Kitabın bu sürümü baskıdan önceki halidir. Kitabın açık erişimle paylaşılan son haline ulaşmak için:

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 otonom kod (akıllı sözleşme) tabanlı merkezi olmayan uygulamalar (decentralized applications) birçok alanda iş yapma şeklimizi değiştirmektedir. Otonom kodlar birçok sistem için devrimsel değişiklikler vaat etmekle birlikte, bu kodların yazıldığı dil ve ortamlar henüz yeterince olgun değildir. Değiştirilmez kayıtların kullanıldığı blokzincir sistemlerinde yazılım geliştirme süreçlerinde daha dikkatli olunması gereklidir. Özellikle değer transferlerinde sorun yaşanmaması için güvenli ve güvenilir uygulamalara ihtiyacımız bulunmaktadır. Bu kodların nasıl yazılması ve test edilmesi gerektiğine dair günümüzde yeterli kaynak bulunmamaktadır. Bu bölümde, bu konuda temel bilgiler verilmektedir. Bölüm blokzincir çözümlerinin nasıl geliştirileceği konusunda bir ön bilgi ile başlamaktadır. Ethereum, Quorum, Hyperledger Fabric, Corda gibi blokzincir platformlarının kıyaslaması verilecektir. Kurmakta olduğumuz DS4H blokzincir araştırma ağından söz edilecektir. Güvenli ve güvenilir uygulama geliştirmeye dair bölümdeki uygulama örnekleri Ethereum ortamında verilecektir. akıllı sözleşme geliştirilmesi, akıllı sözleşmelerin blokzincir sistemine yüklenmesi ve test edilmesi konusunda yapmakta olduğumuz çalışmalardan ve geliştirmekte olduğumuz araçlardan (Tubu-io, GoHammer) söz edilecektir. Bu tür sistemlere olan saldırılar ışığında; akıllı sözleşmeler için güvenlik denetim listesi ve öneriler sunulacaktır. Güvenlik testleri için kullanılabilecek yazılımlar tanıtılacaktır. Blokzincir tabanlı sistemlerin yazılım geliştirme süreçlerinde yaşanan ve aşılması gereken sorunlara da değinilecektir. Merkezi olmayan sistemlerin geleceğine dair saptamalarda bulunulacaktır.

1.1 Giriş

Merkezi bir otoritenin/sunucunun kontrol etmediği ve aracıların (intermediary) olmadığı çözümlere merkezi olmayan (decentralized) çözümler denir. Blokzincir tabanlı merkezi olmayan sistemler, aracılar olmadan taraflar arasında aracılar olmadan güvenilir bir şekilde işlemlerin yapılmasını sağlamaktadır. Bunu da kullandıkları konsensüs protokolleri, şifreleme algoritmaları ve akıllı sözleşmeler (smart contract) ile sağlamaktadırlar. Blok zincir yapısı, Bitcoin ile ilk olarak kripto para transferinde kullanılmış

ve 2009'dan beri çalışabilirliğini kanıtlamıştır. Merkezi olmayan çözümler sürekli gelişmektedir. Etherum ile akıllı sözleşme (smart contract) kavramı, yani bir adreste tanımlı otonom kodlar blokzincir sistemlerinde kullanılabilir olmuştur. Ethereum'dan sonra Corda, Hyperledger gibi farklı blokzincir platformları da bu tür otonom kod geliştirme ortamları sunmuşlardır [1].

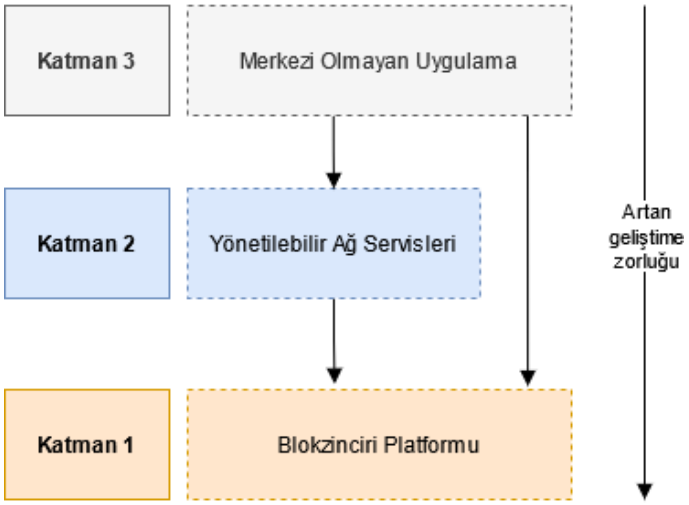
İnsan ürünü hiç bir sistemin yüzde yüz güvenli olamayacağını tekrar hatırlayalım. Blokzinciri ağını oluşturan blokzincir platformları, konsensüs protokolleri ve akıllı sözleşmelerin test edilmesi gereklidir. Bu kapsamda oluşturduğumuz DS4H blokzincir araştırma ağına dair makale [2] sürecimiz devam etmektedir. Yazılımcıların blokzincir sistemleri üzerinde uygulama geliştirme ve test süreçlerinde kolay kullanabilecekleri araçlara ihtiyacımız bulunmaktadır. Geliştirdiğimiz Tubu-io ve GoHammer platformlarına dair ön çalışmalarımızı yayınlarımızda [3, 4] daha önce sunmuştuk. Bu çalışmalar bu bölümde birlikte ele alınarak güvenli ve güvenilir kod geliştirme kapsamında bir temel oluşturulmakta kullanılacaktır. Akıllı sözleşmeleri geliştirirken dikkat edilmesi gerekenlerin temeli de bu bölümde ele alınacaktır. Bu kapsamlı konulara dair bir fikir vermeyi hedefliyoruz.

Bir sonraki bölümde merkezi olmayan çözüm geliştirme konusu ele alınacaktır. Üçüncü bölümde akıllı sözleşmelerde güvenli kod geliştirme Ethereum Solidity üzerinden örneklerle verilecektir. Blokzincir testleri dördüncü bölümde ele alınacaktır. Beşinci bölümde güvenlik kontrol listesi verilecektir. Blokzincir testleri altıncı bölümde tartışılacaktır. Yedinci bölümde blokzinciri için yazılım geliştirmede sıkıntılar ve fırsatlar ele alınacaktır. Bölüm sonuç ve değerlendirmeler ile bitecektir.

1.2 Merkezi olmayan çözüm geliştirme

Şekil 1.1'de de gösterildiği üzere; merkezi olmayan çözümleri üç ana katman olarak ele almak mümkündür:

- Katman 1: Blokzincir platformu (blockchain framework) en temel katmandır. Blokzincir tabanlı çözümlerin geliştirilebileceği ortamlar ve zincir yapıları sunarlar. Günümüzdeki belli başlı blokzincir platformları olarak Ethereum, Quorum, Hyperledger Fabric ve Corda'dan söz edilebilir.
- Katman 2: Yönetilebilir Ağ Servisleri (Managed network services) blokzincir platformunda yazılım geliştirme ve ölçeklendirme (scale) süreçlerini kolaylaştıran ağ servisleridir. Örnek olarak; Simba (https://simbachain.com/), Chainstack (https://chainstack.com/) ve Tubu-io (https://www.tubu.io) verilebilir.
- Katman 3: Merkezi olmayan uygulamalar (Decentralized application - Dapp), son kullanıcının bir mobil uygulama veya web tarayıcısı üzerinden blokzinciri ile iletişimini sağlayan ortamlardır. Merkezi olmayan uygulamalar, çoğunlukla bir API (Application Programming Interface - Uygulama Programlama Arayüzü) üzerinden blokzinciri ile iletişim kurarlar. Blokzincirindeki akıllı sözleşme olarak bilinen otonom kodu çalıştırırlar. Merkezi olmayan uygulamalar, üzerinde çalıştıkları blokzincir platformuna göre farklılıklar içermektedir. Birçok geliştirici bu katmanda yazılım geliştirir.

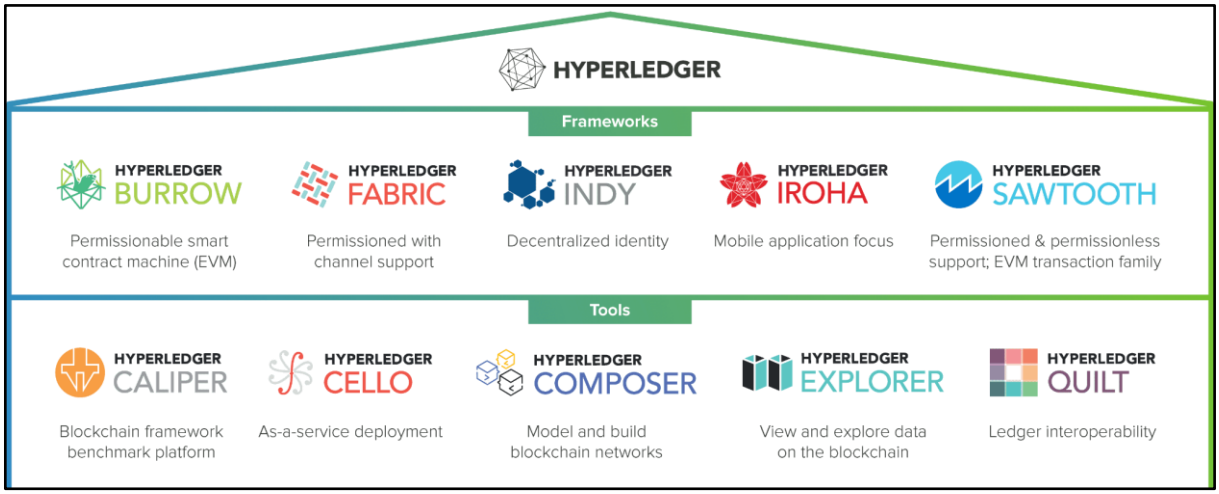


Şekil 1.1 Merkezi Olmayan Çözümler

Merkezi olmayan çözüm geliştirmede en zoru sıfırdan yeni bir blokzincir platformu geliştirmektir. Blokzincir ağı, blokzinciri çalıştıran birden fazla düğümden (node) oluşur. Kullanılan konsensüs protokolüne göre (örn. Raft) en az üç düğümünün kullanılması gerekir. Blokzincir ağı olarak var olan platformları kullanmaktan, kendi sistemlerimizi geliştirmeye; geliştirme sürecini kolaydan zora aşağıdaki gibi sıralamak mümkündür:

- Var olan bir platformu bir bulut servisi olarak alıp kullanmak,
- Var olan bir platformu kendi altyapınıza kurup kullanmak,
- Var olan bir platformu klonlayıp yeni bir sistem oluşturup geliştirmek,
- Sıfırdan yeni bir platform geliştirmektir.

Blokzincir platformları farklı amaçlara yoğunlaşmıştır. 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 1.2'de gösterilmiştir. Prof. Dr. Emin Gün Sirer'in ekibinin geliştirmekte olduğu Avalanche platformu[1] da gelişmekte olan[5] ve geliştiricilerine çeşitli fırsatlar sunan bir başka çözümdür.

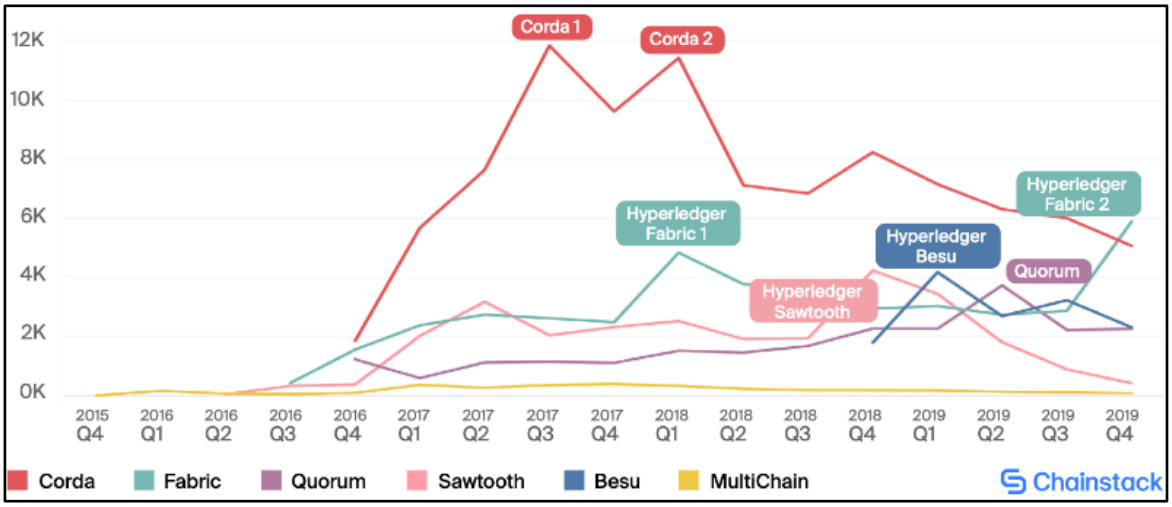


Şekil 1. 2 Belli başlı Hyperledger çözümleri[2]

[1] Avalanche platformu, https://www.avalabs.org/

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

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 arasındaki popülerlik değişimi Şekil 1.3'de gösterilmiştir. Bu zaman diliminde Corda'nın bir dönem ne kadar popüler olduğu ve sonra popülerliğini kaybettiği gözükmektedir. Belirtilen zaman diliminde Hyperdleger Fabric'in popülerliği artmıştır. Rapora[21] göre; bu değişim Fabric'in 2019 Kasımında kod yönetim aracı olarak Github'a geçmesinden sonra yaşanmıştır.



Şekil 1.3 2017-2019 arasında Hyperledger - Quorum - Corda Platform Github faaliyeti[21]

Blokzincir platformu seçerken geliştirici topluluklarının etkinliğine, topluluğun geliştirici sorunlarına cevap verme hızlarına ve projenin sürdürülebilirliğine dikkat edilmesi gereklidir. Günümüzde Ethereum ve kurumsal çözümde Quorum bu konuda sürdürülebilir projeler olarak gözükmektedir. Bazı Hyperledger projelerinde de benzer durumu görmek mümkündür. Bölümün hazırlandığı dönemde bu platformda Hyperledger Fabric ve Hyperledger Besu aktif projelerdendi.

Var olan bir blokzincir platformu üzerinde merkezi olmayan uygulama geliştirmek ise nispeten daha kolay bir süreçtir. Ethereum ve türevlerinde otonom kodlara akıllı sözleşme/anlaşma (smart contact) denmektedir. Hyperledger dünyasında bu tür kodlar zincir kodu (chain code) dense de, literatürde akıllı sözleşme kavramı daha sıklıkla kullanılmaktadır ve bu bölümde de bu şekilde kullanılacaktır. Ethereum'da bu kodlar Solidity dilinde yazılır. Ethereum, şu ana kadar en fazla otonom kod geliştirilen ortamdır. Yönetilebilir ağ servisleri kullanarak merkezi uygulama geliştirme süreci daha da kolaylaştırılabilir.

Bu bölümde Quorum/Ethereum uygulamaları ele alınacaktır. Consensys Quorum Etherum'un kurumsal çözümler, özellikle finansal çözümler için geliştirilmiş bir türevidir. Alt başlıklarda öncelikle ağ kullanımı ele alınacaktır. Sonrasında yönetilebilir ağ servislerden söz edilecektir. En son olarak da güncel geliştirme araçlarına örnekler verilecektir.

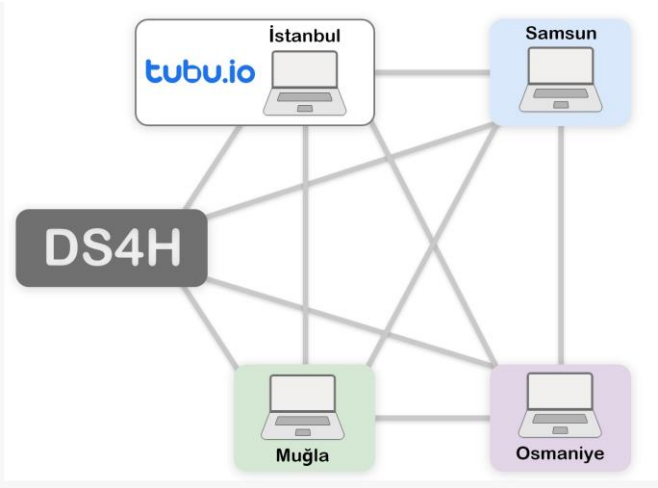
1.2.1 Blokzincir Ağ Kullanımı

Yazılımları gerçek blokzincir ağında denemeden önce, var olan deneme ağları (tesnet) üzerinde denenmesi gerekecektir. Her blokzincir ağı için çeşitli bulut tabanlı deneme ağları (testnet) bulunmaktadır. Ethereum test ağları dışındaki diğer ağlar, özel bir öğrenme eğrisi ve kullanım ücretleri gerektirmektedir.

Blokzincir test ağları, halka açık "public" defterleri (ledger) kullanır ve bu, gizlilik nedenleriyle birçok proje uygulamasında da tercih edilmez. Geliştirici sürecin gizli kalmasını istediğinde kendisine özel bir ağ kurması da gerekecektir. Günümüzde bu tür ağlar docker gibi konteyner (container) teknolojileriyle çok

kısa bir sürede ayağa kaldırılabilmektedir. Geliştirici kendi makinesinde veya bulut ortamında belirli sayıda düğümü çalıştırarak test işlemlerini gerçekleştirebilir.

Test ağları, ölçeklenebilirlik ve performans açısından esnek değildir ve performans testlerinde gerçekçi sonuçlar vermezler. Gerçekçi bir test ortamı oluşturmak için fiziksel bir test ağına ihtiyaç bulunmaktadır. Merkezi olmayan teknolojilerin araştırılıp geliştirilebileceği sürdürülebilir ve ölçeklenebilir bir araştırma ortamı olarak "İnsanlık için Merkezi Olmayan Çözümler" (Decentralized solutions for humanity -DS4H) kurulmaktadır. Internet Society'den (ISOC) alınan hibe desteği ile DS4H'ın ilk deneme düğümleri (node) Türkiye'nin dört farklı bölgesinde konumlandırılmıştır. Şekil 1.4'de ağın kurulan ilk düğümleri gösterilmiştir. Ağın testleri ve yayın süreçleri devam etmektedir [3]. Ağın otonom kodlarla yönetimi için çalışmalar devam etmektedir. Projenin ilerleyen etaplarında, araştırma kurumlarının ağa dahil olarak araştırmacılarını bu ağdan yararlandırması mümkün olacaktır. Süreci takip etmek ve ağdan yararlanmak için DS4H proje sayfasını[3] inceleyebilirsiniz.



Şekil 1.4 DS4H Ağ Yapısı[3]

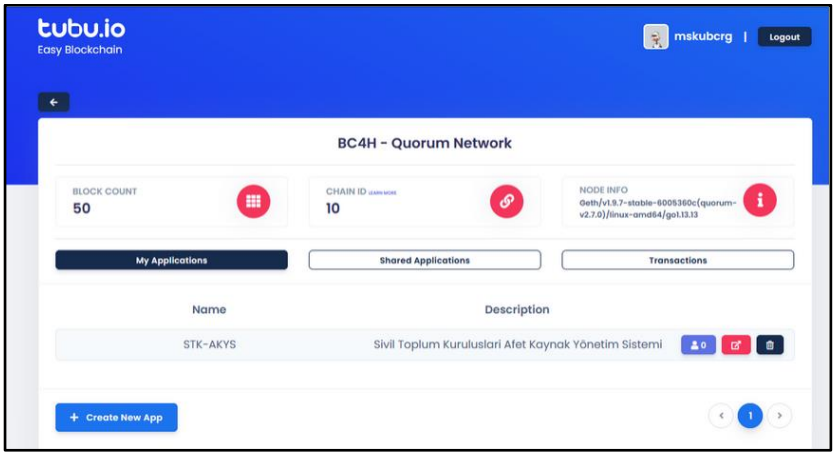
1.2.2 Yönetilebilir Ağ Servisleri

Geliştiricilerin canlı ortam testlerini bu ortamlarda gerçekleştirilmesi kolay süreçler değildir. Blokzincir teknolojileri her geçen gün gelişmektedir ve tüm gelişmeleri yakalamak ciddi zaman ve efor gerektirmektedir. Bu süreçleri kolaylaştıracak yazılımları yönetilebilir ağ servisleri (managed network services) olarak tanımlayabiliriz. Bu servislerin sağlayabilecekleri olarak şunlardan söz edebiliriz:

- State (durum) takibi: Kod çalıştığında sistemde oluşturacağı değişikliğin takibi,
- Sürüm (Versiyon) Takibi: Geliştirilen kodun değişik versiyonlarının takibi,
- Konsensüs protokolü seçimi: Farklı konsensüs protokolleri seçiminde performansın değişiminin gözlenmesidir.

Geliştiricilerin kodlarının sonuçlarını görebileceği, kullanımı kolay bir arayüze ihtiyaç bulunmaktadır. Geliştiricilerin akıllı sözleşmeleri blokzincir ağlarına yüklemeleri ve bu sözleşmelerle kolayca etkileşim kurmaları için Tubu-io blokzincir uygulama ortamı (workbench) (https://www.tubu.io) geliştirilmiştir. Bu ortamın merkezi olmayan uygulama projeleri geliştirmenin geliştirme süresini ve maliyetlerini azaltmada da etkisi olacaktır. Detaylı bilgi için bkz [3]. Geliştiriciler sistemde açacakları hesaplar üzerinden blokzincir ağları ile etkileşimlerini web arayüzünden gerçekleştirebileceklerdir. Tubu-io web arayüzü Şekil 1.5'de gösterilmiştir. Tubu-io, bu konulara yeni başlayacak geliştiriciler için merkezi olmayan uygulama programlamayı öğretmek için de kullanılabilir.

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



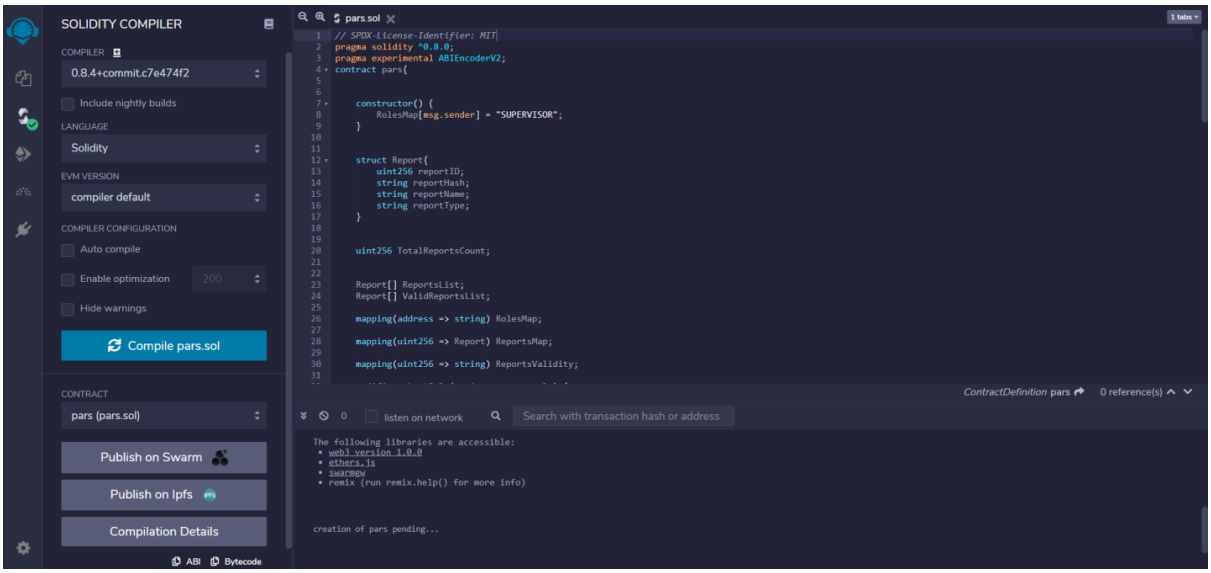
Şekil 1.5 Tubu-io Web Arayüzü [3]

1.3 Geliştirme Araçları

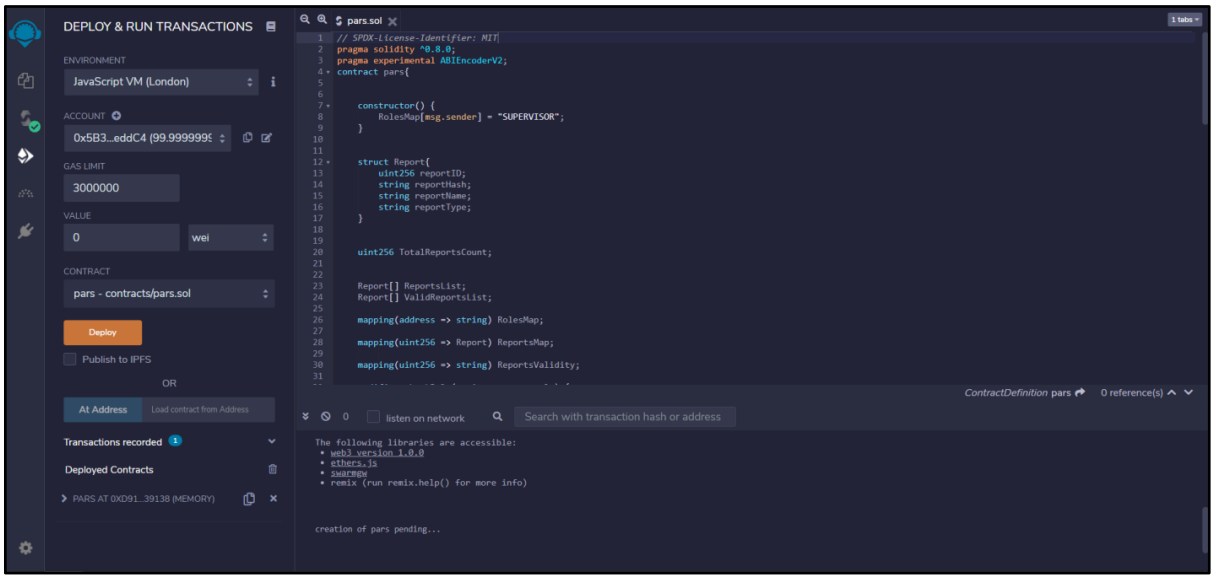
Ethereum ortamında geliştirme sürecinde kullanılan belli başlı araçlar olarak Geth, Clef, Remix, Metamask, Truffle, Ganache ve web3.js örnek verilebilir. Bu araçlar özetle aşağıda açıklanmıştır:

- Geth, Ethereum 'un Golang dilinde geliştirilmiş bir istemcisidir. Geth benzeri birden fazla ethereum istemci (client) seçeneği bulunmaktadır, bunlardan başlıcaları olarak parity-1 ve besu-2 dan söz edilebilir.
- Clef, Ethereum da hesap (account) yönetimi için kullanılan yeni bir sistemdir. Tek başına bir işlem olarak çalışır ve tüm hesap yönetimlerini Ethereum geth istemcisi yerine kendi başına sağlar.
- Remix IDE, açık kaynaklı bir web ve masaüstü uygulamasıdır. Zengin bir eklenti setine sahiptir. Remix, tüm sözleşme geliştirme sürecinde kullanılabileceği gibi, Ethereum'u öğrenmek ve öğretmek için de kullanılabilir. Remix geliştirme ortamı Şekil 1.6'da gösterilmiştir. Bu arayüz üzerinden derleme (compile) ve yazılımı dağıtma (deploy) süreçleri gerçekleştirilebilmektedir.
- Truffle, Ethereum Sanal Makinesi (EVM) kullanarak blokzincirinde sürdürülebilir ve profesyonel uygulamalar geliştirmek için akıllı sözleşmelere yönelik bir dizi programlama aracıdır ve ayrıca geliştirici dostu entegre bir geliştirme ortamında çeşitli testler gerçekleştirir[4]. Ortam arayüzü Şekil 1.7'de gösterilmiştir.
- Truffle ile birlikte gelen Ganache ile kendi makinenize bir Ethereum blokzinciri ağ simülatörü kurabilir ve Solidity sözleşmelerinizi test edebilirsiniz. Genanche arayüzü Şekil 1.8'de gösterilmiştir.
- Metamask, blokzincir uygulamalarına bir kripto cüzdanı ve ağ geçididir. Uygulamalar geliştirilirken Truffle ile birlikte kullanılır. Arayüzü Şekil 1.9'da gösterilmiştir.
- web3. js, HTTP veya IPC bağlantısı üzerinden Ethereum ağı ile etkileşim kurulmasına izin veren bir kitaplık koleksiyonudur.

[4] Welcome to Remix's documentation!, https://remix-ide.readthedocs.io/en/latest/



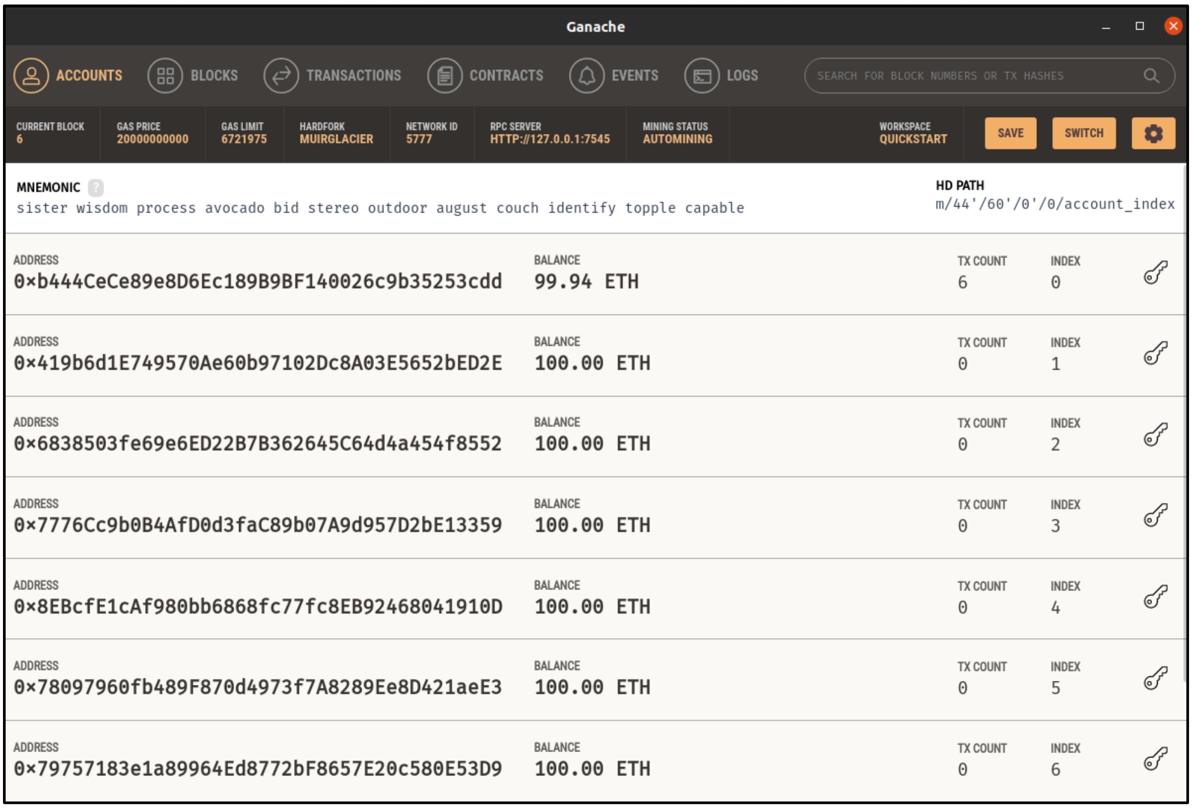
Şekil 1.6a Remix Geliştirme Ortamı - Derleme (compile)



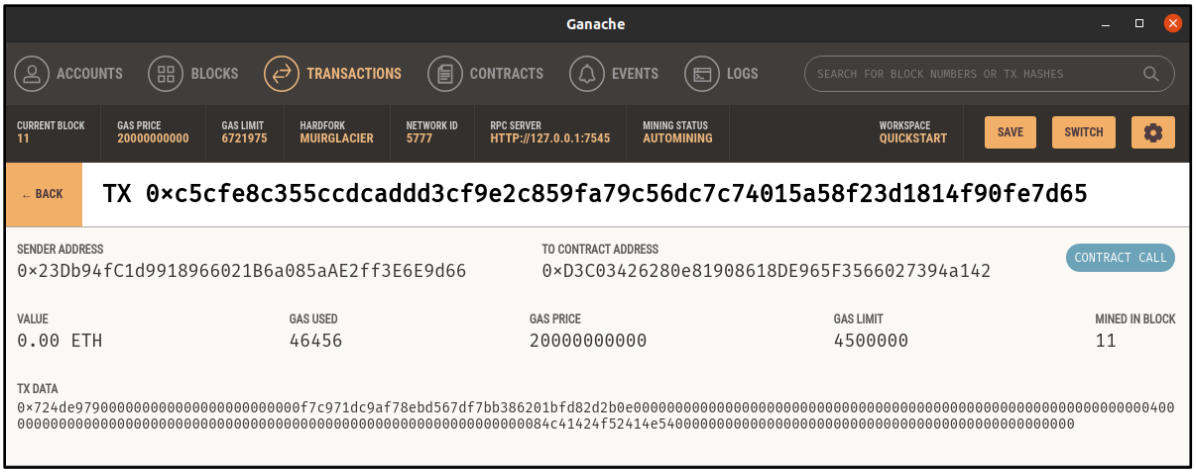
Şekil 1.6b Remix Geliştirme Ortamı - Yazılım Dağıtma (Deploy)

```
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: '0x000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    rawLogs: []
  },
  logs: []
}
truffle(development)> contract.getUserRole(accounts[3])
'LABORANT'
truffle(development)>
```

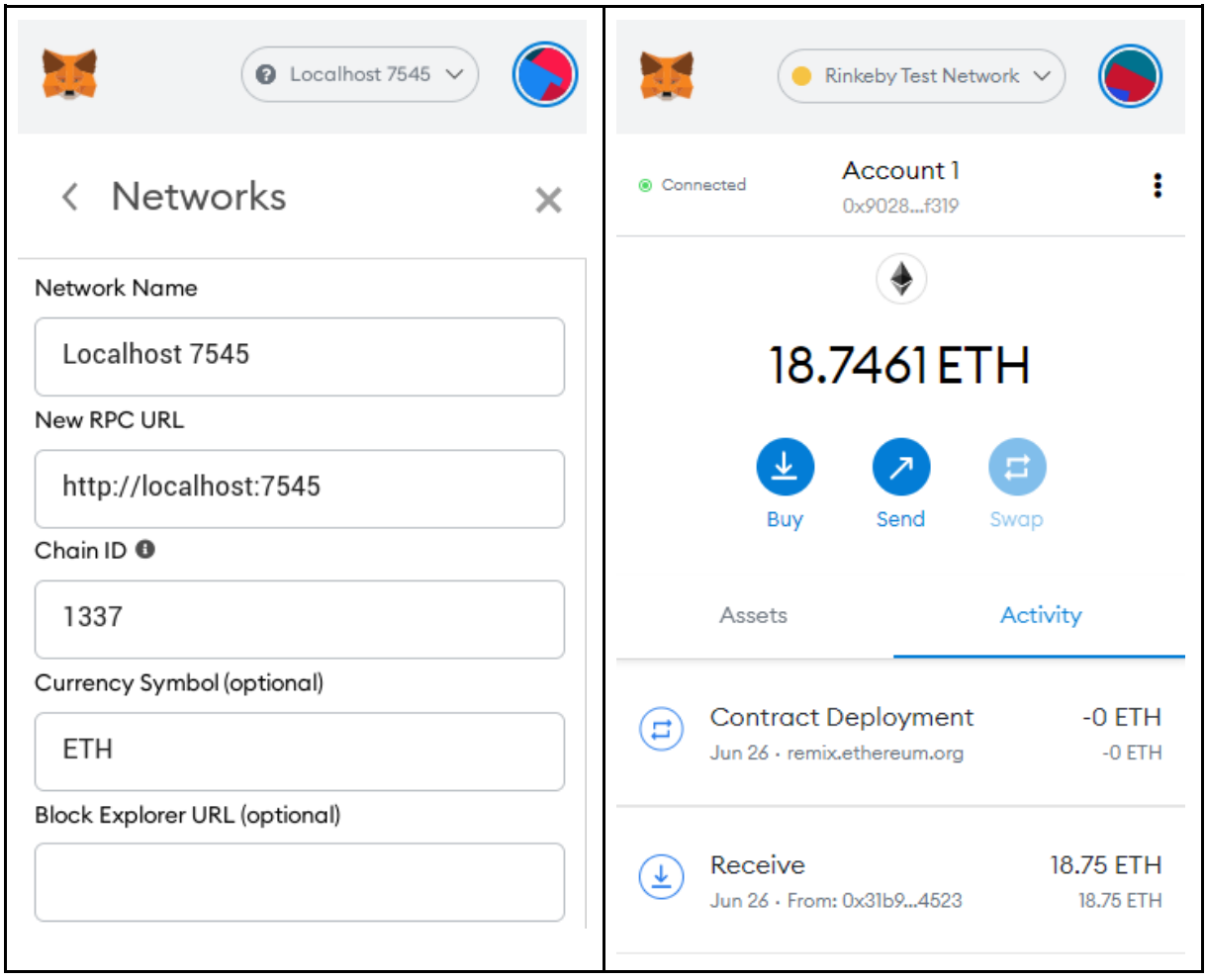
Şekil 1.7 Truffle Arayüzü



Şekil 1.8a Ganache Arayüzünce işlemlerin takibi



Şekil 1.8b Ganache ile bir işlemin detaylı gösterimi



Şekil 1.9 (a) Metamask Network Ayarları (b) Metamask ile Rinkeby Test Networküne bağlı bir hesabın takibi

1.4 Akıllı Sözleşmelerde Güvenli Kod Geliştirme

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

Akıllı sözleşmelerin üzerinde çalışacağı blokzinciri ve özellikle Ethereum çok yeni bir teknoloji ve plaformlardır. Bu teknoloji her geçen gün iyileşmeye ve gelişmeye devam ediyor. Alışılagelmiş programlama dilleri gibi kaynak kodlarının yıllarca incelenmiş ve geliştirmiş olmaması; merkezi olmayan sistemler geliştirirken hatalara ve diğer olası güvenlik açıklarına hazır olmamızı gerektirmektedir. Bu bölümde sunulan güvenlik uygulamalarını takip etmek, bu nedenle bir akıllı sözleşme geliştiricisi olarak yapmanız gereken güvenlik çalışmasının yalnızca başlangıcıdır. Ethereum tabanlı akıllı sözleşme geliştirirken aşağıdaki altı temel kuralı unutmamamız gerekir:

1. Hataya hazırlan: Hatalar her zaman olur, önemli olan hataya önceden hazırlanmaktır. Bu nedenle kodunuz hataya zarif bir şekilde cevap vermelidir. İstisnalar (exception), hata yakalama blokları kullanılmalıdır.
2. Dikkatli kod yaz: Dikkat, kod yazarken en önemli unsurdur. Solidity dilinde, hatalar önceden yakalanmalı desek de dikkatli kod yazma stili geliştirmeye önem verilmelidir.
3. Basit (sade) tasarım oluştur: Akıllı sözleşmeler basit (simple) tutulmalıdır. akıllı sözleşmeler, gelişmiş yazılım dilleri ve sistemleri gibi gelişmiş yapıları olmadıkları için, yazılan sistemi basit tutmak oldukça önemlidir.
4. Her zaman en son çıkan versiyonları kullan: Temel programlama mantığına aykırı olsa da, gelişmekte olan bir sistem olan Ethereum tabanlı akıllı sözleşmelerde her zaman en son versiyonları ve gelişmeleri takip etmek, güvenlik açıklarını daha önceden öğrenmek ve ona göre önlem almak açısından önemlidir.
5. Blokzincir teknolojisini öğren: Blokzincir temellerini ve detaylarını öğrenmeden, üzerinde geliştirme yapacağınız sistemi anlamanız ve ona göre kod yazmanız pek olası değildir. Ethereum 2.0 ve yan zincir gibi yapıları öğrenmek gereklidir.
6. Kriptoloji temellerini asla unutma: Kriptoloji temellerini öğrenmek, anlamak ve sonra da unutmamak gerekiyor.

Alt bölümlerde aşağıdaki konular ele alınacaktır:

1. Dış çağrılarda Dikkat Edilmesi Gerekenler
2. Kod Optimizasyonu

1.4.1. Dış çağrılarda Dikkat Edilmesi Gerekenler

Solidity dili ve Ethereum içerisinde dış çağrılar yapılabilmektedir. Bu dış çağrılar (external call); genelde adresi bilinen başka bir akıllı sözleşme üzerindeki bir fonksiyonu çağırma şeklinde olmaktadır. Bunu bir örnekle inceleyelim. Eğer bunu Ethereum ağında uygulayacaksanız, bu şekilde kullanmamanız çok yerinde olacaktır. Dış çağrıların her zaman o sözleşmede veya bağlı olduğu herhangi bir başka sözleşmede kötü amaçlı bir kod olarak yürütülebileceğini unutmamanız gerekir.

```
interface IA {
    function setNumber(uint n) external;
    function getNumber() external view returns (uint);
}
```

Kontrat A

```
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();}
}
```
Kontrat B - Güvensiz kullanım örneği

Kontrat B'den Kontrat A'daki fonksiyonun çağrıldığı örnekte; IA bir interface olarak tanımlanmış ve B kontratı ile adresi verildikten sonra setNumber fonksiyonu içerisinden A kontrat'ının fonksiyonu çağrılmıştır. Burada ilk aşamada unutulmamalıdır ki A kontratı deploy edildiğinde, uygulama detayı bilinmediği için B kontratı içerisindeki fonksiyonun ne yapacağı bilinmemektedir.

Dış çağrılar ile iletişim halinde olduğunuzda, yazdığınız akıllı sözleşme içerisinde, diğer kontratı güvensiz olarak işaretlemek önemlidir. Dış çağrılar ile yapacağınız etkileşimin kod içerisindeki en son işleminiz olması, sizi olası hatalara karşı daha kapalı hale getirecektir. Kontrat Bv2 de 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 ve ya 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();}
}
```
Kontrat Bv2 - Güvenli kullanım örneği

Dış çağrılarda meydana gelebilecek olası hataların; kendi yazdığınız kontratlarda kontrol edilmesi gerekir. Dış çağrılar, eğer bir address üzerinde gidiyorsa, Solidity low-level çağrı özelliği ile call, callcode, deletecall, send fonksiyonları üzerinden kullanılmalıdır. Bu low-level fonksiyonlar eğer bir hata var ise "false" döndürler 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)(""); // oldukça tehlikeli, doğrudan bütün gas
diğer adrese gönderildi
```
Aşağıdaki şekilde yapın;

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

Address call yapmak yerine, kontrat import ile kullanmak daha doğru olacaktır.

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

1.4.2 Kod Optimizasyonu

Bu bölümde aşağıdaki konular ele alınacaktır:

- Fonksiyon ve durum değişkenlerinin görünürlüklerinin (function visibility) düzgün ayarlanması
- "Abstract" Kontratlar ve "Interface" Kontratlar
- Fallback fonksiyonlar
- Kontrat günlükleri için Olay (Event) tetiklemesi kullanmak
- Tamsayı bölme işlemi ve yuvarlama
- Değer transferinde assert(), require() 'ın doğru kullanımı
- modifier 'ların sadece kontrol amaçlı kullanımı

Fonksiyon ve durum değişkenlerinin görünürlüklerinin düzgün ayarlanması

Fonksiyon ve durum değişkenlerinin (function visibility) Solidity dilinde "external", "public", "internal" ve "private" olmak üzere dört farklı görünürlüğü vardır:

1. Public: Fonksiyonlara ve değişkenlere herhangi bir görünürlük etiketi tanımlanmamışsa "public" otomatik olarak atanır. Değişkenler için otomatik olarak "set" ve "get" fonksiyonları oluşturulur.
2. External: "External" fonksiyonların tanımlanması public tanımlanmasından daha doğru olabilir. Bu tip fonksiyonları "this" ile çağırmak mümkündür. Public olanlar ise doğrudan fonksiyon adı ile çağrılma bilmektedir.
3. Internal: Internal fonksiyonlar ve durum değişkenleri sadece içeride çağrılabilen fonksiyonlardır. Dışarıdan erişim mümkün değildir.
4. Private: Private fonksiyonlar ve durum değişkenleri ise sadece bulunduğu kontrata özel olmaktadır. Eğer bu kontrattan türetilen başka kontrat var ise private fonksiyonlara erişim mümkün değildir.

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

function auto_public() { // public

}
```

Yukarıdaki gibi fonksiyon tanımlarının yapılmaması gerekir. Fonksiyonlar ve durum değişkenlerinin kod içerisinde kullanımına göre ayarlanması en iyi yöntemdir. Saldırılara karşı yapılabilecekler:

- Fonksiyonların varsayılan görünürlüğü[5]
- Durum değişkeni varsayılan görünürlük[6]

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

```
uint private x;

function external_function() external {

    // this.external_function() ile çağrılır

}

function internal_function() internal {

    // doğrudan internal_funciton() olarak this kullanmadan çağrılır.

}
```

"Abstract" Kontratlar ve "Interface" Kontratlar

Her iki tip kontratlar da bir seviyede yeniden kullanım sağlasalar da, "interface" kontratların durum değişkenlerine erişemiyor olması, kod yazımı ve yönetimi bakımından önemlidir. "Abstract" kontrat ise fonksiyon tanımı ve uygulaması içerdiğinden dolayı bu tip kontratları kullanırken kod gözden geçirme stilleri tatbik edilmeli ve bilinmeyen kaynaklardan kontratlar kullanılmamalıdır.

Fallback fonksiyonlar

Bu fonksiyonlar her kontrat içerisinde sadece bir kere tanımlanabilen, ve kontrat içerisinde eğer çağrılan isimde bir kontrat yoksa çağrılan "şalter" (fallback) diye adlandırabileceğimiz fonksiyonlardır.

Bu fonksiyonlar ayrıca kontrat'a doğrudan Ether gönderildiğinde de çalışırlar. Bu fonksiyon, kontrat'a gönderilen tüm mesajlarda çalışır. Bu fonksiyona ether göndermek hata atar çünkü ödenebilir (payable) bir fonksiyon değildir.

```
contract Test {

    uint x;

    fallback() external { x = 1; }

}
```

Aşağıdaki şekilde 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 günlükleri için Olay (Event) tetiklemesi kullanmak

Olaylar kontrat içerisinde kullanılan ve işlemler (transaction) sırasında olan değişimleri haber verebilecek günlük yapılarıdır. Bu sayede, bir kontrata atılan isteklerin durumlarını o kontratın tüm transaction bilgilerini event içerisinden alabilirsiniz.

Tamsayı bölme işlemi ve yuvarlama

Solidity, tüm tamsayı bölmeleri 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 ileride “fixed” keyword’ü kullanılacaktır fakat şu anda bu keyword kullanılabiliyor olsa bile desteklenmemektedir. “fixed” in float ve double a farkı tutulacak küsüratların, determinizim gereği belli olması gereklidir. Bu tip bölmelerde, kullanılması gereken zincir dışında (off-chain üzerinde) işlemlerin yapılmasıdır.

```
uint carpan = 10;

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

Değer transferinde assert(), require() ‘ın doğru kullanımı

Değer transferleri (assert transfer) ciddiye alınmalıdır. Bu iş için kullanılan assert() ve require() fonksiyonları birbirlerine çok benzer işler yapıyor olsalar da opcode bakımından kullanım alanlarının farklı olduğu bilinmelidir. Her iki fonksiyon da kontrol için kullanılıp eğer verilen doğruluk sağlanmıyorsa hata fırlatacak yapıda tasarlanmış olsa da; 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 miktarını transfer öncesindeki miktar ile karşılaştırmaktadır. require() ise sadece değerlerin (girdi (input)

değişkenleri, dış 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" burada her zaman 0 dan büyük veya 0 a eşit olmalı olarak kontrol edilecektir. Eğer doğruluk sağlanmıyorsa hata, bir sonraki parametredeki mesaj ile atılır.

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

        require(_amount > 0,"Transfer amount should be bigger than 0");


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

             "Transfer amount is bigger than current account balance");

        // basic addition and subtraction is not secure!

        balanceOf [to] += _amount;

        balanceOf [msg.sender] -= _amount;


        // internal control for balances, reconciliation

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

    }
```

Yukarıda görüldüğü gibi, transfer fonksiyonu önce değer kontrollerini ve bakiye (balance) kontrollerini yapıyor. Bunu require() fonksiyonu ile yaptığımızda olası hataların önüne geçilmiş olacaktır. Sonrasında yapılacak işlemler, bir toplama çıkarma fonksiyonu olabileceği gibi, bir dış çağrı da olabilir. Burada olası iç hataların önüne geçilmesi için fonksiyon bitiminde "assert" ile tekrar doğrulama yapılarak işlemlerin sağlaması yapılmalıdır.

modifier 'ların sadece kontrol amaçlı kullanımı

Solidity "modifier" keyword'ü ile fonksiyonları, 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 değişkenleri değiştirilebildiği için dikkatli ve önce kullanımı önemlidir. Başka akıllı sözleşmelerin içerisindeki modifier fonksiyonlar iç aktarım (import) yapılmadan önce kod güvenliği açısında kaynak kodun gözden geçirilmesi yapılmalıdır. Modifier fonksiyonların bir diğer amacı da kod okunurluğunu arttırmaktır. Modifer kullanımı yerine require ve assert fonksiyonlarını kullanabileceğinizi de unutmayın.

```
// Amacımız transfer fnks. çağıran cüzdanın bakiye karsilastirmasini
yapmak

    modifier hasBalance(uint _amount) {

        require(_amount > 0,"Transfer amount should be bigger than 0");

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

               "Transfer amount is bigger than current account balance");

        _; }

       // hasBalance modifier fonksiyonunu
       // ihtiyaç olan her yerde kullanılabilecek

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

        // basic addition, not secure!

        balanceOf[to] += _amount;

        balanceOf[msg.sender] -= _amount;



        // internal control for balances, reconciliation

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

   }
```

1.5 Akıllı Sözleşmelere Saldırılar

Akıllı sözleşmeler, günümüzde finansal değer transferinde yaygın olarak kullanılmaktadır. Bu sözleşmelerdeki olası zafiyetlerin 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" güzel bir kaynaktır[7]. Bu konuda bilinen saldırılar Consensys'in Github sayfasındaki en iyi uygulamalar dökümanında[8] ele alınmıştır. Akıllı sözleşme bazlı saldırılar ve buna karşı yapılabilecek korumalar literatürde [6-9] de ele alınmıştır. Bu süreci detaylı işleyen ve önerilerde bulunan [8] de incelebilir. Akıllı sözleşmelerin güvenliği hakkında yapılan akademik çalışmalar ve olası araştırma konuları güncel bir yayında [9] incelenmiştir.

Akıllı sözleşmelere saldırılardan başlıcaları Tablo 1.1'de karşılaştırılmıştır. Ethereum ağına özel ve altta çalışan sistemle alakalı olan akıllı sözleşme saldırıları da Tablo 1.2'de verilmiştir. Yapılabilecek çözümlere dair bazı öneriler de tabloda verilmiştir. Çözüm süreçlerinde güvenli kodlama prensiplerinin uygulanması önceki bölümde ele alınmıştı.

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

Tablo 1.1 Akıllı Sözleşmelere olan temel saldırılar[6]

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 token'i 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[6]

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 Güvenlik Kontrol Listesi

Güncel bir çalışmada [8] de güvenlik riskleri ele alınmış, http://tiny.cc/security_checklist adresinden de ulaşılabilen bir güvenlik denetim listesi önerilmiştir. Bu denetim listesini aşağıdaki şekilde özetlemek mümkündür:

- Etki Etkileşimini Kontrol Etmek: Bir sözleşmede bir fonksiyonu gerçekleştirirken mutlaka şu sırayı takip edin:
    - Tüm ön koşulları kontrol edin
    - Etkileri sözleşmenin durumuna uygulayın
    - Diğer sözleşmelerle etkileşime geçin
- Proxy Temsilcisi kullanmak: Vekil kalıpları (Proxy patterns) , akıllı sözleşmelerin yükseltilmesini kolaylaştırmak için birlikte çalışan bir dizi akıllı sözleşmedir. Vekil, adresi değiştirilebilen bir başka akıllı sözleşmedir. Kullanımında bir yeni sözleşmeyi gösterebilecektir. Böylece blokzincir kaynaklarının idareli kullanılmasını sağlayarak GAS tasarrufu sağlar.
- Yetkilendirme yapmak: Kritik metodlar ancak belirli kullanıcılar tarafından yürütülmelidir. Bu adreslerin eşlenmesi kullanılarak gerçekleştirilebilir ve değiştiriciler (modifier) kullanılarak kontrol edilebilir
- Sahipliği tanımlamak: Sözleşme yönetiminden sorumlu olan ve özel izinlere sahip olan sözleşme sahibini belirtin. Bu da muhtemelen bir önceki maddede belirttiğimiz yetkilendirilen adres olacaktır. örn. kritik metotları çağırmaya yetkili tek adrestir.
- Kahin kullanmak: Kahin (oracle), blokzinciri dışından veri sağlayan bir akıllı sözleşmedir. Güvenilir bir kaynak tarafından beslenmektedir. Ters(reverse) kahin olarak kullanımında; belirli koşulları kontrol etmek için zincir dışı (off-chain) bileşenler tarafından okunacak verileri sağlayan bir akıllı sözleşme kullanılacaktır.
- Oran Sınırlandırmak (Rate limit): Bir akıllı sözleşmeye gönderilen mesaj sayısını ve dolayısıyla hesaplama yükünü sınırlamak için bir görevin belirli bir süre içinde ne sıklıkta yürütülebileceğini düzenleyin.
- Süreçleri Yavaşlatmak (Speed Bump): Hassas görevleri sözleşmelerde yavaşlatın. Bu sayede kötü niyetli eylemler gerçekleştiğinde hasar sınırlı olacaktır ve buna karşı önlem almak için daha fazla zamana sahip olunacaktır.
- Bakiye Limiti kullanımı: Bir akıllı sözleşme içinde tutulan maksimum fon (fund) miktarını sınırlayın.
- Koruma Kontrolü: Akıllı sözleşmenin durumu ve fonksiyon girdilerine dair tüm gereksinimlerin karşılandığından emin olun.
- Zaman Kısıtlaması kullanımı: Zaman kısıtlaması, eyleme ne zaman izin verileceğini belirtir. Bu da işlemi tutan bloktaki kayıtlı zamana dayanır. “ Speed Bump” ve “Rate Limit” yapılarında da kullanılabilir.
- Sonlandırma kullanımı: Bir akıllı sözleşmenin kullanım ömrü sona erdiğinde sonlandırılır. Bir sözleşmeyi feshetme yetkisine genelde sözleşme sahibi sahiptir. Sözleşmeye geçici (ad-hoc) kod ekleyerek veya “selfdestruct“ (kendi kendini yok etme) işlevini çağırarak yapılabilir.
- Mantık oluşturmak: Taşmalardan (overflow), yetersizliklerden (underflow) veya sonlu aritmetiğin diğer istenmeyen özelliklerinden korumak için bazı kritik işlemleri hesaplayan bir mantık oluşturulmalıdır.
- Mahremiyeti Sağlamak: Blokzincirinde kişisel veri tutulmasını tavsiye etmiyoruz. Sıfır bilgi kanıt (zero knowledge proof) [14] protokollerinin kullanımı yerinde olacaktır. Yine de tutulması gerektiğinde; zincirdeki bu tür verileri şifreleyin, gizliliği iyileştirin. Türkiye’de KVKK [15], Avrupa GDPR [16] gibi yasal gereksinimleri karşılayın.
- Yeniden Kullanılabilirlik Yapıları: Çoklu oluşumlar (multiple instance) için sözleşme kitaplıklarını ve şablonlarını kullanın.

- Mutex kullanmak: 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.

Bu süreçlerinde kullanılabilecek belli başlı güvenlik araçlarını şu şekilde listelemek mümkündür:

- Slither[9]: Yaygın güvenlik açıkları için 70'den fazla yerleşik (built-in) algılayıcıya sahiptir [17].
- Echidna[10]: Echidna [18], kodu sözde rasgele (pseudo random) olarak üretilen işlemlerle çalıştırır. Araç (fuzzer), belirli bir özelliği ihlal etmek için bir dizi işlem bulmaya çalışacaktır.
- Manticore[11]: Manticore [19], 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.

Ethereum'un resmi sayfasındaki kıyaslama bilgilerinden[12] yararlanılarak tablolar oluşturulmuştur. En yaygın kullanılan üç test aracının kıyaslanması Tablo 1.3'de verilmiştir. Bu araçların desteklediği testlere göre kıyaslanması Tablo 1.4'de verilmiştir. Detaylı inceleme için ilgili adrese bakılabilir.

Tablo 1.3 Üç Ana Akıllı Sözleşme Güvenlik Test Aracının Kıyaslanması

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

Tablo 1.4 Akıllı Sözleşme Güvenlik Test Aracı Desteklenen Test Çizelgesi

Testler	**Slither**	**Echidna**	**Manticore**
Durum makinesi		✓	✓
Erişim Denetimi	✓	✓	✓
Aritmetik İşlemler		✓	✓
Türeme doğruluğu (Inheritance correctness)	✓		
Dış etkileşimler		✓	✓
Standartlara uyma	✓	✓	✓

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

Durieux ve arkadaşların ilginç çalışmalarında [20], ilk olarak; dokuz otomatik analiz aracının geçerliliğini tespit etmek için güvenlik açığı olan 69 akıllı sözleşme üzerinde denemişlerdir. Sonrasında Etherscan'de bulunan 47,587 akıllı sözleşme üzerinde denetlenme gerçekleştirmişlerdir. Bu analizler 564 gün 3 saat sürmüştür. Bu deneylere göre; bu akıllı sözleşmelerinin %97 'inde 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'un resmi sayfasında bulunan güvenlik listesinde önerilen güvenlik önlemlerini şu şekilde özetlemek mümkündür[13]:

- Bilinen güvenlik sorunlarını kontrol edin: Slither ve Crytic gibi araçlarla sözleşmelerinizi sürekli olarak gözden geçirin.
- Sözleşmenizin özel özelliklerini göz önünde bulundurun:
    - Sözleşmelerinizin yükseltilebilir olup olmadığını "Slither-check-upgradeability" veya "Crytic" ile denetleyin.
    - Sözleşmeleriniz ERC'lere uygunluğunu "slither-check-erc" ile kontrol edin.
    - Truffle'da birim testlerini (güvenlik özellikleri paketini) "Slither-prop" ile otomatik olarak oluşturun.
    - Üçüncü taraf token'larıyla entegrasyon söz konusu ise; ilgili denetim listesine[14] göre davranmak yerinde olacaktır.
- Kodunuzun kritik güvenlik özelliklerini görsel olarak inceleyin:
    - Slither'ın "inheritance-graph" (miras grafik) oluşturucusunu kullanın. İstenmeyen gölgeleme (inadvertent shadowing) ve C3 doğrusallaştırma (linearization) sorunlarından kaçının.
    - Fonksiyon görünürlüğünü ve erişim kontrollerini raporlamak için Slither'ın işlev özeti (function-summary) oluşturucusunu kullanın.
    - Durum değişkenleri üzerindeki erişim kontrollerini raporlamak için Slither'ın değişken ve yetkilendirmeli (vars-and-auth) oluşturucusunu kullanın.
- Kritik güvenlik özelliklerini belgeleyin ve bunları değerlendirmek için otomatik test oluşturucuları kullanın:
    - Kodunuz için güvenlik özelliklerini belgelemeyi öğrenin.
    - Echidna ve Manticore araçlarında kullanım için Solidity'de güvenlik özelliklerini tanımlayın. Sonlu otomata (state machine), erişim kontrollerine, aritmetik işlemlere, harici etkileşimlere ve standartlara uygunluğa odaklanılmalıdır.
    - Slither'in Python API'si ile güvenlik özelliklerini tanımlayın. Kalıtım (inheritance), değişken bağımlılıkları (dependencies), erişim kontrolleri ve diğer yapısal konulara odaklanılmalıdır.
    - Her yapılan (commit) için özellik (property) testlerini Crytic ile çalıştırın.
- Otomatik araçların kolaylıkla bulamayacağı aşağıdaki sorunlara dikkat edin:
    - Mahremiyet eksikliği: Herkese açık havuzlarda sıraya girerken, diğer herkes işlemlerinizi görebilir
    - Önden giden (front running) işlemler
    - Kriptografik işlemler
    - Harici DeFi (Merkezi olmayan finans - Decentralized Finance) bileşenleriyle riskli etkileşimler

[13] Smart Contract Development Checklist, https://ethereum.org/hr/developers/tutorials/secure-development-workflow/
[14] Token integration checklist, https://ethereum.org/hr/developers/tutorials/token-integration-checklist/

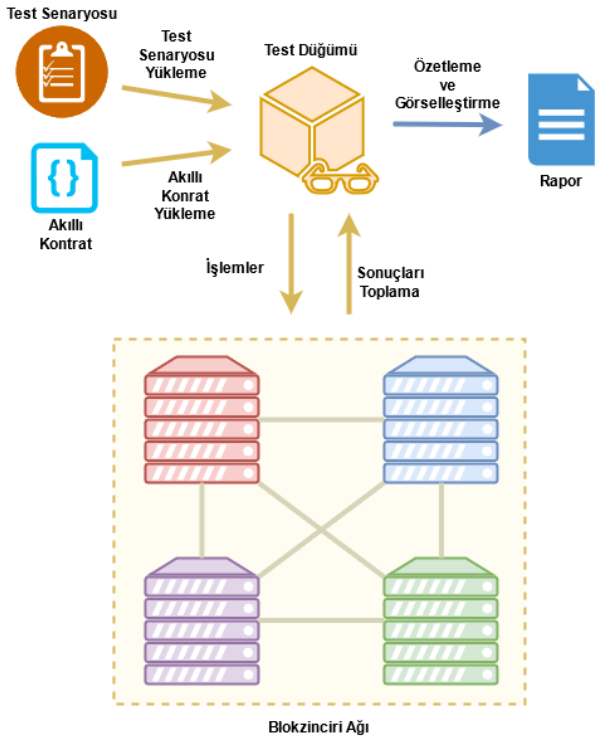
1.7 Blokzincir Testleri

Merkezi olmayan hizmetler giderek daha fazla geliştirilmektedir. Blokzincir ağlarının başarımları ve merkezi olmayan uygulamaların başarım ölçütleri yeterince test edilmemektedir. Yapılabilecek testler iki alt başlık olarak ele alınacaktır.

1.7.1 Blokzincir Ağının Test Edilmesi

Blokzincir ağı test edilerek; gecikme (latency), birim zamanda üretilebilen iş (throughput), kaynak kullanımı ve başarısız/gecikmiş işlem değerleri ölçülebilir. Gecikme değerleri olarak blokzincirinde bir işlemin gerçekleşmesindeki gecikme (transaction latency) ve kayıtlardan okumadaki gecikme (read latency) ölçülebilir. Sistemde birim zamanda üretilebilen iş (throughput) olarak da benzer süreçlerin hesaplamaları yapılabilir. Sistemdeki düğümlerin bu süreçler boyunca kaynak (işlemci, bellek, ağ, vb.) kullanımı da ölçülmelidir [4]. Zaman aşımları nedeniyle başarısız/gecikmiş işlem sayısı da takip edilmesi gereken önemli bir parametredir [10].

Performans değerlendirmesi için kullanılabilecek örnek test süreci Şekil 1.10'da gösterilmiştir. Test edilecek blokzincir ağı ile iletişim kuracak en az bir test düğümü olmalıdır. Test senaryosu; test işlemlerinin kapsamını içeren bir konfigürasyon dosyasıdır. Bu tür bir sisteme; uygulamaya dair özel bir akıllı sözleşme da yüklenebileceği gibi varsayılan (default) bir kontrat da kullanılabilir. Test düğümü; senaryoya göre sistemde yükü oluşturacak ve ardından sonuçları gözlemleyecektir. Test sonuçları özetlendikten sonra görselleştirilerek rapor oluşturulacaktır [3]. Düğümlerin kaynak kullanımı düğümlerin kendilerinde kurulacak sistemlerle takip edilebileceği gibi Simple Network Management Protocol (SNMP) gibi protokoller aracılığıyla da yapılabilir.



Şekil 1.10 Test Süreci [4]

Blokzincir ağlarında kullanılabilecek çeşitli blokzinciri ağ performans test araçları bulunmaktadır, ancak çoğu belirli blokzincir platformları için oluşturulmuştur ve karmaşık konfigürasyon gerektirir. Günümüzde en kapsamlı başarım kıyaslama ortamı Hyperledger Caliper[15]'dir. Quorum dışında birçok platformu

[15] Hyperledger Caliper, https://hyperledger.github.io/caliper/

desteklemektedir. Örnek uygulamalar [10-13]'da verilmiştir. Quorum ortamında Chainhammer[16] ve Quorum Profiling[17] gibi performans test araçları bulunmaktadır. Bu araçların kullanımları kolay değildir, karmaşık kurulum adımlarına ihtiyaç duymaktadırlar. Chainhammer'ın bazı gerekli bağımlılıkları güncel değildir [4].

GoHammer aracı, Ethereum/Quorum blokzincir platformları için kullanımı kolay, esnek bir test aracı sağlamak üzere geliştirilmiştir. Saniyede işlem (TPS) değerleri ve çeşitli performans ölçümlerinin test edilmesi için kullanılabilmektedir. GoHammer üzerindeki geliştirmeler devam etmektedir. Detaylı bilgi için bkz[4].

1.7.2 Akıllı Sözleşme Test Kodları

Ethereum Akıllı Sözleşmeler (Ethereum Smart Contract - ESC) “gas” denilen ücretle sistemde çalışır. Geliştiriciler; kodun sistemde harcayacağı masrafı (gas tüketim seviyelerini) kabul edilebilir bir seviyede tutmak için kodlarını iyileştirilmelidir. Çalışma maliyetlerini en aza indirirken, aynı zamanda kodun çalışma zamanını azaltmak hedeflenmelidir [8]. Wang ve arkadaşları, test üretimini bir Pareto minimizasyon sorunu olarak ele almaktadır ve amacın kapsanmayan dal kapsamını (uncovered branch coverage), zaman maliyetini ve gaz maliyetini en aza indirmek olduğunu vurgulamaktadır [9].

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 kontrattan yararlanabiliriz. Bu da o kontratı diğer test kontratı içerisinde oluşturmamız ile olur. Böylelikle deterministik bir test kontratımız olmuş olur. Aşağıda “remix ide” ile de kullanabileceğiniz “SimpleStorageTest” isimli test kontratı verilmiştir. Bu kontrat her türlü genel ve özel ağlarda test amaçlı kullanılabilir. Bunun için ek bir araç ya da özel bir yazılıma ihtiyacınız bulunmamaktadır.

[16] Chainhammer, https://github.com/drandreaskrueger/chainhammer
[17] Quorum Profiling, https://docs.goquorum.consensys.net/en/stable/Concepts/Profiling/

```
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;
  }
  // ilk değerin 100 olması testi
  function initialValueShouldBe100() public {
        beforeEach();
        // değeri 100 olarak bekliyoruz, ilk seferde pass edecektir.
        // 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 şansı elde edilmektedir.

Farklı bir yöntem olarak “constructor” oluşturularak diğer test fonksiyonlarını da orada çağırmak yapılabilir. Zaten gelişmiş programlama dillerinde de benzer “main” fonksiyonlarla test etme yöntemleri kullanılmıştır. 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);
  }

}
```

1.8 Blokzinciri Ortamında Yazılım Geliştirmede Sıkıntılar ve Fırsatlar

Blokzincir üzerinde yazılım geliştirmekte olan MSKÜ Blokzincir Araştırma Grubu (MSKÜ BcRG) üyelerine blokzincir ortamında yazılım geliştirmede karşılaştıkları ve gözlemledikleri sıkıntıları sorduk. Aldığımız yanıtları şu şekilde özetlemek mümkün:

- Danışılabilecek bilgili kişi az
- Geliştirici topluluğu yeterince gelişmiş değil
- Dökümanlar yetersiz.
    - İnternetteki farklı bilgilerin hangisinin doğru ve güncel olduğu belirsiz
    - Dökümanlardaki örnek kodlar oldukça temel düzeyde.
    - En iyi uygulama (best practice) örnekleri yok denecek kadar az.
- Örnek projeler yetersiz. Bitmiş projeler bulmak zor.
- Bazı kaynaklar oldukça pahalı. Ucunda finansal getiri olma olasılığı, bu işin ticaretine yol açabiliyor.
- Standartlar oturmadığı için, sürekli tekrardan bir şeyleri öğrenmek gerekiyor. Örneğin web teknolojilerinde API geliştirirken sadece o yeni platformu öğrenmek yeterli. Blokzincir ortamlarında ise yeniden öğrenmek gerekiyor.
- Test araçları yetersiz. Kurulum ve kullanımları da çoğunlukla kolay değil.
- Gelişmiş bir blokzincir ekosisteminden söz etmek şu anda mümkün değil. Yazılım geliştirme ekiplerine yeterli ücret ve imkan sağlanması gerekiyor.

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.”

Söz edilenlerin hepsi gözlemlediğimiz eksikler. Bunları çözmek için paylaşıma dayalı çalışmalarımıza devam ediyoruz. Bu konularda daha çok akademik çalışmaya ve fon desteğine ihtiyaç var. Fırsatlar ise birçok alanda devrim yaratacak değişiklikler ve getirileri olacak.

1.9 Sonuç ve Değerlendirmeler

Bu bölümde blokzincir 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 deneyimlerimiz paylaşılmıştır.

İnsanoğlunun kararları önyargılıdır ve bir süreç o 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.

Blokzinciri alanında yetişmiş uzman ihtiyacı bulunmaktadır. Birçok ülke bu konuda çalışmalarda bulunmaya başlamıştır. Örneğin Çin devleti kalkınma planlarına bu konuda eğitim ve geliştirmeleri eklemektedir. Kobi'lerin üzerinde düşük maliyetlerle geliştirme yapabilecekleri BSN (Blockchain Service Network - https://bsnbase.io/) ağının kurulmasına destek verilmektedir. Çindeki firmalar ve dış firmalara ayrı kullanım politikaları uygulanmaktadır[18]. Ülkemizde ise Tübitak Bağ ve DS4H blokzincir ağları gelişmekte olan yapılardır. Güvenilir ve sürdürülebilir blokzincir ağ test ortamları geliştirilmelidir. Bu kapsamda DS4H blokzincir ağını geliştirmeye devam ediyoruz. Herhangi bir merkeze veya kişiye bağlı olmayacak bir ağ için fikir ve katkılarınızı da bekliyoruz. Özellikle akıllı sözleşmelerin güvenlik testleri konusunda çalışmaya devam edeceğiz. Bu konudaki çalışmalara https://github.com/MSKU-BcRG/SC-SecTesting üzerinden ulaşabileceksiniz.

Yönetilebilir ağ servisleri kullanımı; akıllı sözleşme geliştirilmesi, akıllı sözleşmelerin blokzincir sistemine yüklenmesi ve test edilmesi konusunda hem süreçleri kolaylaştıracak hem de hızlandıracaktır. Geliştirilmekte 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.

Blokzincir tabanlı sistemlerde ölçeklenebilirlik (scalability) ve başarım (performance) açısından çözülmesi gereken birçok problem ve fırsat (challenge) bulunmaktadır. Veri bilimi açısından bu sorunları ve çözüm önerileri bir önceki çalışmamızda [1] 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 Interplanetary File System (IPFS) gibi dağıtık dosya sistemlerine entegrasyonu ile hibrid sistemler söz konusu olabilecektir. Bu süreçlerde; ölçeklenebilirlik, birlikte çalışabilirlik ve mahremiyet için önerdiğimiz Çoklu Platform Birlikte Çalışabilir Ölçeklenebilir Mimari (Multi Platform Interoperable Scalable Architecture) MPISA [1] modeline benzer yapıların kullanılması faydalı olacaktır.

Teşekkür

MSKÜ Blokzincir Araştırma Grubu'ndan Cemal Dak, Emre Ertürk, Murat Doğan ve Ümit Kadiroğlu'na çizimler ve ekran çıktılarındaki katkıları için teşekkür ederiz.

Kaynaklar

[1] E. Karaarslan, 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

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

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

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

[4] 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.

[5] 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," In: *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 2020. pp. 515-528.

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

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

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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*

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

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

[16] "GDPR", Official Journal of the European Union, vol. L119, pp. 1-88, April 2016.

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

[18] 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.

[19] 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,” In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019. pp. 1186-1189.

[20] 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.

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

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 blokzinciri araştırma grubunda (MSKÜ BcRG) 2017'den beri blokzincirinin potansiyellerini incelemektedir. DS4H (decentralized solutions for humanity) blokzinciri araştırma ağının kurucularındandır. Araştırma alanları bilgisayar ağları, siber güvenlik, blokzincir, veri bilimi, afet yönetimi ve dijital ikizdir. Akıllı anlaşmalarla ve sıfır bilgi kanıt protokolleri ile güçlendirilmiş mahremiyet, merkezi olmayan kimlik ve blokzinciri 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

Github: https://github.com/MSKU-BcRG

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

Github: https://github.com/melihbirim