Statik Kod Analizinin Yazılım Geliştirme Sürecindeki Yeri ve Yazılım Kalitesine Etkisi

Adnan Kalay

SST - Yazılım Mühendisliği Müdürlüğü, ASELSAN A.Ş., Ankara

e-posta: akalay@aselsan.com.tr

Özetçe

Yazılımların hata tespit ve doğrulama sürecinde birçok teknik kullanılmaktadır. Bu teknikler tek başına yazılımdaki tüm hataları tespit edememekte ve genellikle birbirlerini tamamlayıcı olarak uygulanmaktadır. Statik kod analizi de bu tekniklerden biri olup, yazılımın çalıştırılmadan kodlama zamanında analiz edilmesini ve olası hataların tespit edilmesini sağlamaktadır. Bu bildiride statik kod analizinin önemi, yazılım geliştirme sürecindeki yeri ve yazılım kalitesine etkisi anlatılmaktadır. Bildiride statik kod analizinde incelenen konular, en çok tespit edilen hata tipleri ve maliyet açısından sağladığı kazanımlar farklı statik kod analiz araçlarının gerçek bir proje kaynak koduna uygulanması ve sonuçların sunulması yoluyla anlatılmaktadır. Bildiride ayrıca statik kod analiz tekniğinin karşılaşılan bazı sınırlamalarına değinilmektedir.

1. Giriş

Yazılımların geliştirilmesi sırasında yapılan eklemeler ve değişiklikler, tasarımsal ve kaynak kod hatalarının da zamanla yazılımda yer almasına neden olmaktadır. Bu nedenle geliştirilen yazılımda oluşan hataların ortaya çıkarılması ve giderilmesi için geliştirme süreci adımlarında çeşitli teknikler uygulanmaktadır. Bu teknikler arasında kod gözden geçirmeleri, otomatikleştirilmiş statik kod analizi gibi yazılımın çalışmasını gerektirmeyen statik yöntemler ile birim testleri, yazılım testleri, sistem entegrasyon testleri ve sistem testleri gibi dinamik yöntemler yer almaktadır.

Statik analiz ile yazılımlar çalıştırılmadan analiz edilmekte ve elde edilen analiz sonuçlarına göre gerekli önlemler alınmaktadır. Gözden geçirme yoluyla da yapılabilen statik analiz, günümüzde otomatik kod analiz araçlarının belli bir olgunluğa gelmesiyle birlikte zamansal ve ekonomik maliyet açısından daha uygun olmasından dolayı otomatikleştirilmektedir. Bu analizler sadece olası hataları tespit etmekle sınırlı kalmayıp incelenen yazılımın kalite metrikleri, mimari yapısı, kodlama standartlarına uygunluğu gibi önemli özelliklerini ortaya çıkarmaktadır.

Statik kod analizi, yazılımın çalıştırılmasını ve belli bir test düzeneğinin kurulmasını gerektirmediği için yazılım geliştirme sürecinin her safhasında kolaylıkla yapılabilmektedir. Böylelikle yazılımda meydana gelen hatalar statik kod analizi araçları kullanılarak erken safhalarda tespit edilebilmektedir. Geliştirilen yazılımlarda hata tespit ve düzeltme faaliyetlerinin maliyetinin yazılım yaşam döngüsünün ileri aşamalarına doğru artan bir hızla yükseldiği [1, 2] düşünüldüğünde, statik kod analizinin yazılım geliştirme sürecinin erken safhalarından itibaren benimsenmesinin önemi ortaya çıkmaktadır. Özellikle günümüz yazılımlarının gittikçe karmaşıklaşması ve teslimat sürelerinin kısalması da, yazılım düzeltme faaliyetlerinin mümkün olduğu kadar erken, düşük maliyetle ve etkili bir şekilde yapılmasını gerektirmektedir. Otomatikleştirilmiş statik kod analizi ile yazılımlardaki hatalar erken ve düşük maliyetle tespit edilebilmekte ve böylece fonksiyonel testlerin daha etkili bir biçimde yapılmasına olanak tanınmaktadır [3]. Sonuç olarak daha kaliteli ve güvenilir yazılım ürünleri ortaya çıkmaktadır.

Bildirinin sonraki bölümlerinde sırasıyla statik kod analizinin kapsamı, yazılım geliştirme sürecine dâhil edilmesi, gerçek bir proje koduna uygulanması, sağladığı kazanımlar ve sınırlamaları konu edilmektedir. Son bölümde ise elde edilen bulgular ışığında varılan sonuçlar özetlenmektedir.

2. Statik Kod Analizi

Statik kod analizi, bir yazılımın içerdiği hataların yazılım çalıştırılmadan tespit edilmesi ve sahip olduğu çeşitli özelliklerin elde edilmesi amacıyla yapılan çalışmalar bütünüdür.

2.1. Statik Kod Analiz Şekilleri

Yazılımların statik olarak analiz edilmeleri gözden geçirmeler şeklinde yapılabileceği gibi statik kod analiz araçlarının kullanıma alınmasıyla otomatik olarak da yapılabilmektedir. Her iki yöntemin de avantaj ve dezavantajları bulunmaktadır. Araştırmalar gözden geçirmelerin otomatik analizlere oranla daha etkili olduğunu göstermektedir [4]. Statik analiz araçları daha çok atama ve denetim gibi programlama hatalarını tespit ederken, gözden geçirmeler fonksiyonel hataları da ortaya çıkarabilmektedir [3]. Bununla birlikte, gözden geçirmelerde insan faktörüne bağlı olarak zamanla oluşan dikkat dağılması hataların gözden kaçırılmasına neden olabilmektedir. Gözden geçirmeler zaman ve nitelikli iş gücü gerektirdiği için maliyet açısından da olumsuz karşılanmaktadır. Ayrıca incelenecek kaynak kodun boyutuyla orantılı olarak harcanacak çaba da artacağından ölçeklenebilir değildir.

Otomatik statik kod analizinde ise yazılımda bulunan problemler çok hızlı bir biçimde tespit edilebilmektedir. Araç kullanımı sayesinde farklı boyutlara sahip kaynak kodlardaki hatalar benzer maliyetlerle tespit edilebilmektedir. Ayrıca yazılım hakkında istenen pek çok bilgi ve analiz sonuçları kullanılan araçlar yardımıyla metinsel ve grafiksel olarak elde edilebilmektedir. Bununla birlikte kaynak kodların otomatik olarak analiz edilmesi, yazılım bağlamına tam olarak hâkim olunmamasına neden olabilmektedir. Büyük yazılımlarda yazılım bağlamına tam hâkimiyet beklenmediğinden bu durum bir dezavantaj olarak görülmemektedir. Sonuç olarak, otomatikleştirilmiş statik kod analizinin ölçeklenebilir olması, tutarlı yapısı ve maliyet açısından uygun olması sebebiyle kaynak kod gözden geçirme yöntemine göre öne çıkmaktadır. Bu çalışmada otomatikleştirilmiş statik kod analizi incelenmekte ve bu nedenle bildirinin devamında "statik kod analizi" ve "SKA" ifadeleri, araç kullanımıyla otomatikleştirilmiş statik kod analizi yerine kullanılmaktadır.

2.2. Statik Kod Analizinde İncelenen Konular

Yazılımların statik olarak incelenebilecek çeşitli unsurları bulunmaktadır. Açık kaynak kodlu veya ticari olarak geliştirilen pek çok SKA aracı bu unsurları inceleyerek raporlar üretmektedir. SKA şemsiyesi altına girebilecek bu unsurlar arasında hata ve güvenlik açıklarının tespiti, kaynak kod metrikleri, mimari analiz, kodlama standartlarına uygunluk ve tersine mühendislik çalışmaları gösterilebilir.

2.2.1. Hata ve Güvenlik Açıkları

SKA'da temel amaç yazılımdaki hata ve güvenlik açıklarının tespit edilerek ortadan kaldırılmasıdır. SKA araçları yazılım kaynak kodlarını inceleyerek çeşitli hata tiplerini ortaya çıkarmaktadır. Bu hatalardan en çok karşılaşılanlar arabellek taşmaları, değersiz gösterici kullanımı, yarış durumları, hafıza kaçakları, ilklenmemiş değişken kullanımı, tahsis edilmemiş hafıza kullanımı, kaynakların uygunsuz salınması, tip uyumsuzlukları, ulaşılamayan ölü kodlar ve olası sıfıra bölünme durumu gibi programlama hatalarıdır. Bu hatalar yazılımda herhangi bir koşulda servis reddi, verilerin yetkisiz ifşa edilmesi, değiştirilmesi veya silinmesi ile sonuçlandığında güvenlik zaafı olarak nitelendirilmektedir [5]. SKA araçları inceledikleri yazılımlar üzerinde veri akış analizi, kontrol akış analizi, bilgi akış analizi, yol yordam analizi, uyumluluk analizi, söz dizim denetimi, değer aralığı denetimi, zamanlama analizi, hafıza kullanım analizi ve nesne kodu analizi yaparak hataları tespit etmektedir [6]. Geçmişte meydana gelen yazılımsal felaketlerin birçoğunun statik olarak tespit edilebilecek hatalardan kaynaklandığı düşünüldüğünde hata tespitinde SKA'nın önemi daha iyi anlaşılmaktadır [7].

2.2.2. Kaynak Kod Metrikleri

Yazılımların çeşitli özelliklerini ölçmek ve yazılımı değerlendirmek üzere birçok yazılım metriği geliştirilmiştir [8, 9]. SKA araçları da bu metriklerin ölçümünü yaparak yazılımın kalitesini göz önüne sermeyi amaçlar. Bu araçlar yazılım kalite yaklaşımlarını genellikle belli standartlara dayandırmaktadır [10]. İncelenen yazılım belirlenen kalite metrikleri doğrultusunda değiştirilebilirlik, test edilebilirlik, bakım yapılabilirlik, kararlılık, modüllerin kullanılabilirliği gibi açılardan değerlendirilmektedir. Bu kalite elemanlarının bazıları ölçülen metriklerle doğrudan formülleştirilip gösterilebilmektedir.

2.2.3. Mimari Analiz

Bazı SKA araçları inceledikleri yazılımları mimari açıdan değerlendirebilmektedir. Geliştirilen bir yazılımın belli bir mimariye uygun olup olmadığı, bünyesinde barındırdığı mimari anormallikler, mimarinin esneklik seviyesi gibi konular bu araçların ilgi alanına girmektedir. Yazılım mimarisi değerlendirilirken bileşenler arası bağımlılıklar ve fonksiyon çağırım grafikleri gibi verilerden faydalanılmaktadır. Özellikle bazı araçlar bağımlılık yapı matrisleri ve bağımlılık kurallarını kullanarak yazılım sistemlerinin mimarilerini görselleştirmekte ve test etmektedir[1]. Bu araçlar yazılım geliştirme sürecinin herhangi bir safhasında mimariyi anlamak ve geliştirmek, gereksiz ve zararlı bağımlılıkları ortadan kaldırmak amacıyla kullanılabilmektedir. Gereksiz ve çift yönlü bağımlılıkların kaldırılmasıyla incelenen yazılım mimarisi daha esnek hale gelmektedir. Ayrıca bu araçlar yardımıyla yeni mimari kurallar tanımlanabilmekte ve yazılımın bu kurallara uygunluğu denetlenebilmektedir.

2.2.4. Kodlama Standartları

Yazılımların daha kaliteli ve güvenilir bir şekilde geliştirilmesi için çeşitli kodlama standartları oluşturulmuştur[2] [11]. Ayrıca yazılımların farklı geliştiriciler tarafından daha anlaşılır kılınması için belli kodlama biçemleri geliştirilmiştir. Bu biçem ve standartlara uygun üretilen yazılımlar değiştirilebilirlik, bakım yapılabilirlik gibi kalite ölçütlerini yerine getirirken, diğer taraftan uygulanan resmi yöntemler sayesinde daha güvenilir bir yapıya kavuşmaktadır [6]. Özellikle beklenildiği gibi çalışmaması halinde önemli can ve mal kayıpları meydana gelebilecek güvenlik-kritik yazılımların geliştirme süreçlerinde kodlama standartlarına uygunluk önemli bir yer tutmaktadır [6, 12]. SKA araçları yazılım kaynak kodlarını destekledikleri ilgili kodlama standartlarına göre inceleyerek tespit edilen kodlama ihlallerini raporlayabilmektedir. Bu ihlallerin düzeltilmesiyle olası çalışma zamanı hatalarının da önüne geçilmiş olmaktadır.

2.2.5. Tersine Mühendislik

Bazı durumlarda önceden yazılmış kaynak kodların başka programcılar tarafından idame edilmesi gerekmektedir. Bunun için miras olarak gelen yazılım ya da yazılım parçasının, idame edecek personel tarafından iyi anlaşılması gerekmektedir. SKA araçlarının sağladığı tersine mühendislik alt yapısıyla kaynak kod incelenerek yazılım tasarımı sınıf ilişkileri, mesaj sıra grafikleri gibi gösterimlerle görselleştirilebilmekte ve anlaşılırlığı arttırılabilmektedir[3] [13, 14]. Bu araçlar aynı zamanda "Bilgi Soyutlama Sistemleri" olarak da anılmaktadır [15]. Ayrıca yazılımların yeniden düzenlenmesi faaliyetlerinde de SKA araçlarından yararlanılabilmektedir [16].

2.3. Statik Kod Analiz Araçlarının Kullanımı

Yazılım kaynak kodlarının statik analizleri için araç kullanımının maliyet açısından önemi ortadır. Analiz sırasında 2.2'de belirtilen konulardan bir veya birkaçını inceleyen açık kaynak kodlu ve ticari çok sayıda araç mevcuttur. Etraflı bir analiz için kapsamı en geniş olan aracı tercih etmekte yarar vardır. Bununla birlikte birbirini tamamlayan araçlar da kullanılabilir. Statik olarak incelenebilecek konuların çok çeşitli olması sebebiyle, bazı firmalar tek bir araç yerine bir araç seti sunmaktadır[4]. Bu araç setinde bulunan ilgili araç kullanılarak istenilen analiz yapılabilmektedir.

Mevcut SKA araçlarının bir kısmı yazılımın kaynak kodunu metinsel olarak incelemekte ve ekstra bir bilgiye ihtiyaç duymamaktadır[5]. Daha kapsamlı analiz yapan araçlar ise kullanılan platform ve derleyici bilgilerine ihtiyaç duymaktadır[4,6]. Kaynak kodun derlenerek analiz edilmesi daha güvenilir ve detaylı sonuçların alınmasına olanak sağlamaktadır [12, 16].

[1] http://www.lattix.com/products/LDM.php, son geçerlilik tarihi yoktur.

[2] http://www.misra.org.uk/, son geçerlilik tarihi yoktur.

[3] http://frama-c.cea.fr/, son geçerlilik tarihi yoktur.

[4] http://www.ldra.com/, son geçerlilik tarihi yoktur.

[5] http://www.dwheeler.com/flawfinder/, son geçerlilik tarihi yoktur.

[6] http://www.klocwork.com/, son geçerlilik tarihi yoktur.

3. Statik Kod Analizinin Yazılım Geliştirme Sürecine Dâhil Edilmesi

Özel bir test ortamının kurulmasını gerektirmemesi sayesinde yazılım geliştirme sürecinin paketleme öncesi her adımında rahatlıkla uygulanabilir olan SKA'nın her firmanın kendi geliştirme sürecine en çok uyum gösterecek şekilde entegre edilmesi azami faydayı sağlayacaktır. SKA'nın benimsenmesi ve etkili olarak kullanılması sadece aracın elde edilmesine bağlı olmayıp, iyi bir stratejinin ve örgütsel yönlendirmelerin belirlenmesini gerektirmektedir [17]. Bu nedenle SKA'nın yazılım geliştirme sürecine dâhil edilmesi aşamasında, SKA aracının kimler tarafından kullanılacağı, hangi süreç adımlarında kullanıma alınacağı ve sonuçlarının nasıl değerlendirileceği belirlenmelidir.

3.1. SKA Uygulanmasında Bireyler Arası İş Bölümü

Literatürde SKA'nın kullanıma alınmasıyla ilgili çeşitli çalışmalar bulunmaktadır [3, 17]. Örneğin Nortel Networks firmasında kaynak kod belli bir olgunluğa ulaşıncaya kadar SKA araçları ile uzman bir grup tarafından değerlendirilerek analiz hatalarının bir ön elemeye tabi tutulduğu ve böylece yanlış uyarıların önemli ölçüde azaltıldığı raporlanmıştır. Yine bu uzman grup tarafından araç kullanımıyla tespit edilen gerçek hataların yazılım geliştirme ekibine iletilerek düzeltildiği ve bu aşamanın sonunda geliştirme ekibinin SKA aracını artık ek hata tespit filtresi olarak kullanacağı temiz bir yazılım taban çizgisi elde edildiği belirtilmiştir [3]. [17]'de de statik analiz aracı kullanımında hata analizi ve geliştirme ekibinin ortak çalışması önerilmektedir.

3.2. SKA'nın Kullanıma Alınacağı Süreç Adımları

SKA'nın kullanıma alınabileceği yazılım geliştirme adımları olarak ise çeşitli öneriler bulunmaktadır [17]. Bunlardan ilki kaynak kodun derlenmesi aşamasıdır. SKA araçlarının derleme aşamasında otomatik olarak kullanıma alınması tekrarlanabilirlik sağlarken harcanan süre açısından bir dezavantaj getirmektedir. Bu nedenle statik analizin geliştirici tarafından istenildiğinde otomatik olarak çalıştırılacak şekilde derleme sürecine bağlanması gerekebilir. Benzer bir yöntem de kaynak kodun geliştirme ortamında yazılması sırasında taranarak kontrol edilmesidir. Diğer bir yöntem ise, SKA'nın yazılım geliştirme ortamında var olan sürekli entegrasyon alt yapısının bir parçası olarak yapılmasıdır. Bu yöntemler ile yazılımda yer alan olası hatalar geliştirme sürecinin erken safhalarında tespit edilip ayıklanabilir ve böylelikle düzeltme maliyetleri düşük seviyede tutulabilir. Ayrıca yazılımın geliştirme sırasında sürekli olarak analiz sonuçlarının elde edilmesi, öğretici sonuçlar doğurarak geliştirme ekibinin daha az hatalı kodlar yazmasını sağlaması beklenmektedir.

SKA'nın yazılım geliştirme sürecine dâhil edilmesi aşamasında ilk adımın en az sorun çıkaracak yöntemle atılması uygun olacaktır. İdeal kullanım olarak, statik analizin yazılım geliştirme sırasında uygulanarak hem hataların erken tespiti hem de öğretici olması beklenmektedir. Fakat bu yönteme doğrudan geçiş yerine, statik analizin öncelikle belirli yazılım kilometre taşlarında uygulanması ve sonrasında bu sürecin zamanla güncellenerek ideal duruma yaklaşılması daha uygulanabilir bir yaklaşım olarak öne çıkmaktadır. Bu kademeli geçiş sayesinde SKA'nın yazılım geliştiriciler arasında benimsenmesi de daha kolay olacaktır. Bununla birlikte proje takvimlerinin sıkışık olması ve diğer kısıtlar yazılım geliştiricilerin SKA'ları yapmama eğiliminde olmalarına neden olmaktadır. Bu problem SKA'nın girdi ve çıktılarıyla beraber yazılım geliştirme sürecinde somut olarak yer almasıyla giderilebilir. Yazılımın statik analiz sonuçlarının saklanması, gerek sonradan gerekebilecek analizler için, gerekse müşterinin muhtemel inceleme isteği açısından önem taşımaktadır.

3.3. SKA Sonuçlarının Değerlendirilmesi

Yazılımın statik analizi yapıldıktan sonra elde edilen sonuçların nasıl değerlendirileceği de önem taşımaktadır. Tespit edilen hatalara risk seviyelerine göre öncelikler atanması ve hataların bu doğrultuda düzeltilmesi gerekmektedir. Ayrıca araç tarafından tespit edilen bazı zaaflar gerçekte hata olmalarına karşın zararsız ya da kritiklik seviyesi çok düşük hatalar olabilmektedir. Bu gibi durumlarda maliyet/fayda analizi yapılarak hataların düzeltilip düzeltilmemesine karar verilmelidir. Ayrıca SKA araçlarının yapılandırma alt yapıları kullanılarak belli tiplerdeki hataların uyarı olarak verilmemesi sağlanabilir.

Yazılım kaynak kodunun metrikleri çıkarılarak yazılım kalitesinin elde edilmesi içinse öncelikle belli bir kalite modelinin oluşturulması gerekmektedir. Tüm projeler için ortak olabilecek bu kalite modeli, aynı zamanda istenen belli projeler için özelleştirilebilir olmalıdır. Sonuç olarak kullanılacak SKA aracı, ilgili kalite modelinin içerdiği kalite özellikleri doğrultusunda belirlenen kalite metriklerini ölçmek üzere yapılandırılmalıdır. Bu yapılandırma ile statik analiz araçlarının kalite modelinde tanımlanmış metrikler doğrultusunda analiz yapması ve yazılım kalite değerlendirmesinin belirlenen kalite modeline uygun olarak yapılması sağlanmış olur.

3.4. ASELSAN Bünyesinde SKA Kullanımı

ASELSAN SST grubu bünyesinde geliştirilen atış kontrol ve silah sistemleri yazılımlarının yüksek kritiklik düzeyine sahip olması, bu yazılımların gerek statik gerekse dinamik yöntemlerle kapsamlı olarak analiz edilmesini gerektirmektedir. Bu amaçla yazılımların statik analizi için LDRA Testbed ve Telelogic Tau Logiscope araçları kullanılmaktadır. SKA yazılım geliştirme sürecinin en başından itibaren değil, yazılım kaynak kodu belli bir olgunluğa eriştiğinde ve belli kilometre taşlarında yapılmaktadır. Bu sayede önemli süre kayıpları önlenmekte ve SKA'nın kullanıma alınması daha uygulanabilir kılınmaktadır. SKA yapılacağı zaman öncelikle ilgili yazılımın kalite öncelikleri doğrultusunda aracın sunduğu analiz konuları içerisinden bir alt küme seçilmektedir ve daha sonra bu alt küme doğrultusunda gerekli analizler yapılmaktadır. Bu sayede araç raporlarını inceleme süresinden kazanç sağlanmaktadır. Araçlar tarafından raporlanan olası yazılım hataları öncelikle incelenerek yanlış hata uyarıları ayıklanmaktadır. Tespit edilen gerçek hatalara ise uygun risk seviyeleri atanarak gerekli düzeltme faaliyetleri yürütülmektedir. Düzeltme faaliyetlerinin tamamlanmasının ardından yazılımlar bir kez daha analiz edilerek hataların giderildiğinden ve yeni hataların meydana gelmediğinden emin olunmaktadır. Bunun yanında yazılımlar hakkında edinilen kaynak kod metrikleri incelenerek gerekli görülen yazılım parçalarında yeniden düzenleme çalışmaları uygulanmaktadır. Bu sayede ortaya daha kaliteli ve güvenilir yazılımlar çıkmaktadır. Ayrıca araçlardan elde edilen çeşitli raporlar, ileride kullanılmak üzere referans olarak saklanmaktadır. SKA kullanımında edinilen tecrübeler arttıkça

ve geliştiriciler arasında yaygınlaştıkça, statik analizin ideal kullanımına uygun olarak geliştirme sürecinin başından itibaren kullanıma alınabileceği düşünülmektedir.

4. Statik Kod Analizinin Uygulanması

Bu uygulama kapsamında ASELSAN SST grubu bünyesinde C++ programlama dili ile geliştirilen bir atış kontrol projesinin yaklaşık 250.000 satıra sahip kaynak kodu farklı SKA araçlarıyla analiz edilmiştir. Bu analiz çerçevesinde çeşitli ilgi odakları doğrultusunda bulgular tespit edilerek SKA'ya ilişkin çıkarımlar elde edilmiştir. Bu sonuçların değerlendirilmesi suretiyle SKA'nın yazılım geliştirme sürecine sağlayacağı katkılar araştırılmıştır. Uygulama süresince aşağıdaki sorulara cevap aranmıştır:

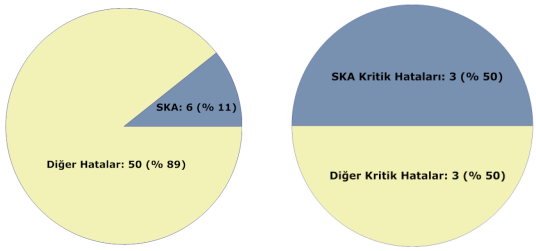
- SKA ile yazılım iyileştirme ve test maliyetleri ne kadar azaltılabilir?
- SKA hangi türdeki hataları tespit etmektedir ve yazılım testlerinde karşılaşılan hatalarla ilişkisi nedir?
- SKA ile yazılımda bulunan problemli bileşenler tespit edilebilmekte midir?
- SKA ile yazılım geliştiricilerin en çok hangi programlama hatalarını yaptıkları tespit edilebilir mi? Bunlara çözüm önerileri sunulabilir mi?
- SKA'da kullanılan araçlar genellikle ortak tespitlerde mi bulunmaktadır, yoksa birbirini tamamlayan analizler mi gerçekleştirmektedir?

Yukarıda belirtilen ilgi odaklarına yönelik yürütülen çalışmada ASELSAN SST grubu bünyesinde mevcut olan LDRA Testbed ve Telelogic Tau Logiscope araçlarının yanı sıra, açık kaynak kodlu olarak temin edilebilen Flawfinder ile firmalarından deneme sürümleri istenen Klocwork, RSM ve Lattix LDM araçları kullanılmıştır. Statik analizler sonucunda elde edilen bulgular, belirtilen sorular ekseninde alt bölümlerde sırasıyla değerlendirilmektedir.

4.1. Statik Kod Analizinin Yazılım İyileştirme ve Test Maliyetlerine Etkisi

Bu analizde yazılım geliştirme sürecince kaydı tutulan yazılım sorunları ve iyileştirme isteklerinden ne kadarının SKA ile tespit edilebileceği ve daha önceden önlem alınabileceği incelenmiştir. ASELSAN SST grubu bünyesinde yazılım geliştirme süresince belirli zaman aralıklarında testler yapılmakta ve tespit edilen hatalar değişiklik yönetim sistemine girilmektedir. Ayrıca kaynak kod üzerinde yapılan ve resmi olmayan gözden geçirmeler sonucu ortaya çıkan iyileştirme istekleri de bu sistemde kayıt altına alınmaktadır. Buna göre projenin belirli bir kilometre taşında alınan kayıtlar doğrultusunda karşılaştırmalar yapılmıştır.

Şekil 1'de testler sonucu ortaya çıkarılan 56 hatadan 6'sının SKA ile daha önceden tespit edilebileceği görülmektedir. Bu sonuca göre testlerde ortaya çıkarılan hataların yaklaşık %10'u statik analiz ile daha önceden tespit edilebilirdi. Bunun yanında resmi olmayan testlerde bulunan her hatanın kayıt altına alınmamasından dolayı bu oranın beklenenden düşük çıktığı düşünülmektedir. Daha ilginç bir sonuç ise yüksek kritikliğe sahip hataların tespit edilme oranıdır. Kayıt altına alınan 56 hatadan 6'sının sistemin servis dışı kalmasına neden olan yıkıcı hata olduğu ve bu hatalardan 3 tanesinin SKA ile tespit edildiği görülmüştür. Bu durumda Şekil 1'de gösterildiği gibi, SKA ile testlerde tespit edilen yıkıcı hataların %50'sinin daha önceden giderilebileceği görülmüştür.

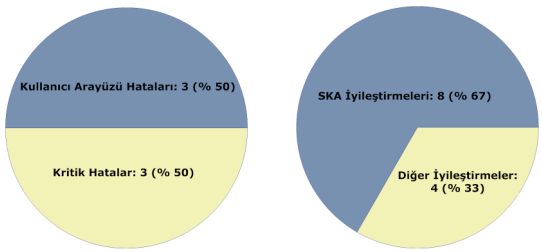


(a) (b)

Şekil 1: SKA ile Tespit Edilen Hataların Oranı (a), Kritik Hataların SKA ile Tespit Oranı (b)

Şekil 1'de ifade edildiği gibi SKA ile tespit edilen hataların %50'sini çok kritik hatalar oluştururken diğer %50'sinin ise kullanıcı arayüzü hataları olduğu Şekil 2'deki gibi tespit edilmiştir. Bu hatalardan sistemin çalışmasını engelleyen hataların arabellek taşmalarından kaynakladığı anlaşılırken, kullanıcı arayüzü hatalarının ise ilklenmemiş değişkenlerin kullanımı nedeniyle ortaya çıktığı görülmüştür.

Kayıt altına alınan hataların yanı sıra, resmi olmayan gözden geçirmeler sonrasında ortaya çıkan iyileştirme isteklerinin de SKA ile ilişkisi incelenmiştir. Kayıt altına alınmış 22 tane iyileştirme isteği tespit edilmiştir. Fakat bu isteklerin 10 tanesinin fonksiyonel iyileştirme olduğu görülmüştür. Kod kalitesini arttırmaya yönelik olan 12 isteğin 8 tanesinin SKA ile daha önceden tespit edilebileceği belirlenmiştir. Bu durumda kod gözden geçirmeleri ile ortaya çıkan iyileştirme isteklerinin Şekil 2'de görüldüğü gibi %67'sinin daha önceden tespit edilebileceği belirlenmiştir. SKA tarafından tespit edilemeyen iyileştirme isteklerinin ise yazılım geliştirme ekibi tarafından kullanılan model tabanlı geliştirme aracına ait model özelliklerinin değiştirilmesi, bazı yazılım parçalarının projeye eklenip çıkarılması ve mantıksal ilişkilerin güncellenmesiyle ilgili olduğu saptanmıştır.

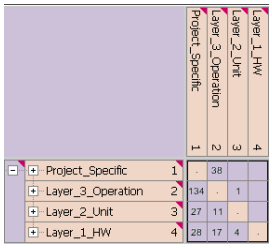


(a) (b)

Şekil 2: SKA ile Tespit Edilen Arızaların Dağılımı (a), SKA ile Tespit Edilen İyileştirme İsteklerinin Oranı (b)

Yazılım parçaları arasındaki istenmeyen bağımlılıkların kaldırılmasına yönelik istekler Lattix LDM aracı kullanılarak kolaylıkla tespit edilmiştir. Araç, incelenen projenin bağımlılık yapı matrisinin oluşturularak yazılımdaki bağımlılıkların görselleştirilmesini sağlamaktadır. Bu matriste satırlar kullanılan modülleri gösterirken, sütunlar ise kullanan modülleri ifade etmektedir. Katmanlı mimariye sahip bir

yazılımın bağımlılık yapı matrisinde köşegenin üstünde hiçbir bağımlılık bulunmamalıdır. Lattix LDM aracı ile ayrıca aynı katmanda bulunan modüller arası bağımlılık kuralları tanımlanabilmektedir. Şekil 3'te incelenen yazılımın bağımlılık yapı matrisi gösterilmektedir. Matristeki satırlar aşağıdan yukarı, sütunlar ise sağdan sola doğru en alt katmandan en üst katmana doğru bir geçiş göstermektedir. Matris incelendiğinde, ikinci katmandaki bir bileşenin daha üst katmanda yer alan üçüncü katmana ait bir bileşeni kullandığı görülmektedir ve bu istenmeyen bağımlılık kaldırılmalıdır. Üçüncü katmandan projeye özel katmana 38 tane bağımlılık gösterilmiş olsa da, bu bağımlılıkların çoğu gerçekte yine projeye özel olan bazı bileşenlerin üçüncü katmanda yer almasından kaynaklanmaktadır.

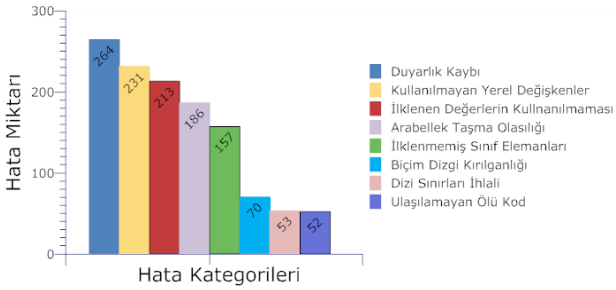


Şekil 3: Yazılım Bağımlılık Yapı Matrisi

İyileştirme isteklerinin bu şekilde daha önceden tespit edilmesi hem kod gözden geçirmelerinde kaybedilen vaktin azaltılması, hem de gerekli değişikliklerin daha az maliyetle yerine getirilmesi açısından önem taşımaktadır.

4.2. Statik Kod Analizi ile Tespit Edilen Hatalar ve Yazılım Testlerinde Karşılaşılan Hatalarla İlişkisi

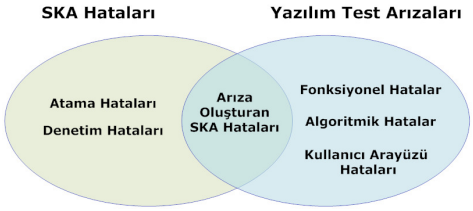
Bu analizde SKA ile hangi yazılım hatalarının çoğunlukla bulunduğu ve bu hataların yazılım testlerinde ortaya çıkan hatalarla olan ilişkisi araştırılmaktadır. Yapılan statik analizlerde yazılıma ait çeşitli hatalar tespit edilmiştir. Bu hataların çoğunluğunun dağılımı Şekil 4'te gösterildiği gibidir.



Şekil 4: SKA ile Tespit Edilen Hataların Dağılımı

SKA tarafından tespit edilen bu hatalardan 6 tanesi arıza olarak tespit edilmiştir. Literatürde yapılan çalışmalara bakıldığında yazılımda bulunan kodlama ve tasarım hatalarının sadece belli bir kısmının arıza olarak ortaya çıktığı raporlanmaktadır [5, 18]. Buna göre SKA ile sadece testlerde karşılaşılan sorunlar değil, sistem teslimatından sonra ortaya çıkabilecek birçok problem de tespit edilebilmektedir. Bu hataların ortadan kaldırılmasıyla daha kaliteli ve güvenilir yazılımların dağıtılması hedeflenmektedir. Fakat yine literatürdeki çalışmalara bakıldığında SKA ile tespit edilen hataların ortalama arıza oluşturma sürelerinin farklılık gösterdiği ve bu sürenin düşük olduğu hatalara odaklanılması gerektiği vurgulanmaktadır [18].

SKA sonucunda tespit edilen hatalar Şekil 4'ten de anlaşılacağı gibi çoğunlukla değişken atamaları ve denetim eksikliğiyle ilgili olan programlama hataları olarak karşımıza çıkmaktadır. Yazılım testlerinde tespit edilen arızalar ise daha çok fonksiyonel, algoritmik ve kullanıcı arayüzü hatalarıdır. Bu iki yazılım doğrulama yönteminin ortaya çıkardığı hata türlerinin farklılığı birbirlerini tamamlayıcı olarak kullanılmaları gerektiğine işaret etmektedir. SKA tarafından tespit edilen programlama hataları, sonuçlarının yazılımın çalışmasına engel olduğu veya kullanıcı arayüzünün beklenildiği gibi çalışmamasına neden olduğu durumlarda yazılım testlerinde ortaya çıkan sorunlarla kesişmektedir. Hatalar arasındaki bu ilişki aşağıdaki şekilde ifade edilmektedir.



Şekil 5: SKA Hataları ile Yazılım Testleri Sırasında Tespit Edilen Arızaların İlişkisi

4.3. SKA ile Problemli Bileşenlerin Tespit Edilmesi

SKA ile tespit edilen hataların dağılımı göz önüne alınarak problemli bileşenlerin önceden tespit edilip edilemeyeceği bu analiz çerçevesinde incelenmiştir. Problemli yazılım bileşenleri çalışma zamanında arızaya sebep olan program parçaları olarak kabul edilerek, bileşenlerin statik analiz sonucu sahip olduğu tespit edilen hata miktarı ve yoğunluğunun problemli bileşenlerle olan ilişkisi irdelenmiştir. Bileşenlerin hata oranlarının belirlenmesinde ise Klocwork Insight aracının kategori detaylarını listeleyen özelliğinden yararlanılmıştır. Bu araç belli hata kategorilerinde en fazla hata miktarı ve yoğunluğuna sahip yazılım parçalarını listelemektedir. Buna göre, arıza oluşturan SKA hatalarının tespit edildiği bileşenlerin Klockwork aracı tarafından ilgili hata kategorilerindeki sıralamaları incelenmiştir. Bu inceleme sonucunda Tablo 1'de yer alan sonuçlar elde edilmiştir.

Tablo 1: Arıza Oluşturan Yazılım Bileşenlerinin Hata Kategorilerindeki Sıralamaları

Yazılım Bileşeni	Hata Kategorisi	Sıralama
Tanım Okuyucu	Uygunsuz Bellek Salımı	1
Hata Kodları Menüsü	Hafıza Kaçakları	3
Semboloji Yöneticisi	İlklenmemiş Değişken	9
Menü Görüntüleyici	Arabellek Taşmaları	1
Platform Yöneticisi	Kullanılmayan Değişkenler	2
Eksen Limitleri Menüsü	-	-

Tablodan da anlaşılacağı üzere çalışma zamanında arıza oluşumuna sebep olmuş problemli bileşenler, genellikle hata sayısı ve yoğunluğuna göre oluşturulmuş kategori detaylarında üst sıralarda yer almaktadır. Bu durum da bir bileşenin statik analiz sonucu sahip olduğu tespit edilen hata sayısı ve yoğunluğunun, ilgili bileşenin çalışma zamanında arızaya sebep olacak problemli bir yazılım parçası olup olmadığı konusunda önemli veriler sunduğunu göstermektedir. İstisnalar arasında platform yöneticisi ve eksen limitleri menüsü bulunmaktadır. Platform Yöneticisindeki arıza ilklenmemiş değişken nedeniyle kaynaklanırken, araç tarafından başka bir kategoride yer almıştır. Eksen limitleri menüsü ise herhangi bir kategoriye dâhil edilmemiştir. [19]'da yapılan benzer bir araştırmada da SKA hata yoğunluğu ile sürüm öncesi hata yoğunluğu arasındaki ilişki incelenmiş ve %82,9 oranında bir doğruluk ile sürüm öncesi hata eğilimli bileşenlerin tespit edildiği raporlanmıştır.

Yazılımların ve yazılım bileşenlerinin sürüm öncesi hata yoğunluğunun kestirilmesi, yazılımın güvenilirliğinin tespiti ve gerekli kararların zamanında alınması açısından büyük önem taşımaktadır. Yazılım endüstrisinde bu tip kestirimlerin, yazılım kalitesini iyileştirme faaliyetlerinin uygulanabilirliği açısından genellikle çok geç elde edildiği belirtilmektedir [19].

4.4. SKA ile Genel Programlama Hatalarının Tespit Edilmesi ve Çözüm Önerileri Sunulması

SKA sonuçları incelendiğinde, araçların tespit ettikleri hataları nedenleriyle beraber listeledikleri ve bunlara çözüm önerileri sundukları görülmüştür. Özellikle arabellek taşmalarına sebep olabilecek *sprintf*, *strcpy*, *strcat* gibi korumasız dizgi fonksiyonları yerine korumalı olan *snprintf*, *strncpy*, *strncat* fonksiyonlarının kullanımı önerilmektedir. Ayrıca sabit boyutlu dizilere kontrollü erişim yapılması gerektiği belirtilmektedir. Bu tip güvensiz programlama pratikleri hakkındaki araç önerilerinin yazılım geliştiricilere geri besleme olarak verilmesi ile daha güvenli programlamaya geçilebileceği ve yazılım güvenilirliğinin proaktif bir şekilde arttırılabileceği düşünülmektedir. Araç çıktıları arasında servis reddine sebep olan hataların yanı sıra, işlevselliği bozan duyarlık kaybı ve ilklenmemiş değişkenler gibi kodlama kusurları için de öneriler yer almaktadır.

Kaynak kod metriklerini analiz eden araçların da yazılımın okunabilirlik, bakım yapılabilirlik gibi kalite özelliklerini olumsuz yönde etkileyen kodlama kusurlarını ortaya çıkardığı ve bu kusurların nasıl giderileceğine dair yönlendirmelerde bulunduğu görülmüştür. Özellikle kaynak koda yorumların eklenmesi, kod bloklarının küme parantezi içerisine alınması, kod satırlarının belli bir uzunluğu aşmaması, *switch* ifadelerine *default* bölümü eklenmesi, kontrol ifadeleri içerisinde değişken ataması yapılmaması gibi bakım yapılabilirliği arttıran tavsiyeler araçlar tarafından belirtilmiştir. Tespit edilen önerilerden önemli görülenlerin geliştiricilere iletilmesi ve uygulamaya alınması ile daha kaliteli yazılımların oluşturulacağı değerlendirilmektedir.

4.5. Farklı Statik Kod Analiz Araçlarının Kullanımı

Bu uygulamada farklı SKA araçları kullanılarak ortaya koydukları bulguları değerlendirme fırsatı da doğmuştur. Alınan sonuçlar, araçların incelenen yazılımın farklı özelliklerini ortaya çıkarmakta daha başarılı olduklarını göstermiştir. Örneğin Klocwork Insight aracı olası hataları tespit ederken, Logiscope ve RSM araçları daha çok kaynak kod metriklerini çıkarmaktadır. Flawfinder aracının da hataları tespit etmesine karşın, Klocwork Insight aracının daha geniş bir hata kümesini tespit ettiği görülmüştür. LDRA araç setinde ise kod ihlallerinin ve metriklerin ortaya konulmasının yanı sıra, birim test ve dinamik analiz gibi fazladan kabiliyetler mevcuttur. Lattix LDM aracının yazılımın mimarisinin incelenmesi ve mimari katmanların bağımlılıklarının ortaya çıkarılmasında başarılı olduğu saptanmıştır. Kullanılan araçlar belirtilen özelliklerden daha fazlasına sahip olsa da, her aracın belli bir konuda daha gelişmiş olduğu görülmüştür. Ortak inceleme konularına sahip iki araçtan ilki bir konuda daha iyi ve anlaşılır sonuçlar üretirken, ikinci aracın ise diğer bir konuda daha iyi sonuçlar ortaya çıkardığı tecrübe edilmiştir. Bu durumda SKA için tek bir araç kullanmak yerine, farklı yönlerde gelişmiş araçların birbirlerini tamamlayıcı olarak kullanılmalarının en uygun yöntem olacağı sonucuna varılmıştır.

5. Statik Kod Analizinin Sağladığı Kazanımlar

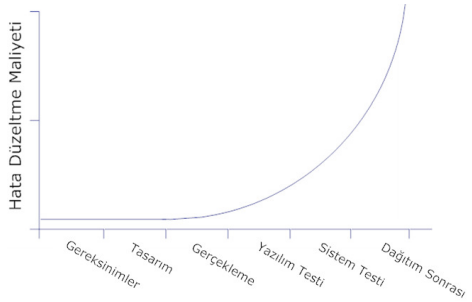
Statik kod analizinin yazılım geliştirme sürecinde yerini alması ile yazılımların daha hızlı ve kaliteli geliştirilmesi sağlanabilecektir. Bu durumu mümkün kılan kazanımlar alt başlıklar halinde sıralanmıştır.

5.1. Gözden Geçirme Sürelerinin Azalması

Yazılımlar, geliştirme sürecinin belirli evrelerinde gözden geçirmelere tabi tutulmaktadır. Resmi olmayan gözden geçirmeler geliştirme döneminin herhangi bir zamanında yapılabilirken, resmi gözden geçirmeler ise genellikle yazılım tasarımı ve gerçeklemesi tamamlandığında yapılmaktadır. Gözden geçirmelerin ölçeklenebilir olmaması nedeniyle, yazılımın boyutuyla orantılı olarak harcanan süre de artmaktadır. Yazılımların gözden geçirmelerden önce SKA ile incelenmesi ise bu süreyi oldukça azaltmaktadır. Statik olarak tespit edilebilen pek çok kodlama hatasının bu şekilde önceden düzeltilmesi, gözden geçirmelerde daha çok statik olarak tespit edilemeyen, fonksiyonel anormalliklerin incelenmesini mümkün kılmaktadır. Bu sayede kod gözden geçirmelerinde kaybedilen süre önemli oranda azaltılmaktadır. SKA'ya uygun olarak hazırlanan tasarımların gözden geçirmelerinde de aynı şekilde faydaların sağlanacağı düşünülmektedir.

5.2. Test Maliyetlerinin Azalması

Yazılımlar, geliştirme süreci boyunca çeşitli testlere tabi tutulmaktadır. Bu testlerde tespit edilen hatalar giderilerek yazılımın güvenilirliği arttırılır. Fakat hataların test aşamasından daha önce tespit edilmesi, test maliyetlerinin düşmesini sağlayacaktır.



Şekil 6: Yazılım Geliştirme Yaşam Döngüsündeki Hata Düzeltme Maliyetleri [1]

Şekil 6'da gösterildiği gibi yazılımdaki hataları düzeltme maliyetinin yazılım yaşam döngüsünün ilerleyen fazlarında giderek arttığı belirtilmektedir. Buna göre, yazılım ve sistem testlerinde karşılaşılan hataların daha önceki geliştirme fazlarında tespit edilmesi daha düşük maliyetlerle düzeltilmesine olanak tanıyacaktır. Ayrıca testlerde harcanan emek ve süre de azalacaktır. Bu sebeplerden dolayı, yazılım gerçekleme zamanında yapılan SKA ile olası test hatalarının bulunması test maliyetlerinin azalmasını sağlamaktadır.

5.3. Bakım Maliyetlerinin Azalması

Geliştirilen yazılımlar sahaya sürülmeden önce mümkün olduğunca hatalardan arındırılmaya çalışılmaktadır. Bu sayede yazılımların hata giderme verimi arttırılmaktadır. [8]'de hata giderme verimi yüksek olan yazılım projelerinin kısa geliştirme süresi, düşük geliştirme maliyeti, yüksek müşteri memnuniyeti ve yüksek ekip moraline sahip olduğu belirtilmektedir. Fakat yazılımları tamamen hatasız dağıtmak neredeyse imkânsızdır [8]. [1]'de yazılımlarda bulunan hataların genellikle %10'luk bölümünün dağıtım sonrasında tespit edildiği vurgulanmaktadır. Dağıtım sonrası hataları düzeltme maliyeti de dikkate alındığında, bu oranın çok yüksek düzeltme maliyetleri getirdiği görülmektedir. Bu çalışmada gerçekleştirilen uygulama, SKA'nın testlerde karşılaşılabilecek hataların dışında pek çok anormalliği ortaya çıkardığını göstermiştir. SKA ile yazılımdaki hataların mümkün olduğunca bulunup düzeltilmesi, bakım zamanında ortaya çıkabilecek arızaların geliştirme zamanında çok daha düşük maliyetle giderilmesini sağlayacağı gibi, aynı zamanda müşteri memnuniyetini de arttıracaktır.

Bu maliyet analizlerinde statik kod analizinin geliştirme sürecine katılım maliyeti de hesaba katılmalıdır. Araçların kullanımının öğrenilmesi ve yeterli tecrübenin kazanılması gerekliliği başlangıçta belli bir maliyet getirse de, bu maliyetin zamanla azalacağı ve getirisinin çok daha fazla olacağı düşünülmektedir.

5.4. Yazılım Kalitesinin Yükselmesi

Yazılım kalitesi doğruluk, kullanılabilirlik, bakım yapılabilirlik, test edilebilirlik gibi çeşitli faktörlere bağlıdır. SKA sonucunda olası hataların önceden giderilmesi, yazılımın doğruluğunun artmasını ve böylece saha kullanımı sırasında daha az arıza ile karşılaşılmasını sağlamaktadır. Kaynak kod kalitesini ölçen SKA araçlarının kullanıma alınmasıyla ise diğer kalite faktörlerine ait metrikler elde edilerek gerekli düzeltmeler yapılmakta ve yazılım kalitesi arttırılmaktadır. Ayrıca yazılımın mimari açıdan incelenmesi ile mevcut mimari anormallikler giderilirken, yazılım tasarımının güçlü ve zayıf yönleri ortaya konulmaktadır. Yazılımların statik olarak incelenmesi sırasında performansın iyileştirilebileceği yazılım parçaları da tespit edilmektedir. Bu kod bölümlerinin iyileştirilmesiyle, daha yüksek performansa sahip yazılımlar elde edilmektedir. Son olarak, yazılımların istenen kodlama standartlarına uygunluğunun kontrol edilmesi sayesinde, bu standartların getirdiği kalite özelliklerinin yazılımlara uygulanması sağlanmaktadır.

6. Statik Kod Analizinin Sınırlamaları

Bu bölümde SKA'nın uygulanması sırasında karşılaşılan bazı sınırlamalara değinilmektedir. SKA araçlarının güvenilirlik ve bütünlük olmak üzere iki temel özelliği bulunmaktadır [4]. Güvenilirlik yazılımda yer alan hataların bulunma oranını gösterirken, bütünlük ise sahte hatalarla karşılaşılma seyrekliğini belirtir. Bu iki özellik arasında bir ödünleşim bulunmakta ve pratikte hiçbir araç aynı anda tamamen güvenilir ve bütün olamamaktadır. Bu nedenle SKA'da tespit edilen olası hataların tecrübeli kişilerce incelenmesi ve sahte hata olduğu anlaşılan uyarıların düzeltme sürecine dâhil edilmemesi gerekmektedir [17]. Sahte hata sayısı arttıkça, bu kontrollerde harcanan emek de artmaktadır. Sahte hataların belli bir bölümü kaynak kodların gereğinden fazla karmaşık olmasından kaynaklanmakta ve daha basit kodlamalar ile bu sorunun azaltılabileceği düşünülmektedir [16]. SKA sonrasında tespit edilemeyen gerçek hataların ise gözden geçirmeler ve test aşamalarında mümkün olduğunca yakalanması yazılımın güvenilirliğini arttıracaktır.

Geçmişte yaşanan tecrübeler doğrultusunda, göz ardı edilen bazı SKA hatalarının sürüm sonrası önemli sorunlara neden olduğu görülmüştür[1]. SKA sonucunda tespit edilen bazı olası zaafların ise gerçek problemler olmamasına karşın tam olarak anlaşılmadan düzeltilmeye çalışılması sonucunda gerçek problemlerin oluştuğu belirtilmiştir[1]. Bu nedenle SKA sonuçlarının deneyimli kişilerce ön incelemeye tabi tutulması gerekmektedir. Bu gereklilik ekiplere belli bir maliyet yükü getirmektedir. Bu yükün en aza indirilmesi için ise, güvenilirlik ve bütünlük arasındaki dengeyi en uygun şekilde kurmuş araçlar kullanılmalıdır.

Bu çalışmada karşılaşılan en önemli sınırlamalardan biri ise modelleme aracı tarafından otomatik olarak üretilen kaynak kodların incelenmesi ve raporlanmasıdır. Yazılım geliştiricilerin otomatik üretilen bu kodlara müdahalesi uygulanabilir olmamakta ve kod parçalarıyla ilgili üretilen raporlar inceleme süresini gereksiz yere arttırmaktadır. Geliştiricilerin yalnızca elle yazılan kaynak kodları analiz ederek düzeltmesi, modelleme aracı kaynaklı hataların ise sonraki versiyonlarda giderilmesi beklenmektedir. Bu problemi, kullanılan modelleme aracına tam olarak entegre olmuş SKA araçlarının geliştirilmesi veya modelleme araçlarının kendi statik analiz alt yapılarını oluşturması giderecektir.

Kaynak kodu bulunmayan üçüncü parti kütüphaneleri gibi yazılım parçaları SKA ile incelenememektedir. Bu durumlarda araçlar bir takım varsayımlarda bulunmakla beraber, genellikle çok gerçekçi sonuçlar üretilememektedir. Bu soruna karşı nesne kodunun incelenmesi ve kayıp yazılım parçasının özelliklerini gösteren modeller kullanılması önerilmektedir [12].

SKA ile daha çok programlama hataları tespit edilebilmekte, fakat fonksiyonel hatalar genellikle bulunamamaktadır. Fonksiyonel hataların ise yazılım testleri sırasında tespit edilmesi beklenmektedir. Ayrıca dinamik testlerde karşılaşılan hataların etkileri hemen anlaşılırken, statik analizlerde tespit edilen hataların çalışma zamanı etkilerini kestirmek kolay olmayabilir.

Analiz sonuçlarının farklı SKA araçları tarafından farklı biçimlerde raporlandığı görülmektedir. Bazı araçların zayıf, bazı araçların ise karmaşık raporlama özellikleri bulunmaktadır. Ayrıca bir aracın tüm konularda üstün olmaması ve bu nedenle birkaç aracın birlikte kullanılma gerekliliği, analiz raporlarının farklı biçimlerde elde edilmesine neden olmaktadır. [20]'de belirtilen çalışmada geliştirilen yardımcı araç sayesinde bu soruna çözüm getirildiği ve farklı araçların çıktılarının ortak bir biçimde sunulduğu belirtilmektedir. Yine bu araç ile SKA araçlarından

[1] http://www.dwheeler.com/flawfinder/, son geçerlilik tarihi yoktur.

farklı olarak, incelenen yazılımın kalite modelinin oluşturulabileceği ve yalnızca bu modele uygun çıktılar alınabileceği vurgulanmaktadır.

7. SKA Uygulamasının Sınırlamaları

Bu çalışmada gerçekleştirilen SKA uygulamasının gömülü bir atış kontrol yazılımı özelinde olması ve sınırlı sayıda araç kullanılması, elde edilen sonuçların genelleştirilebilirliğini etkilemektedir. Ayrıca incelenen yazılımın tek başına çalışan bir atış kontrol sistemine ait olması nedeniyle güvenlik zaafları ekseninde detaylı analizler yapılmamıştır. Buna karşın incelenen yazılım kaynak kodunun önemli ölçüde bir boyuta sahip olması ve farklı özelliklere sahip araçlar ile analiz edilmesi sebebiyle, elde edilen sonuçların önemli veriler sunduğu değerlendirilmektedir.

8. Sonuç

Bu bildiride öncelikle statik kod analizi ve incelediği konular açıklanmış, sonrasında ise statik kod analizinin yazılım geliştirme sürecindeki yeri ve sağladığı kazanımların yazılım kalitesine etkisi vurgulanmıştır. Bu kapsamda, ASELSAN SST grubu bünyesinde geliştirilen bir atış kontrol yazılımı çeşitli statik kod analiz araçlarıyla incelenerek elde edilen sonuçlar sunulmuş ve bu sonuçlar ışığında merak edilen sorulara cevap aranmıştır.

Statik kod analizinin kod gözden geçirmeleri ve testler öncesinde uygulanmasının önemli faydalar sağladığı görülmüştür. SKA ile gerekli yazılım iyileştirmelerinin yapılması erken safhalarda mümkün olurken, gözden geçirmelerde harcanan süre de azaltılmaktadır. Olası hataların ise giderilme maliyeti üssel olarak artmadan ortadan kaldırılabileceği tecrübe edilmiştir. Bu sayede test maliyetlerinin azaldığı belirlenirken, testlerde karşılaşılmayan SKA hataların giderilmesi ile bakım maliyetlerinin de önemli ölçüde azalacağı düşünülmektedir. Ayrıca SKA sonuçlarının yorumlanmasıyla yazılımın hata yoğunluğunun önceden kestirilebileceği ve gerekli kalite düzeltme faaliyetlerinin geç kalınmadan uygulanabileceği değerlendirilmektedir. Yazılımın diğer kalite özelliklerinin de SKA ile çıkarılarak olumsuzlukların giderilebileceği görülmüştür. Belirtilen şekillerde yazılımın iyileştirilmesi ve hataların ortadan kaldırılması ile çok daha kaliteli yazılımlar geliştirilebilecektir.

Bu çalışma çerçevesinde gerçekleştirilen uygulamada SKA'nın testlerde çıkabilecek hataların bir kısmını tespit etmesine karşın daha çok atama ve denetim gibi programlama anormalliklerini tespit ettiği ve bu nedenle işlevselliği sınayan yazılım testlerine alternatif olarak düşünülmemesi gerektiği görülmüştür. Testler öncesinde tamamlayıcı olarak uygulanacak SKA ile daha güvenilir ve kaliteli yazılımların dağıtılacağı değerlendirilmektedir. Ayrıca SKA araçlarının birbirini tamamlayıcı şekilde kullanılmasıyla daha etkili sonuçlar elde edileceği görülmektedir. Bununla beraber SKA araçlarının kullanımında bazı sınırlamaların olduğu da tespit edilmiştir. Bu sınırlamaların ilerleyen dönemlerde araç teknolojilerinin de gelişmesiyle birlikte ortadan kalkacağı ve böylece SKA araçları kullanımının geliştiriciler arasında giderek yaygınlaşacağı düşünülmektedir.

9. Kaynakça

[1] Park, R. E., Goethert, W. B. ve Florac, W. A., *Goal-Driven Software Meaurement – A Guidebook*, CMU/SEI-96-HB-002, Ağustos 1996

[2] Everett, G. D. ve McLeod, R., *Software Testing: Testing Across the Entire Software Development Life Cycle*, IEEE Pres, 2007

[3] Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl , J. P. ve Vouk, M. A., "On the Value of Static Analysis for Fault Detection in Software", *IEEE Transactions on Software Engineering*, 32(4):240-253, Nisan 2006

[4] Louridas, P., "Static Code Analysis", *IEEE Software*, 23(4):58-61, Temmuz-Ağustos 2006

[5] Baca, D., Carlsson, B. ve Lundberg, L., "Evaluating the Cost Reduction of Static Code Analysis for Software Security", *Proceedings of the third ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'08)*, Haziran 2008

[6] German, A., "Software Static Code Analysis Lessons Learnt", *CrossTalk*, Kasım 2003

[7] Ganssle, J., "Learning From Disaster", *Military and Aerospace Programmable Logic Devices Conference (MAPLD 2005)*, Eylül 2005

[8] Jones, C., *Applied Software Measurement*, McGraw-Hill, 2008

[9] Mills, E. E., *Software Metrics*, CMU/SEI, Aralık 1988

[10] Lincke, R., Lundberg, J. ve Löwe, W., "Comparing Software Metrics Tools", *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA'08)*, Temmuz 2008

[11] Moore, J. W. ve Seacord, R. C., "Secure Coding Standards", *CrossTalk*, Mart 2007

[12] Anderson, P., "The Use and Limitations of Static-Analysis Tools to Improve Software Quality", *CrossTalk*, Haziran 2008

[13] Wang, Q., Meng, N., Zhou, Z., Li, J. ve Mei, H., "Towards SOA-based Code Defect Analysis", *IEEE International Symposium on Service-Oriented System Engineering, (SOSE'08)*, Aralık 2008

[14] Olson, K. A. ve Overstreet, C. M., "Enhancing Model Understanding Through Static Analysis", *Virginia Space Grant Consortium Student Research Conference*, Nisan 2006

[15] Chen, Y. F., Nishimoto, M. Y. ve Ramamoorthy, C. V., "The C Information Abstraction System", *IEEE Transactions on Software Engineering*, 16(3):325-334, Mart 1990

[16] Sotirov, A. I., "Automatic Vulnerability Detection using Static Source Code Analysis", *Yüksek Lisans Tezi*, The University of Alabama, 2005

[17] Chandra, P., Chess, B. ve Steven, J., "Putting the Tools to Work: How to Succeed with Source Code Analysis", *The IEEE Security & Privacy*, 4(3):80-83, Mayıs-Haziran 2006

[18] Schilling, W. W., "Modeling the Reliability of Existing Software using Static Analysis", *IEEE International Conference on Electro/information Technology*, Mayıs 2006

[19] Nagappan, N. ve Ball, T., "Static Analysis Tools as Early Indicators of Pre-Release Defect Density", *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, Mayıs 2005

[20] Plösch, R., Gruber, H., Pomberger, G., Saft, M. ve Schiffer, S., "Tool Support for Expert-Centred Code Assesments", *International Conference on Software Testing, Verification and Validation*, 2008