A comparison and contrast of APKTool and Soot for injecting blockchain calls into Android applications

Sean Sanders
University at Buffalo
spsander@buffalo.edu

Lukasz Ziarek
University at Buffalo
lziarek@buffalo.edu

Abstract

The injection of blockchain calls into an Android Application is an emerging and important tool for Android application developers. Blockchain technology provides a way of securely storing sensitive data and distributing that data while providing immutability. This paper will compare two compiler-based tools, APKTool, and the Soot framework and how they can inject blockchain calls into Android applications. A major contribution of this paper is that it compares the APKTool, and the Soot framework compilers for injecting blockchain calls, and the difficulties each tool introduces when implementing the injection of a blockchain call. To the best of our knowledge, the use of the Soot framework and the APKTool have never been used to inject blockchain calls. The reason behind this situation is the complexity of configuring blockchain calls in Android applications. Part of the difficulty is because of the constant changes in the API calls in the Android framework. This presents a challenge because the Soot and APKTool compilers have to be modified to adapt to changes in the Android API.

1. Introduction

The goal of this paper is to demonstrate how to inject blockchain calls into Android applications using compiler tools, such as the APKTool and the Soot framework. These two tools are a special advanced class of compilers that have specialized tools for instrumenting Android apps and Java code and are more powerful than traditional compilers. This paper is part of a larger research effort on trying to ensure that advertisers and application developers for mobile applications do not violate advertising frequency, size, and related spamming activities.

The primary research question is: *What compiler framework is best suited for injecting blockchain calls into Android applications?*

The major contribution of this paper is a description of how modern compilers can be used to inject blockchain calls into Android applications. This is a unique approach to inject blockchain calls into Android applications that has not been done before. The reason many developers and researchers have not incorporated blockchain injection using compilers is because of the vast amount of networking knowledge and blockchain knowledge that is required and the inherent complexity of the compiler frameworks. Definitions of the terminology used in this paper can be found in Section 7.

2. Blockchain and Smart contracts

Satoshi Nakamoto conceptualized blockchain concepts and BitCoin in 2008 [1]. The first BitCoin transaction was when Hanyecz bought a pizza at a current value of about $9,500 on May 22, 2010 [2]. The blockchain is a peer-to-peer network that allows for the sharing of data among a vast number of peers [3].

Smart contracts, such as Ethereum, allow for the secure storing of sensitive data [4]. The word contract refers to a legally binding document. Smart contracts are written rules that allow data to be stored in a structured way. Smart contracts allow data to be stored and viewed by as many people as the smart contract rules allow. Smart contracts use a special code referred to as Solidity Ethereum code. This code uses a set of special syntax and semantics that must be strictly followed based on the Solidity compiler version specified.

The primary reason for using blockchain technology is that it allows for the immutability of data. That means that once the data is in the blockchain, it can't be deleted or modified. The blockchain enables the tracking and logging of transactions that take place on the blockchain. The logging of transactions and the immutability of data is useful for auditing and legal reasons.

We opted to use the Ethereum private blockchain because it is a mature private blockchain technology compared to other blockchain technologies. Ethereum allows developers to create and program smart contracts through the highly developed Remix graphical user interface (see: `http://remix.ethereum.org/`). Ethereum was founded in 2013 by Vitalik Buterin. Vitalik developed Ethereum because of the poor functionality of BitCoin's scripting language.

Ethereum allows developers to create decentralized applications referred to as DApps (data applications). The DApps run on a peer-to-peer network and no single entity has control of the network [5]. The peer-to-peer network allows for increased security because of the consensus protocol. A well-known consensus problem that blockchain solves is known as the Byzantine Generals problem. The Byzantine problem occurs when a messenger running from army to army camp delivering confidential messages from each camp, but there is the possibility of the messenger being replaced by a fake messenger. Ethereum solves this problem through the use of a consensus protocol. The next section discusses how modern compilers function and provides a historical background.

3. Compilers

Compilers are used to transform a high-level language to a low-level language. Byte-code and machine code are examples of low-level languages. Decompilation, takes a low-level language and transforms it to a high-level language. Soot decompiles the Java program by first executing the main method in the main class. Then the resolver is called to fetch a reference to a class source (referred to as a ClassSource). A ClassSource is an interface between the file containing the Java byte-code and Soot. When the resolver has a reference to a class source, it attempts to resolve it. The resolver is used to create the Soot class from the Java Byte-code class. Finally, the Soot class methods are set to an object. The Soot object is used to assist with the creation of the jimple representation of the method.

There are many types of compilers that exist. Soot and the APKTool are a special type of compiler. They can both decompile and re-compile. Both tools take the byte-code or java code and deconstruct it into an intermediate representation and then re-compile to byte-code form after the injection or modification of the intermediate representation.

There are very few articles that discuss the injection attacks of Android applications [6]. One of the papers discusses a mobile attack that leverages a well-known vulnerability of HTML5-based mobile applications. HTML5-based mobile applications are vulnerable because they use Contact, SMS, Barcode, MP3, and other channels and can be leveraged for the mobile application attack. The authors created a tool, *NoInjection*, to mitigate such attacks. The tool has a false-positive rate of 2.30%.

Another closely related article discusses how an attack can be conducted against the in-app billing feature for Android [7]. Their generic attack uses a dynamic Dalvik instrumentation approach they developed to inject arbitrary code into a running process. Specifically, the authors dynamically subvert and modify the Play Store application on the device that has been rooted. The author's approach is to abuse the JNI layer of Dalvik, and to modify the interpreted Dalvik method and to replace it with a corresponding native variant that is provided by the attacker. The authors opted to use the APKTool for deconstructing the Android applications.

There are very few articles that closely relate to the injection of blockchain calls into Android applications and there are none that make a comparison of compiler-based frameworks.

One article, *A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages* [8], compares APKTool, dex2jar, and the Soot framework. The article did a monumental job comparing the two frameworks. But the article is dated because of the numerous updates that have been applied to Android Studio and the phone APIs. For example, they tested on Android 6.0, with an API Level of 23. The latest Android API is 9.0 which is now API level 28. That massive jump in the API device level has included many optimizations and updates.

The most influential compiler article related to our research discusses the role of control flow analysis [9]. Control flow analysis is essential because much of the Soot framework incorporates control flows to analyze Android applications. Control flow analysis is showing the flow of data through the application. Without the knowledge of control flow analysis, it is not possible to incorporate more sophisticated analysis. There is a default control flow analysis that can be used, but it limits you from analyzing more complex applications and reduces the chances of successfully injecting blockchain calls at specific/critical locations in the Android app.

Another interesting compiler research topic is the analysis of Dalvik byte-code using the Soot tool [10]. The article illustrates how to use Soot to analyze Dalvik byte-code. This is important because in the past it was not possible for Soot to analyze any Dalvik byte-code. Dalvik byte-code is important because it includes information about the Android Application. This includes information such as the move instructions and other assembly related operations. The authors also discuss the limitations of Soot in analyzing Dalvik byte-code. This brief article was a seminal contribution to compiler research.

Another interesting Soot compiler article discusses the optimization of byte-code and the integration of intra-procedural and whole program optimization. The paper was the groundwork for the Soot framework tool that we are using [11]. Soot was implemented to assist in the optimization of Java byte-code. The initial intention of this research was to focus on the instrumentation of Java code. Instrumenting or instrumentation refers to the modification and analysis of a programming language through the use of compiler technology. The original experiment for the study consisted of evaluating the Soot framework using 12 large benchmarks, including 8 SPECjvm98 benchmarks running on JDK 1.2 for GNU/Linux. Soot provided an improvement of 8% when the interpreter was run. Soot provided a 21% improvement when using the just in time (JIT) compiler.

3.1. Soot Framework

The Soot framework reads in Java files and Android APK files. When Soot reads in these files, it examines the main class, and then builds an object that references all the main methods in the class. The Soot object constructs the jimple representation. Then it looks for any code that was specified for the injection. Finally, Soot attempts to inject the code and will build the output of either jimple, baf, shimple, or the Android APK file.

The Soot framework has some important concepts that need some discussion. For example, a scene is a critical idea in Soot. A scene manages the Soot classes for the application being analyzed. The scene holds all the Android applications classes associated with the APK that is being analyzed.

Another important concept is the SootClass. The SootClass represents the current class that Soot tool is analyzing. The soot class can contain many methods and each method is referred to as a SootMethod. Another important output format that the Soot tool uses is jimple. Jimple is the simplified java code format that the Soot framework uses for the constructing and deconstruction of Android applications. Each Android application method contains a JimpleBody. The JimpleBody is a body, or the code that is enclosed in the current SootMethod, that is represented in jimple form. Each SootMethod contains many units. Units in this context are code fragments. Figure 1 presents an example of how Soot represents an Android application in jimple form.

As illustrated in Figure 1, it is easy to manipulate Android APK files using the jimple data structures that Soot provides. The Soot framework is a mature and powerful tool, carefully crafted by McGill University. The Soot tool allows the efficient and quick injection of blockchain calls into Android APK's. Other compiler-based tools, such as the APKTool, are not as easy and efficient for blockchain code injection.

Another important feature of the Soot framework is the vast number of analysis functions that are included. Some analysis functions include forwards and backwards flow analysis, flow-through, points-to analysis, template-driven intra-procedural data-flow, and directed call graphs. Forward flow analysis provides information about the future, or new code, path of execution. Backwards flow analysis provides information about the code in terms of what variables will be used and not used.

3.2. Soot Framework Setup

To set up the Soot Framework, Ganache, Android Studio, and Eclipse need to be installed. Ganache is a tool that allows the generation of the private blockchain environment. Ganache allows an effortless way of specifying the blockchain accounts and other settings for the private blockchain environment. Without Ganache, it would be nearly impossible to set up a private Ethereum blockchain environment because a command-line tool called geth would be required to set up a private blockchain. The geth command-line tool is very tricky to set up and to set the correct input parameters.

Android Studio is the integrated development

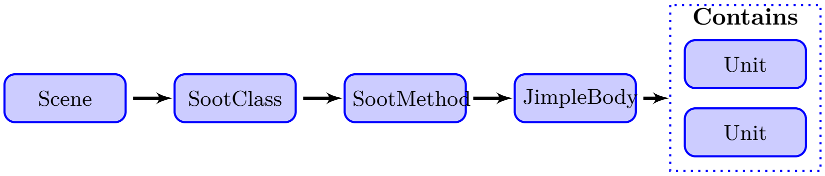


Figure 1. Soot application overview

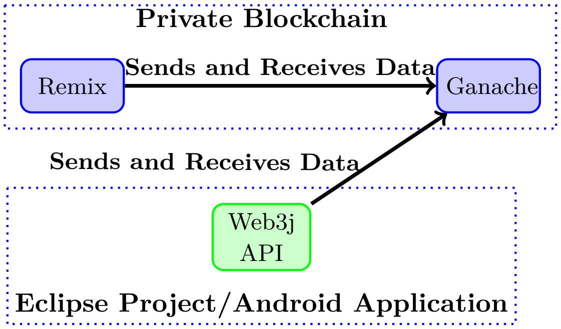


Figure 2. Blockchain interaction overview

environment that allows users to create an Android test application. Android Studio is easy to maintain, test, and generate the final Android application. The other integrated development environment tool is Eclipse that is used for injecting the private blockchain call into the Android application.

3.3. Linking it all together

Remix is Ethereum's graphical user interface that allows developers to program their smart contracts. It also allows them to connect to the private blockchain environment. Ganache is a tool that allows developers to create a private Ethereum blockchain. The fundamental idea of how everything fits together is that the Remix environment connects to Ganache as shown in Figure 2. Remix allows users to program their smart contract through an online graphical user interface and enables them to push their smart contracts to the private blockchain. Ganache is a tool that allows for the creation of a private Ethereum blockchain environment.

Android Studio is used to create the application that will have the API code that will connect and interact with their smart contract on the private blockchain. Then they have the Soot framework in the form of a JAR file that the Eclipse project can include. A JAR file is a prepackaged set of Java files that allow for the creation of a single application that can be executed. The programming of the Soot analysis code injection allows users to inject the blockchain calls into the simple test Android application.

The increasing number of interconnected technologies to use these tools imparts a high level of complexity. In addition, knowledge of computer networking is necessary for implementing Ethereum blockchain calls. Code injection also introduces complexity and there are several troubleshooting techniques necessary to implement the code injection.

Figure 3 illustrates the process of injecting a blockchain call into an Android application. Remix was used to connect to Ganache. Ganache is the private blockchain environment that contains our smart contract. Remix was used to connect to Ganache and send the smart contract to Ganache. Then the Soot/APKTool was utilized to inject the blockchain call into the test application. The Android test application already had the API installed to eliminate some complexity. The Android application when installed and run on the Android phone emulator sends the data to Ganache through the Web3j API. This might appear to be simple, but in reality, it is not and becomes very time consuming.

3.4. APKTool

APKTool is another tool that is used for injecting blockchain calls into the Android test application (Figure 4). It appears to be easier to use than Soot at first, but it still requires a vast amount of knowledge to understand where to inject calls and how the blockchain calls need to be structured. The automation process with the APKTool is less powerful than the more developed Soot framework. For example, you need to manually find an entry point into the Android application. The main class file of the Android application is the entry point. More details are provided in our documentation which will be available at http://www.artbarts.com.

APKTool injects blockchain calls into Android

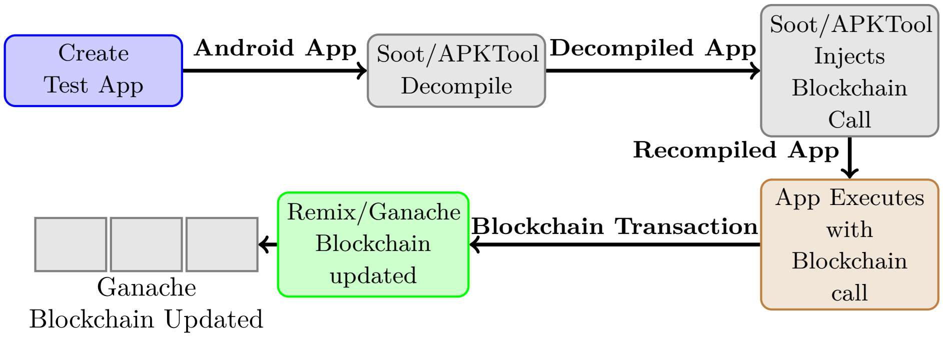


Figure 3. Overview of code injection

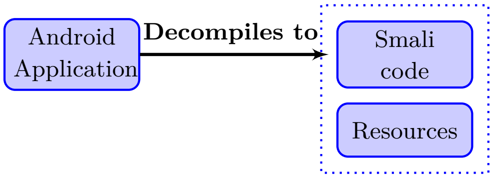


Figure 4. APKTool overview

apps by first decompiling the Android APK into smali code in which you then have to inject your blockchain code manually and then recompile. Smali code is the human-readable format of the Dalvik byte-code and is the assembly language. Smali code uses registers like the x86 assembly language. Individuals that are not familiar with assembly language will have trouble understanding Smali code and how it functions. Individuals who have assembly language knowledge will be comfortable with Smali code. More importantly, you have to be able to manipulate the memory registers that the APKTool uses.

3.5. Code Injection using Soot

When running the Soot framework Ganache should be running and the Ethereum smart contract should be already submitted to the private blockchain. All the instructions for setting up the Soot Framework and APKTool at `https://artbarts.com/` under HICSS 2021 Blockchain Injection Documentation section.

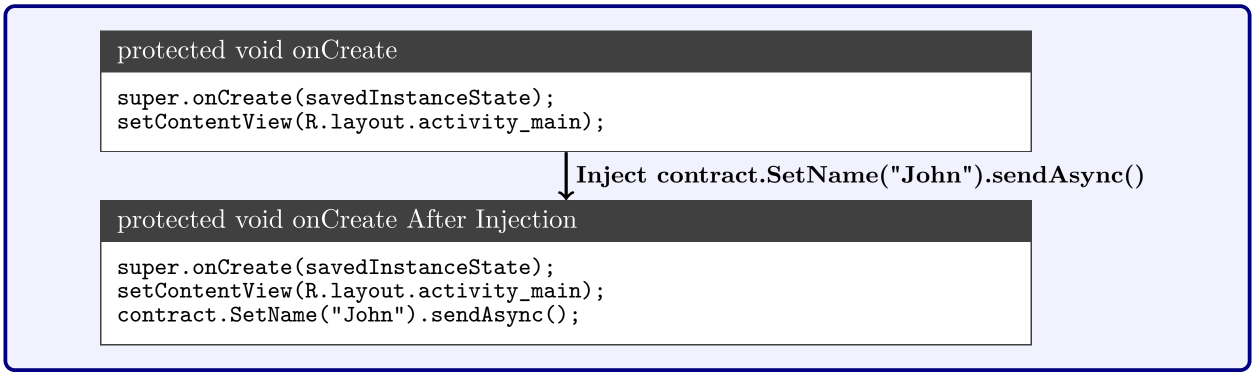
Suppose you want to inject the function contract.SetName("John").sendAsync(), as displayed in Figure 5, into an Android application. The function is used to send the name "John" from the Android application to the Ethereum smart contract. This is necessary in order to inject the function after the setContentView that is located in the onCreate method. The injection statement represented in jimple form consists of three lines of code as displayed in Figure 6.

Figure 7 represents the code that is required in order to inject the function contract.SetName("John").sendAsync() into the Android application. The purpose of the if condition in Figure 7 is to check to see if the Android application has the setContentView function. If the condition is true, then the code is injected. Once finished, Soot will run the injected code and be able to automatically create for APK as described in the documentation.

3.6. Code Injection using APKTool

For the APKTool, there needs to be some thought into where to inject the blockchain code. After an in-depth examination of the Android code it was decided to inject the blockchain code after the setContentView method, which is located in the onCreate method. This requires a manual check for invoke-virtual p0, v0, com/example/simpleapplication/MainActivity;-> setContentView(I);. The next step is to change the locals from 1 to 2 to allow for the use of one extra local for the registers. The next step is to inject the lines that are displayed in Figure 8. The only tricky part of this, is navigating to the entry point in the code where the MainActivity file is located. The process to find the proper entry point in the code is made clear in the documentation contained at `www.artbarts.com`.

Once finished, the APKTool will run and inject the code and will automatically create the Android application package (APK). It is similar to an exe file in Microsoft Windows. This process is described in detail in the documentation.



protected void onCreate

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
```

Inject contract.SetName("John").sendAsync()

protected void onCreate After Injection

```
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
contract.SetName("John").sendAsync();
```

Figure 5. Code to inject

```
$r3 = $r0.<com.example.simpleapplication.MainActivity: com.example.simpleapplication.Hello contract>;
$r2 = virtualinvoke $r3.<com.example.simpleapplication.Hello: org.web3j.protocol.core.RemoteFunctionCall SetName(java.lang.String)>("John");
virtualinvoke $r2.<org.web3j.protocol.core.RemoteFunctionCall: java.util.concurrent.CompletableFuture sendAsync()>();
```

Figure 6. Jimple code

```
if(Check_If_SetContentView_Exists(StringLastKnownUnit)) {
                    //ADD LOCALS
                LocalGenerator l = new LocalGenerator(body);
                Local lgContractReference = l.generateLocal(RefType.v("com.example.simpleapplication.Hello"));
                Local lgRemoteFunctionCallReference = l.generateLocal(RefType.v("org.web3j.protocol.core.RemoteFunctionCall"));

                // Create $r3 = $r0.<com.example.simpleapplication.MainActivity: com.example.simpleapplication.Hello contract>;
                LinkedList<Value> ContractInitArgs = new LinkedList<>();
                AssignStmt AssignStmtContract = Jimple.v().newAssignStmt(Return_Local(body, "Hello"),Jimple.v().newVirtualInvokeExpr(Return_Local(body,"MainActivity"), Scene.v().getMethod("<"+ EntryPointApp +": "+ EntryPointBlockchain +" ContractInit()>").makeRef(), ContractInitArgs));
                units.insertAfter(AssignStmtContract, LastKnownUnit);
                // Create $r2 = virtualinvoke $r3.<com.example.simpleapplication.Hello: org.web3j.protocol.core.RemoteFunctionCall SetName(java.lang.String)>("John");
                LinkedList<Value> ContractPersonName = new LinkedList<>();
                ContractPersonName.add(StringConstant.v("John"));
                AssignStmt AssignStmtContractFunc = Jimple.v().newAssignStmt(Return_Local(body, "RemoteFunctionCall"),Jimple.v().newVirtualInvokeExpr(Return_Local(body,"Hello"), Scene.v().getMethod("<"+ EntryPointBlockchain +": "+ RemoteFunctionCallWeb3J +" SetName(java.lang.String)>").makeRef(), ContractPersonName));
                units.insertAfter(AssignStmtContractFunc, AssignStmtContract);
                // Create virtualinvoke $r2.<org.web3j.protocol.core.RemoteFunctionCall: java.util.concurrent.CompletableFuture sendAsync()>();
                List<String> BlockchainArgumentsForSendAsync = new LinkedList<>();
                SootMethodRef SendAsyncMethodRef = makeMethodRef("org.web3j.protocol.core.RemoteCall","sendAsync", "java.util.concurrent.CompletableFuture",BlockchainArgumentsForSendAsync, false);
                VirtualInvokeExpr InvokeExprSendAsync = Jimple.v().newVirtualInvokeExpr(Return_Local(body,"RemoteFunctionCall"), SendAsyncMethodRef, Collections.<Value>emptyList());
                InvokeStmt InvokeStatementSendAsync = Jimple.v().newInvokeStmt(InvokeExprSendAsync);
                units.insertAfter(InvokeStatementSendAsync, (Unit) AssignStmtContractFunc);
}
```

Figure 7. Code to include for Soot

```
.line 23
iget-object v0, p0, Lcom/example/simpleapplication/MainActivity;
↪  ->contract:Lcom/example/simpleapplication/Hello;

const-string v1, "John"

invoke-virtual {v0, v1}, Lcom/example/simpleapplication/Hello;
↪  ->SetName(Ljava/lang/String;)Lorg/web3j/protocol/core/RemoteFunctionCall;

move-result-object v0

invoke-virtual {v0}, Lorg/web3j/protocol/core/RemoteFunctionCall;
↪  ->sendAsync()Ljava/util/concurrent/CompletableFuture;
```

Figure 8. Code to include for APKTool

4. Comparing the Compilers for Blockchain Code Injection

The Soot framework does a much better job allowing users to inject blockchain APIs' and other APIs' into Android applications. For example, the APKTool does not have the ability to determine the application's entry points. Whereas, Soot automatically determines the Android applications entry point for the injected code. If a developer has an assembly language background and the code to be injected is simple, they may be a good candidate for using the APKTool. Below is a table representing a comparison of the APKTool and the Soot framework.

Features	Soot	APKTool
Automated Analysis and Injection of blockchain calls	Yes	No
Language Output	Jimple, Shimple, and Baf	Smali
Can define main class for Android APK	Yes	No
Uses assembly like language	No	Yes
Can generate APK as output	Yes	Yes
Has poor documentation	No	Yes
Better suited for blockchain injection	Yes	No

Table 1. Soot and APKTool comparison

As illustrated in Table 1, Soot appears to be the best option for injecting blockchain calls in Android applications. This is especially true, if the Android application is complex.

There are instances where the APKTool might be advantageous. For example, if the developer has a decent understanding of assembly language the APKTool might be preferable because the implementation of the injections could take less time. Unfortunately, the Soot framework can be quite complex which leads to the difficulty of understanding how to use the tool effectively. For example, injecting a blockchain call in Soot could take at least six lines of code in order to create the injection. And the Soot code would be very complex. However, the APKTool might take only three lines of Smali code. If the developer understands Smali code, the injection could be much simpler. This does not mean though that the Smali lines of code for blockchain are easy to create. Smali code can be quite complex when using more sophisticated blockchain calls with numerous parameter inputs.

5. Future Work

In the future it would be useful to look at whether the Soot framework or the APKTool does a better job at optimizing the run-time of the code that has been injected into an Android application. Phone emulators are often used to test the functionality of Android code.

Another future research topic is to investigate how to inject blockchain calls into more sophisticated applications involving numerous diverse APIs. For example, an application that utilizes a weather data API might require the developer to inject private blockchain code into a specific location other than the entry point of the app. In many instances, injection needs to occur outside of the main application entry point. Another potential area of investigation would be to identify strategies to automate the APKTool injection of blockchain calls directly into an Android app similar to the Soot framework. This would be very labor intensive, but could provide insight into how to optimize Android code.

Finally, an investigation is needed on how to inject an entire Ethereum API library into an Android app. This would eliminate the developer for having to include the Ethereum API library in their applications. In our paper we only injected the blockchain call of our smart contract into the Android application, but we did not include the library calls that allow the Android app to connect to the private blockchain. This means that instead of injecting a smart contract blockchain call, we would need to include the code for connecting to the private blockchain network. This would require more sophisticated knowledge of writing syntactically correct Smali code. In addition, it would require more complex code and a greater level of detail, compared to the high-level view discussed in this paper.

6. Conclusion

This paper illustrated how to inject a blockchain smart contract call into an Android application using the Soot framework and APKTool. Using these two tools can be a useful mechanism for developers who want to track their applications using a private blockchain. Injecting a blockchain call into an already packaged Android application is powerful and useful because now companies can leverage this technology to track and monitor applications for security and auditing purposes.

The limiting factor of using the APKTool is the degree to which the developer understands assembly language. The APKTool is more useful and easier to understand if you understand assembly language. Otherwise, the learning curve for the APKTool is much steeper than the Soot Framework.

The Soot framework is useful when the developer has to create complex analysis or wants to do an analysis on multiple Android apps at once. This would be more daunting with the APKTool because the tool requires the decompilation of the Android application. Then the insertion of the required assembly code is required and finally the code has to be re-compiled back to an APK. The Soot framework eliminates this step by decompiling and re-compiling the Android app.

One interesting aspect of this research is that we have essentially created malware to inject private blockchain smart contract calls into Android applications. The application developer could theoretically create malware to track their Android applications and to send immutable data to the private blockchain. This is a double-edge sword because hackers could maliciously use this knowledge to inject malicious code into Android applications without the consent of the user or Android app developer.

We are currently working on the enforcement and mitigation of bad actors misusing Android advertisement libraries. This has been a complex task because of the complexities of integrating various software technologies and the timing of the blockchain smart contract calls. For example, it becomes problematic when injecting a blockchain smart contract call after another blockchain smart contract call has been executed. The simultaneous calls end up not registering to the private blockchain correctly because of the mining requirement time necessary to register the data.

References

[1] B. Marr, “A Very Brief History Of Blockchain Technology Everyone Should Read.” Library Catalog: www.forbes.com Section: Tech.

[2] “10 Years On, Laszlo Hanyecz Has No Regrets About His $45M Bitcoin Pizzas,” May 2020. Library Catalog: www.coindesk.com.

[3] L. Mearian, “What is blockchain? The complete guide,” Jan. 2019. Library Catalog: www.computerworld.com.

[4] “What is Ethereum? | history, roadmap, usage, team, mining | Messari.” Library Catalog: messari.io.

[5] D. . M. Hussey and D. . M. Hussey, “What Are Dapps? | The Beginner’s Guide,” Jan. 2019. Library Catalog: decrypt.co Section: Learn.

[6] X. Jin, X. Hu, K. Ying, W. Du, H. Yin, and G. N. Peri, “Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation,” in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS ’14*, (Scottsdale, Arizona, USA), pp. 66–77, ACM Press, 2014.

[7] C. Mulliner, W. Robertson, and E. Kirda, “VirtualSwindle: an automated attack against in-app billing on android,” in *Proceedings of the 9th ACM symposium on Information, computer and communications security*, ASIA CCS ’14, (Kyoto, Japan), pp. 459–470, Association for Computing Machinery, June 2014.

[8] Y. L. Arnatovich, L. Wang, N. M. Ngo, and C. Soh, “A Comparison of Android Reverse Engineering Tools via Program Behaviors Validation Based on Intermediate Languages Transformation,” *IEEE Access*, vol. 6, pp. 12382–12394, 2018. Conference Name: IEEE Access.

[9] O. Shivers, “Control flow analysis in scheme,” in *Proceedings of the ACM SIGPLAN 1988 conference on Programming language design and implementation*, PLDI ’88, (Atlanta, Georgia, USA), pp. 164–174, Association for Computing Machinery, June 1988.

[10] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, “Dexpler: converting Android Dalvik bytecode to Jimple for static analysis with Soot,” in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, SOAP ’12, (Beijing, China), pp. 27–38, Association for Computing Machinery, June 2012.

[11] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot: a Java bytecode optimization framework,” in *CASCON First Decade High Impact Papers*, CASCON ’10, (Toronto, Ontario, Canada), pp. 214–224, IBM Corp., Nov. 2010.

7. Appendix Terminology

Term	**Definition**
compiler	A program that translates statements written in a source programming language and into machine language, object code or assembly.
decompiler	A program that translates machine language, object code or assembly into a high level language such Java.
bytecode	A low-level representation of program code that has been compiled. It can closely resemble assembly language.
APK	The Android Package Kit is used to distribute and for the subsequent execution of an Android application. It is similar to the exe format in Microsoft Windows.
code injection	The process of injecting statements into an application at a specific location without disturbing the flow of the application code.
soot	A compiler framework that is able to decompile and compile Java code with the capability of analysing and instrumenting Java code.
instrumentation	Refers to the modification and analysis of a programming language through the use of compiler technology.
jimple	An intermediate representation of Java code that Soot generates as output.
APKTool	A compiler framework that is able to simply decompile and compile Java code.
smali	An intermediate representation of Java code that APKTool generates as output.
blockchain	A peer-to-peer network that allows for the sharing of data among a vast number of peers [3]. All data stored on the blockchain is immutable.
Ethereum blockchain	A blockchain environment that allows the use of smart contracts.
smart contract	A contract with written rules and terms allowing for controlling the storage, sharing, and modification of data.
Ganache	A tool used for creating an Ethereum blockchain environment.
solidity	A smart contract object-oriented programming language that was developed by Ethereum.
Remix	Ethereum's tool that helps developers program smart contracts. It enables smart contract developers to connect and push smart contracts to the Ethereum blockchain.
DApps	This refers to the decentralized, resilient, transparent, and incentivized applications that reside on blockchain infrastructures. These applications are supposedly less prone to errors.
Backward flow analysis	Provides information about the future code along the path of execution.
Forward flow analysis	Provides information about past code along the path of execution.