilir. Öncelikle kod için güvenlik özelliklerinin belgelenmesi öğrenilmelidir. Farklı araçlar kullanılarak; sonlu otomata (state machine), erişim kontrolleri, aritmetik işlemler, harici etkileşimler, standartlara uygunluk, kalıtım (inheritance), değişken bağımlılıkları (dependencies), erişim kontrolleri ve diğer yapısal konulara odaklanılmalıdır.

Tablo 1.4 Akıllı Sözleşme Güvenlik Kontrol Listesi

Güvenlik Kontrol Listesi	**Yapılma Durumu** ☑	**Notlar**
Mantık Kontrolü	☐	
Koruma Kontrolü	☐	
Fonksiyon Etkileşimleri Kontrolü	☐	
Mahremiyet	☐	
Yetkilendirme ve Sahiplik	☐	
Sonlandırma	☐	
Yükseltilebilirlik ve Özel Durumlar	☐	
Kısıtlamalar		
Bakiye Limiti Kullanımı	☐	
Oran Sınırlandırması	☐	
Süreçlerin Yavaşlatılması	☐	
Zaman Kısıtlaması	☐	
Ek Destek		
Vekil Temsilcisi Kullanılması	☐	
Kahin Kullanımı	☐	
Yazılım Mühendisliği Pratikleri		
Sürekli Entegrasyon (CI)	☐	
Sürekli Geliştirme (CD)	☐	
Yeniden Kullanılabilirlik	☐	
Mutex Kullanımı	☐	
Testler		
Araçlarla Güvenlik Testleri	☐	
Birim Test	☐	
Kritik Güvenlik Özelliklerinin Görselleştirmesi ve Belgelenmesi		
Görsel Olarak İnceleme	☐	
Belgeleme	☐	

--

1.8 Blokzincir Ortamında Yazılım Geliştirmede Sıkıntılar ve Fırsatlar

Yeni gelişen bu teknoloji için yazılım geliştirme süreçlerinde sıkıntılar yaşandığının farkında olmak gerekir. Blokzinciri ortamında yazılım geliştirmede karşılaştıkları ve gözlemledikleri sıkıntıları MSKÜ Blokzincir Araştırma Grubu (MSKÜ BcRG) üyelerine sorduk. Bu konuda gördükleri fırsatları da iletmelerini istedik. Bu bölümde; sekiz üyeden aldığımız yanıtları özetleyerek çıkarımlarda bulunacağız.

Öncelikle blokzinciri konusunda kaynaklar yetersiz. Internet üzerindeki farklı bilgilerin hangisinin doğru ve güncel olduğu da çok net değil. Dökümanlardaki örnek kodların da oldukça temel düzeyde olduğunu gözlemliyoruz. En iyi uygulama (best practice) örnekleri yok denecek kadar az. Örnek projeler de yetersiz, özellikle bitmiş proje bulmak zor. Bazı kaynaklar da oldukça pahalı olabiliyor. Ucunda finansal getiri olma olasılığı, bu işin ticaretine yol açabiliyor. Blokzincir teknolojisine dair danışılabilecek bilgili kişiler az. Geliştirici topluluğu yeterince gelişmiş değil. Gelişmiş bir blokzincir ekosisteminden söz etmek ise şu anda mümkün değil. Blokzincir teknolojisinde standartlar oturmadığı için, sürekli tekrardan bir şeyleri öğrenmek gerekiyor. Örneğin web teknolojilerinde API geliştirirken sadece o yeni platformu öğrenmek yeterli olurken, blokzincir ortamlarında ise yeniden öğrenmek gerekiyor. Test araçları yetersiz; kurulum ve kullanımları da çoğunlukla kolay değil.

Bu gözlemlediğimiz problemleri çözmek için araştırma, geliştirme ve billinçlendirmeye dayalı çalışmalara devam ediyoruz. Özellikle blokzincir araştırma ağlarının daha kolay kullanılabilir olmasının sağlanması gerektiğini düşünüyoruz. Sürdürülebilir araştırma ağları için birlikte çalışmamız gerekiyor. Bu konularda daha çok akademik çalışmaya ve fon desteğine ihtiyaç var. Bir blokzinciri ekosisteminin gelişmesi için; bu konuda çalışacak yazılım geliştirme ekiplerine yeterli ücret ve imkan sağlanması gerekiyor. Fırsatlar ise birçok alanda devrim yaratacak değişiklikler ve getirileri olacaktır.

Şafak Öksüzer şu anda blokzinciri sistemlerinde mahremiyet sağlanması üzerine yüksek lisans yapıyor ve kendisinin fırsatlara dahil yorumları: “Birçok farklı sektörde blokzinciri uygulamalarını görmeye başladık. Özel sektör uygulamalarının çoğu kapalı ve izinli blokzinciri yapıları üzerinde geliştirilmektedir. Ethereum gibi açık ve izinsiz yapılarda uygulama geliştirmenin maliyetinden ötürü, yakın zamanda kapalı ve izinli blokzinciri yapılarına olan ihtiyacın daha da artacağını düşünüyorum. Kripto paraya ihtiyaç duymayan, daha az enerji tüketen, kapalı ve izinli blokzinciri yapılarının önemi artacaktır. Bu şartlar altında akıllı sözleşme güvenliği daha fazla göz önüne gelecektir. Güvenli akıllı sözleşme geliştirmek uzmanlık gerektiren bir meziyettir. Bunun için akıllı sözleşmelerin çalışma mantığını, dağıtık sistemler ve determinizm kavramlarını iyi kavramış geliştiricilere ihtiyaç duyulacaktır. Akıllı sözleşme geliştiricisi olma konusunda önümüzde büyük fırsatların olduğunu rahatlıkla söyleyebilirim.”

Ahmet Önder Gür şu anda finans yazılımı geliştiren bir şirkette çalışıyor. Onun fırsatlara dair yorumları: “Kripto paraların ve kripto borsaların yükselişi, popülerleşmesi ile birlikte finans yazılımlarında daha çok kripto para, kripto borsa desteği görmeye başlayacağımızı düşünüyorum. Finans yazılımı geliştiren bazı şirketler; kripto borsalarda yapılan usulsüzlükleri bulmak için zincir incelemesi vb yazılımları geliştiriyorlar. Bunun gibi, blokzincir geliştiricisi olmaktan ziyade, işin finans tarafında, usulsüzlüklerin incelenmesi bulunması tarafında, daha fazla geliştiriciye ihtiyaç olabileceğini düşünüyorum.”

1.9 Sonuç

Bu bölümle merkezi olmayan çözümlerinin nasıl güvenli ve güvenilir şekilde geliştirileceği konusunda bir ön bilgi verilmiştir. Sürekli gelişmekte olan blokzincir dünyasına dair güncel bilgiler sunulmuştur. Akıllı sözleşmelerde güvenlik süreçlerine ve gas optimizasyonuna dikkat edilmelidir. Blokzincir tabanlı sistemlerin yazılım geliştirme süreçlerinde yaşanan ve aşılması gereken sorunlar konusunda da çözüm önerileri paylaşılmıştır. Bölümde verilen örneklerin en güncel haline SC-SecTesting Github reposundan (https://github.com/MSKU-BcRG/SC-SecTesting), çizimlere de BlockchainDiagrams Github reposundan (https://github.com/MSKU-BcRG/BlockchainDiagrams) ulaşılabilir.

Yönetilebilir ağ servisleri kullanımı; akıllı sözleşme geliştirilmesi, blokzincir sistemine yüklenmesi ve test edilmesi konusunda hem süreçleri kolaylaştıracak hem de hızlandıracaktır. Geliştirmekte olduğumuz Tubu-io ve GoHammer yazılımlarının bu anlamda bir fark yaratacağını düşünüyoruz. Bu yazılımlar halen birçok ArGe projesinde ve MSKÜ Bilgisayar Mühendisliği bölümündeki eğitim ve akademik araştırmalarda kullanılmaktadır. Akıllı sözleşmelerin güvenlik testleri konusunda çalışmaya devam etmekteyiz.

Blokzinciri alanında yetişmiş uzman ihtiyacı bulunmaktadır. Birçok ülke bu konuda çalışmalarda bulunmaktadır. Çin devleti kalkınma planlarına bu konuda eğitim ve geliştirmeleri eklemektedir. Kobilerin düşük maliyetlerle geliştirme yapabilmeleri için BSN (Blockchain Service Network - https://bsnbase.io/) blokzincir ağının kurulmasına destek verilmektedir[24]. Ülkemizde ise Tübitak Bağ ve DS4H blokzincir ağları gelişmekte olan yapılardır. Blokzincir ağ test ortamlarının güvenilir ve sürdürülebilir olması sağlanmalıdır. Bu kapsamda DS4H blokzincir ağını geliştirmeye devam ediyoruz.

İnsanoğlunun kararları önyargılıdır; herhangi bir süreç, dahil olan insana, o insanın bulunduğu duygu durumuna ve zamana göre farklı işleyebilir. Kurallara göre işleyecek deterministik bir dünya için akıllı sözleşmelerin daha yaygın kullanımı gerekecektir. Yapay zeka ile akıllı sözleşmelerin entegrasyonu, blokzincir yapılarının güvenliği alanında birçok potansiyel çalışma konusu bulunmaktadır. Blokzincir tabanlı sistemlerde ölçeklenebilirlik (scalability) ve başarım açısından çözülmesi gereken birçok problem ve fırsat (challenge) bulunmaktadır. Veri bilimi açısından bu sorunları ve çözüm önerilerini önceki çalışmamızda [12] ele almıştık. Blokzincir sistemlerini bir alternatif çözümden daha çok tamamlayıcı bir çözüm olarak görmek daha yerinde olacaktır. Blokzincir sistemlerinin var olan çözümlere, bulut yapılarına ve IPFS (Interplanetary File System) gibi dağıtık dosya sistemlerine entegrasyonu ile karma sistemler kurulabilir. Bu süreçlerde; ölçeklenebilirlik, birlikte çalışabilirlik ve mahremiyet için önerdiğimiz MPISA (Multi Platform Interoperable Scalable Architecture - Çoklu Platform Birlikte Çalışabilir Ölçeklenebilir Mimari) [12] modeline benzer yapıların kullanılması faydalı olacaktır. Blokzincir sistemlerinde mahremiyetin sağlanması, kuantum sonrası kriptografi (post-quantum cryptography) kullanan blokzincir sistemleri [35] de üzerinde çalışma yapılması gereken alanlardandır.

Teşekkür

Yorumlarıyla bölümün şekillendirilmesine yardımcı olan Dr. Senem Yazıcı Yılmaz'a teşekkürler. MSKÜ Blokzincir Araştırma Grubu'ndan Cemal Dak'a çizimlerdeki katkıları, Ayça Öksüztepe'ye yazım kontrolündeki katkıları; Emre Ertürk, Murat Doğan ve Ümit Kadiroğlu'na ekran çıktılarındaki katkıları için teşekkür ederiz.

Kaynaklar

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," Whitepaper. 2008.

[24] Inside China's Effort to Create a Blockchain It Can Control,
https://www.coindesk.com/china-to-create-it-can-control

[2] QYResearch, “Smart Contracts Market”, Valuates Reports, 2020, [Online]. Available: https://reports.valuates.com/market-reports/QYRE-Auto-31L1599/global-smart-contracts

[3] N. Atzei, M. Bartoletti and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," In: International conference on principles of security and trust. Springer, Berlin, Heidelberg, 2017. p. 164-186.

[4] V. Dhillon, D. Metcalf, M. Hooper, "The DAO hacked," In: Blockchain Enabled Applications. Apress, Berkeley, CA, 2017. p. 67-78.

[5] E. Işık, M. Birim and E. Karaarslan, “Tubu-io Decentralized Application Development & Test Workbench,” *arXiv preprint arXiv:2103.11187*, 2021.

[6] M. Birim, H. E. Ari and E. Karaarslan, “GoHammer Blockchain Performance Test Tool,” *Journal of Emerging Computer Technologies (JECT)*, 2021, 1.2: pp. 31-33

[7] E. Karaarslan and E. Konacaklı, “Decentralized solutions for data collection and privacy in healthcare”. In: *Artificial Intelligence for Data-Driven Medical Diagnosis*. De Gruyter, 2021. p. 167-190.

[8] A. M. Antonopoulos and G. Wood, "Mastering ethereum: building smart contracts and dapps," O'reilly Media, 2018.

[9] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," arXiv preprint arXiv:1707.01873, 2017.

[10] F. Vogelsteller and V. Buterin, "Ethereum whitepaper," Ethereum Foundation, 2014.

[11] K. Wüst and A. Gervais, "Do you need a blockchain?". Presented at 2018 Crypto Valley Conference on Blockchain Technology (CVCBT). IEEE, 2018. p. 45-54.

[12] E. Karaarslan and E. Konacaklı, “Data Storage in the Decentralized World: Blockchain and Derivatives” in “ Who Run The World:DATA”, 1st ed. Istanbul, Turkey, Istanbul University Press, 2020, ch.3, pp. 37-69. [Online]. Available: https://iupress.istanbul.edu.tr/tr/book/who-runs-the-world-data/chapter/data-storage-in-the-decentralized-world-blockchain-and-derivatives

[13] Z. Durğay and E. Karaarslan, “Blokzinciri Teknolojisinin E-Devlet Uygulamalarında Kullanımı: Ön İnceleme,” *Akademik Bilişim Konferansı, Karabük*, 2018.

[14] T. Jensen, J. Hedman and S. Henningsson, "How TradeLens Delivers Business Value With Blockchain Technology," MIS Quarterly Executive, 2019, 18.4.

[15] O. Goldreich and Y. Oren, “Definitions and properties of zero-knowledge proof systems,” *Journal of Cryptology*, 1994, 7.1: 1-32.

[16] C. Li, P. Li, D. Zhou, Z. Yang, M. Wu, G. Yang, ... and A .C. C. Yao, “A decentralized blockchain with high throughput and fast confirmation,” Presented at *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 2020. pp. 515-528.

[17] A. Zmudzinski, " Hyperledger Fabric Sees More Dev Activity Than Corda in Q3 2019: Report," CoinTelegraph, 2020.

[18] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, 1997.

[19] K. Chatterjee, A. K. Goharshady and A. Pourdamghani "Probabilistic smart contracts: Secure randomness on the blockchain," Presented at 2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). IEEE, 2019. p. 403-412.

[20] L. Marchesi, M. Marchesi, L. Pompianu and R. Tonelli, “Security checklists for ethereum smart contract development: patterns and best practices,” arXiv preprint arXiv:2008.04761, 2020.

[21] Z. Wang, H. Jin, W. Dai, K. K. R. Choo, and D. Zou, “Ethereum smart contract security research: survey and future research opportunities,” *Frontiers of Computer Science*, 2021, 15.2: pp. 1-18.

[22] S. Sayeed, H. Marco-Gisbert and T. Caira, “Smart contract: Attacks and protections,” IEEE Access, 2020, 8: pp. 24416-24427.

[23] B. Prasad, ” Vulnerabilities and Attacks on Smart Contracts over BlockChain,” Turkish Journal of Computer and Mathematics Education (TURCOMAT), 2021, 12.11: pp 5436-5449.

[24] E. Karaarslan, M. Birim and H. E. Ari, “Forming a Decentralized Research Network: DS4H”, to be published.

[25] C. Wickboldt, “*Benchmarking a blockchain-based certification storage system*,” (No. 2019/5). Diskussionsbeiträge, 2019.

[26] M. Kuzlu, M. Pipattanasomporn, L. Gurses, and S. Rahman, “Performance analysis of a hyperledger fabric blockchain framework: throughput, latency and scalability,” Presented at *2019 IEEE international conference on blockchain (Blockchain)*. IEEE, 2019. po. 536-540.

[27] S. Pongnumkul, C. Siripanpornchana and S. Thajchayapong, “Performance analysis of private blockchain platforms in varying workloads,” Presented at *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2017. pp. 1-6.

[28] Q. Nasir, I. A. Qasse, M. Abu Talib and A. B. Nassif, “Performance analysis of hyperledger fabric platforms,” *Security and Communication Networks, vol. 2018, Article ID 3976093, 14 pages, 2018.* https://doi.org/10.1155/2018/3976093

[29] J. Feist, G. Grieco and A. Groce, “Slither: a static analysis framework for smart contracts,” Presented at *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019. pp. 8-15.

--

[30] G. Grieco, W. Song, A. Cygan, J. Feist and A. Groce, “Echidna: effective, usable, and fastW. fuzzing for smart contracts,” In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020. pp. 557-560.

[31] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, ... and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” Presented at *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019. pp. 1186-1189.

[32] T. Durieux, J. F. Ferreira, R. Abreu and P. Cruz, “Empirical review of automated analysis tools on 47,587 ethereum smart contracts,” In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020. pp. 530-541, https://doi.org/10.1145/3377811.3380364.

[33] "Kişisel Verileri Koruma Kanunu, Kanun numarası: 6698", *Resmi Gazete Sayı*, pp. 29677, 2016.

[34] “GDPR”, Official Journal of the European Union, vol. L119, pp. 1-88, April 2016.

[35] T. M. Fernández-Carames and P. Fraga-Lamas, "Towards post-quantum blockchain: A review on blockchain cryptography resistant to quantum computing attacks," IEEE access, 2020, 8: 21091-21116.

Yazarlar

Dr. Enis Karaarslan

MSKÜ Bilgisayar Mühendisliği bölümünde Siber Güvenlik Abd. Başkanıdır. MSKÜ Yapay Zeka disiplininin kurucularındandır ve öğretim üyesidir. İletişim ağları, güvenlik eğitimi ve araştırması için NetSecLab'ı kurdu. MSKÜ blokzincir araştırma grubunda (MSKÜ BcRG) 2017'den beri blokzincirinin potansiyellerini incelemektedir. DS4H (decentralized solutions for humanity) blokzincir araştırma ağının kurucularındandır. Araştırma alanları bilgisayar ağları, siber güvenlik, blokzinciri, veri bilimi, afet yönetimi ve dijital ikizdir. Akıllı sözleşmelerle ve sıfır bilgi kanıt protokolleri ile güçlendirilmiş mahremiyet, merkezi olmayan kimlik ve blokzincir teknolojisinin etkin kullanımı üzerine patent başvuruları, danışmanlık ve yayınları bulunmaktadır.

Linkedin: https://www.linkedin.com/in/enis-karaarslan-1b195617/

Google Scholar: https://scholar.google.com/citations?hl=tr&user=D3dqZ5UAAAAJ

Melih Birim

Melih Birim 2006 yılında Marmara Üniversitesi Bilgisayar Mühendisliği Bölümünden mezun olmuştur. 2016 dan beridir blokzinciri konusunda TUBU ARGE firmasında kurucu ortak olarak araştırmalar yapmaktadır. Ayrıca Consensys GoQuorum sisteminde gönüllü elçi olarak seminerler ve eğitimler düzenlemektedir. Telekom operatörlerinin ve EPİAŞ yeşil enerji sertifika sisteminin blokzinciri mimarı olarak Türkiye'deki ender projelere imza atmıştır. tubu.io, gohammer gibi açık kaynak kodlu blokzincir yazılım araçlarının geliştirilmesinde katkıda bulunmuştur.

Linkedin: https://www.linkedin.com/in/melihbirim/

Google Scholar: https://scholar.google.com/citations?hl=tr&user=0so1uMAAAAAJ