KAPITEL 2

Ethereum-Grundlagen

In diesem Kapitel starten wir mit der Erkundung von Ethereum und lernen, wie man Wallets nutzt, Transaktionen erzeugt und einfache Smart Contracts durchführt.

Ether-Währungseinheiten

Ethereums Währungseinheit heißt *Ether* und wird auch mit *ETH* bezeichnet oder mit den Symbolen Ξ (dem griechischen Buchstaben »Xi«, der wie ein stilisiertes großes E aussieht) bzw. (weniger oft) ♦: 1 Ether oder 1 ETH oder Ξ1 oder ♦1.

Verwenden Sie das Unicode-Zeichen `U+039E` für Ξ und `U+2666` für ♦.

Ether wird in kleinere Einheiten weiter unterteilt. Die kleinstmögliche Einheit wird *Wei* genannt. Ein Ether entspricht einer Trillion Wei ($1 * 10^{18}$ oder 1.000.000.000.000.000.000). Manche nennen die Währung auch »Ethereum«, doch das ist ein typischer Anfängerfehler. Ethereum ist das System, Ether die Währung.

Der Ether-Wert wird Ethereum-intern immer als vorzeichenloser ganzzahliger Wert in Wei angegeben. Wenn Sie einen Ether überweisen, codiert die Transaktion den Wert als 1000000000000000000 Wei.

Ethers unterschiedliche Stückelungen besitzen sowohl einen *wissenschaftlichen Namen* nach dem internationalen Einheitensystem (*SI*) als auch einen *umgangssprachlichen Namen* als Hommage an viele große Geister der Informatik und Kryptografie.

Tabelle 2-1 zeigt die verschiedenen Stückelungen, ihre umgangssprachlichen sowie ihre SI-Namen. Die Tabelle hält sich an die interne Darstellung von Werten, d. h., alle Stückelungen werden in Wei (erste Spalte) und Ether als 10^{18} Wei in der siebten Zeile angegeben.

Tabelle 2-1: Ether-Stückelungen und -Namen

Wert (in Wei)	Exponent	Umgangssprachlicher Name	SI-Name
1	1	Wei	Wei
1.000	10^3	Babbage	Kilowei oder Femtoether
1.000.000	10^6	Lovelace	Megawei oder Picoether
1.000.000.000	10^9	Shannon	Gigawei oder Nanoether
1.000.000.000.000	10^{12}	Szabo	Mikroether oder Mikro
1.000.000.000.000.000	10^{15}	Finney	Milliether oder Milli
1.000.000.000.000.000.000	*10^{18}*	*Ether*	*Ether*
1.000.000.000.000.000.000.000	10^{21}	Grand	Kiloether
1.000.000.000.000.000.000.000.000	10^{24}		Megaether

Eine Ethereum-Wallet wählen

Der Begriff *Wallet* wird mittlerweile für viele Dinge verwendet, auch wenn sie einander ähneln und es im täglichen Leben letztlich immer auf das Gleiche hinausläuft. Wir verwenden »Wallet« für eine Softwareanwendung, die Ihnen dabei hilft, Ihren Ethereum-Account zu verwalten. Kurz gesagt, ist eine Ethereum-Wallet Ihr Tor zum Ethereum-System. Sie hält Ihre Schlüssel vor und kann für Sie Transaktionen erzeugen und übertragen. Die Wahl einer Ethereum-Wallet kann schwierig sein, da es eine große Auswahl mit unterschiedlichen Features und Designs gibt. Einige eignen sich eher für Einsteiger, während sich andere an Experten wenden. Die Ethereum-Plattform selbst wird ständig verbessert, und die »besten« Wallets sind häufig diejenigen, die die Änderungen berücksichtigen, die mit Plattform-Updates einhergehen.

Doch keine Sorge! Wenn Sie eine Wallet wählen und ihre Funktionsweise nicht mögen oder wenn Sie sie zuerst mögen und später etwas anderes ausprobieren wollen, können Sie die Wallet recht einfach wechseln. Dazu müssen Sie Ihr Guthaben nur über eine Transaktion von der alten an die neue Wallet senden oder Ihre privaten Schlüssel exportieren und in die neue Wallet importieren.

Wir haben drei unterschiedliche Wallet-Typen ausgewählt, die wir als Beispiele im gesamten Buch verwenden: eine mobile Wallet, eine Desktop-Wallet und eine webbasierte Wallet. Wir haben diese drei Wallets gewählt, weil sie eine große Bandbreite an Komplexität und Features abdecken. Allerdings ist die Wahl dieser Wallets keine Garantie für deren Qualität oder Sicherheit. Sie sind einfach ein guter Ausgangspunkt für Demonstrationen und Tests.

Denken Sie daran, dass eine Wallet-Anwendung nur funktionieren kann, wenn sie Zugriff auf Ihre privaten Schlüssel hat. Es ist daher besonders wichtig, dass Sie eine Wallet-Anwendung nur von einer Quelle herunterladen und nutzen, der Sie vertrauen. Glücklicherweise ist eine Wallet-Anwendung üblicherweise umso vertrau-

enswürdiger, je beliebter sie ist. Dennoch besteht eine bewährte Praxis darin, nicht »alle Eier in ein Nest zu legen«, also Ihre Ethereum-Accounts auf mehrere Wallets zu verteilen.

Nachfolgend einige gute Wallets für Einsteiger:

MetaMask
: MetaMask ist eine Erweiterung, die in Ihrem Browser (Chrome, Firefox, Opera oder Brave Browser) läuft. Sie ist einfach zu nutzen und für Testzwecke praktisch, da sie sich mit einer Vielzahl von Ethereum-Nodes und Test-Blockchains verbinden kann. MetaMask ist eine webbasierte Wallet.

Jaxx
: Jaxx ist eine Wallet für mehrere Plattformen und Währungen, die auf einer Vielzahl von Betriebssystemen läuft, darunter Android, iOS, Windows, macOS und Linux. Sie ist für Neueinsteiger häufig eine gute Wahl, da sie einfach zu nutzen ist. Je nachdem, wo Sie Jaxx installieren, ist sie entweder eine mobile oder eine Desktop-Wallet.

MyEtherWallet (MEW)
: MyEtherWallet ist eine webbasierte Wallet, die in jedem Browser läuft. Sie verfügt über mehrere anspruchsvolle Features, die wir uns in vielen unserer Beispiele ansehen werden.

Emerald Wallet
: Emerald Wallet wurde für die Arbeit mit der Ethereum-Classic-Blockchain entwickelt, ist aber auch mit anderen Ethereum-basierten Blockchains kompatibel. Die Open-Source-Anwendung läuft auf dem Desktop unter Windows, macOS und Linux. Emerald Wallet kann als Full Node fungieren oder in einem »Light«-Modus arbeiten und die Verbindung mit einem öffentlichen entfernten Node herstellen. Sie enthält auch ein Tool, mit dem alle Operationen über die Kommandozeile abgewickelt werden können.

Wir beginnen mit der Installation von MetaMask auf dem Desktop, wollen vorher aber noch kurz die Kontrolle und Verwaltung von Schlüsseln ansprechen.

Kontrolle und Verantwortung

Offene Blockchains wie Ethereum sind wichtig, weil sie als *dezentralisiertes* System arbeiten. Das beinhaltet vieles, doch ein wesentlicher Punkt ist, dass jeder Ethereum-Nutzer seine eigenen privaten Schlüssel kontrollieren kann – und muss –, da sie den Zugriff auf Guthaben und Smart Contracts kontrollieren. Wir bezeichnen diese Kombination aus Zugriff auf Guthaben und Smart Contracts manchmal als »Account« oder als »Wallet«. Diese Begriffe können in ihrer Funktionalität recht komplex werden, weshalb wir später detaillierter auf sie eingehen. Ein grundlegendes Prinzip ist aber, dass ein privater Schlüssel einem »Account« entspricht. Einige Nutzer überlassen die Kontrolle über ihre privaten Schlüssel einem Treuhänder,

etwa einer Onlinebörse. In diesem Buch zeigen wir Ihnen, wie Sie Ihre privaten Schlüssel selbst kontrollieren und verwalten.

Mit der Kontrolle geht aber auch eine große Verantwortung einher. Wenn Sie Ihre privaten Schlüssel verlieren, verlieren Sie auch den Zugriff auf Ihr Guthaben und Ihre Smart Contracts. Niemand kann Ihnen dabei helfen, den Zugriff wiederzuerlangen – Ihr Guthaben ist für immer gesperrt. Hier einige Tipps, die Ihnen dabei helfen, mit dieser Verantwortung umzugehen:

- Improvisieren Sie nicht bei der Sicherheit. Verwenden Sie erprobte und getestete Standardverfahren.
- Je wichtiger der Account (d. h. je höher das Guthaben oder je wichtiger die Smart Contracts), desto höhere Sicherheitsvorkehrungen sollten getroffen werden.
- Die höchste Sicherheit bietet ein Air-Gap-Gerät, doch dieses Niveau ist nicht für jeden Account nötig.
- Speichern Sie Ihren privaten Schlüssel niemals im Klartext, insbesondere nicht digital. Glücklicherweise zeigen Ihnen die meisten Nutzerschnittstellen den privaten Schlüssel gar nicht erst im Rohformat an.
- Private Schlüssel können in verschlüsselter Form in einer digitalen *keystore*-Datei gespeichert werden. Da sie verschlüsselt sind, benötigen Sie zum Entsperren ein Passwort. Wenn Sie nach einem Passwort gefragt werden, wählen Sie ein starkes aus (d. h. lang und zufällig). Sichern Sie es und teilen Sie es mit niemandem. Wenn Sie keinen Passwort-Manager nutzen, schreiben Sie es auf und bewahren es an einem sicheren und geheimen Ort auf. Um auf Ihren Account zugreifen zu können, benötigen Sie sowohl die *keystore*-Datei als auch das Passwort.
- Speichern Sie keine Passwörter in digitalen Dokumenten, digitalen Fotos, Screenshots, Onlinegeräten, verschlüsselten PDFs und so weiter. Noch mal: Improvisieren Sie nicht bei der Sicherheit. Verwenden Sie einen Passwort-Manager oder Stift und Papier.
- Sollten Sie dazu aufgefordert werden, einen Schlüssel als mnemonische Wortfolge zu sichern, verwenden Sie Stift und Papier für ein physikalisches Backup. Verschieben Sie diese Arbeit nicht »auf später«, Sie vergessen es. Diese Backups können genutzt werden, um den privaten Schlüssel wiederherzustellen, falls die auf Ihrem System gespeicherten Daten verloren gehen oder falls Sie Ihr Passwort vergessen oder verlieren. Allerdings können sie auch von Angreifern genutzt werden, um Ihre privaten Schlüssel zu stehlen, weshalb Sie sie niemals digital speichern dürfen und eine physikalische Kopie in einer abgeschlossenen Schublade oder einem Safe vorhalten sollten.
- Bevor Sie große Mengen (insbesondere an neue Adressen) senden, sollten Sie zuerst eine kleine Testtransaktion vornehmen (z. B. kleiner als ein Dollar) und auf eine Empfangsbestätigung warten.

- Wenn Sie einen neuen Account anlegen, senden Sie zuerst eine kleine Testtransaktion an die neue Adresse. Sobald Sie die Testtransaktion empfangen, senden Sie etwas von diesem Account zurück. Das Anlegen von Accounts kann aus verschiedenen Gründen schiefgehen, und wenn etwas schiefgegangen ist, findet man das besser mit einem kleinen Verlust heraus. Wenn die Tests funktionieren, ist alles gut.
- Öffentliche Block-Explorer stellen eine einfache Möglichkeit dar, herauszufinden, ob eine Transaktion vom Netzwerk akzeptiert wurde. Allerdings wirkt sich diese Bequemlichkeit negativ auf Ihre Privatsphäre aus, da Sie Ihre Adressen den Block-Explorern offenlegen, die Sie verfolgen können.
- Senden Sie kein Geld an die in diesem Buch abgedruckten Adressen. Die privaten Schlüssel sind in diesem Buch aufgeführt, und jemand wird das Geld sofort abgreifen.

Nachdem wir die grundlegenden bewährten Praktiken für die Verwaltung und die Sicherheit von Schlüsseln diskutiert haben, wollen wir uns der Arbeit mit MetaMask zuwenden!

Einführung in MetaMask

Öffnen Sie den Google-Chrome-Browser und wechseln Sie zu *https://chrome.google.com/webstore/category/extensions*.

Suchen Sie nach »MetaMask« und klicken Sie das Logo des Fuchses an. Das Ergebnis sollte in etwa aussehen wie das in Abbildung 2-1.

Abbildung 2-1: Die Detailseite der MetaMask-Chrome-Erweiterung

Manchmal gelingt es jemandem, die Google-Filter mit bösartigen Erweiterungen zu unterwandern, also achten Sie darauf, wirklich die MetaMask-Erweiterung herunterzuladen. Die echte

- zeigt die ID `nkbihfbeogaeaoehlefnkodbefgpgknn` in der Adressleiste an,
- wird von *https://metamask.io* angeboten,
- hat mehr als 1.400 Bewertungen und
- hat mehr als 100.000.000 Nutzer.

Sobald Sie sicher sind, die richtige Erweiterung vor sich zu haben, klicken Sie auf *Add to Chrome*, um sie zu installieren.

Eine Wallet anlegen

Sobald MetaMask installiert ist, sollten Sie ein neues Icon (den Kopf eines Fuchses) in der Werkzeugleiste Ihres Browsers sehen. Klicken Sie es an, um die Erweiterung zu starten. Sie werden zuerst dazu aufgefordert, die allgemeinen Geschäftsbedingungen zu akzeptieren, und dann, eine neue Ethereum-Wallet anzulegen, indem Sie ein Passwort eingeben (siehe Abbildung 2-2).

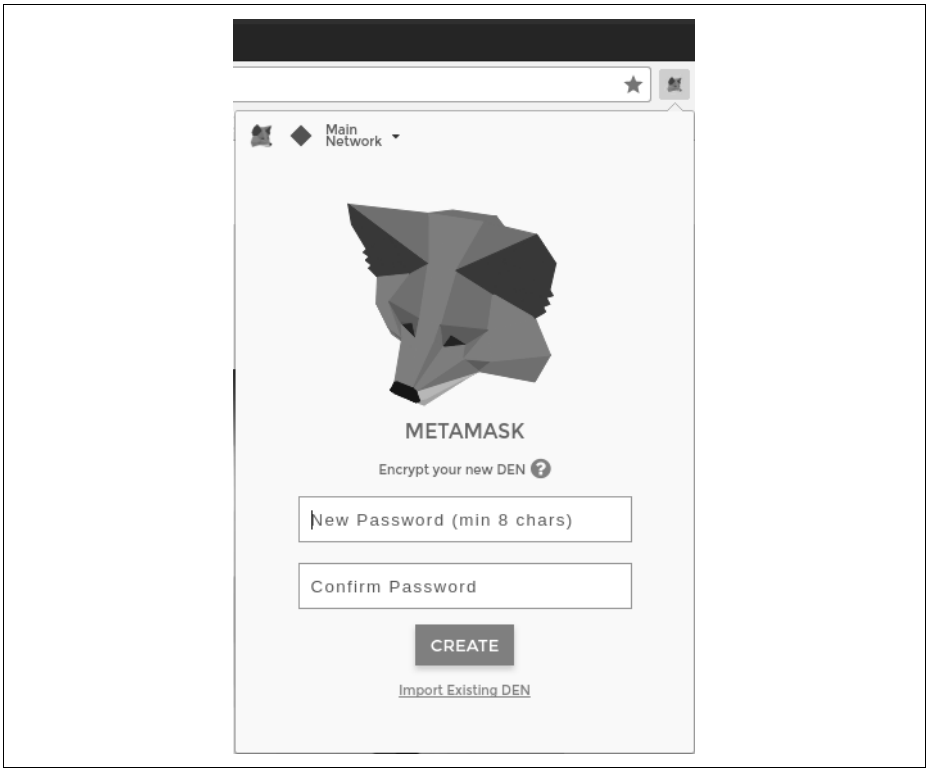

Abbildung 2-2: Die Passwortseite der MetaMask-Chrome-Erweiterung

Das Passwort kontrolliert den Zugriff auf MetaMask, sodass es niemand nutzen kann, der keinen Zugang zu Ihrem Browser hat.

Sobald Sie ein Passwort festgelegt haben, erzeugt MetaMask eine Wallet für Sie und gibt ein *mnemonisches Backup* aus, das aus zwölf Wörtern besteht (siehe Abbildung 2-3). Diese Wörter können in jeder kompatiblen Wallet verwendet werden, um den Zugriff auf Ihr Guthaben wiederherzustellen, sollte auf Ihrem Computer etwas mit MetaMask passieren. Das Passwort benötigen Sie zur Wiederherstellung nicht. Die zwölf Wörter reichen aus.

Sichern Sie Ihr Mnemonic (zweimal) auf Papier. Legen Sie die beiden Papiersicherungen an zwei verschiedenen sicheren Stellen ab, etwa in einem feuerfesten Safe, einer verschließbaren Schublade oder in einem Schließfach. Betrachten Sie die Papier-Backups als gleichwertig mit dem, was Sie in Ihrer Ethereum-Wallet vorhalten. Jeder, der auf diese Wörter zugreifen kann, kann auf die Wallet zugreifen und Ihnen Ihr Geld stehlen.

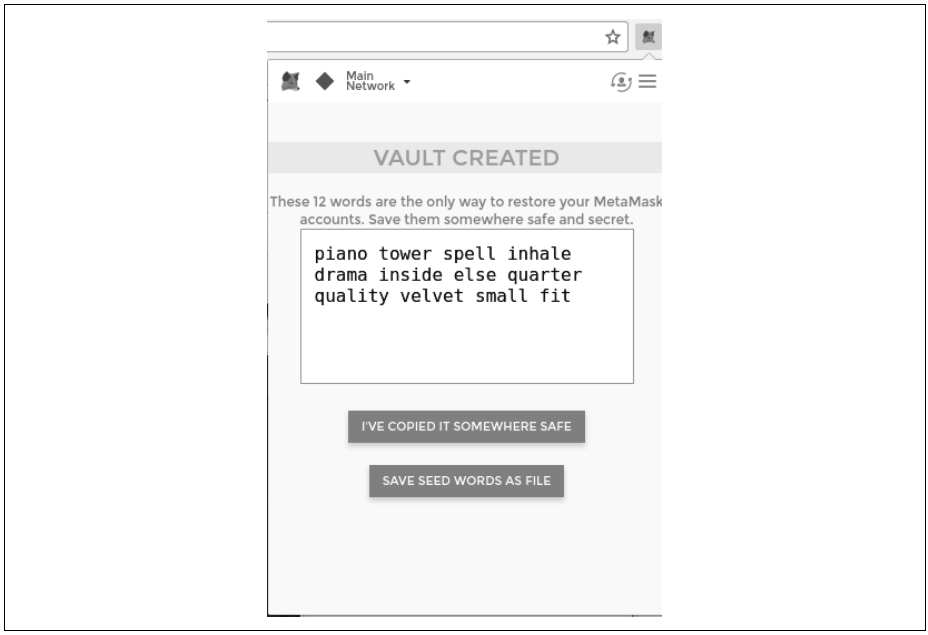

Abbildung 2-3: Von MetaMask erzeugtes mnemomisches Backup Ihrer Wallet

Sobald Sie die sichere Verwahrung Ihres Mnemonic bestätigt haben, können Sie sich die Details Ihres Ethereum-Account ansehen (siehe Abbildung 2-4).

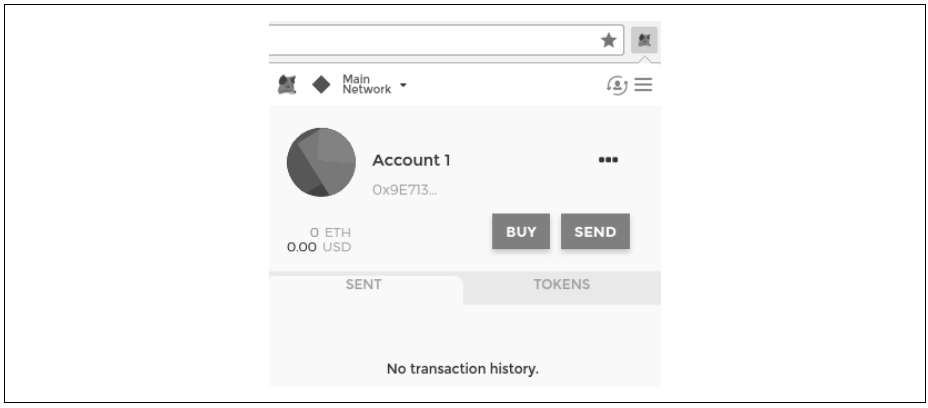

Abbildung 2-4: Ihr Ethereum-Account in MetaMask

Ihre Account-Seite zeigt den Namen Ihres Account (standardmäßig *Account 1*), eine Ethereum-Adresse (in diesem Beispiel `0x9E713...`) sowie ein farbiges Icon, anhand dessen Sie diesen Account von anderen Accounts unterscheiden können. Am Anfang der Account-Seite sehen Sie, mit welchem Ethereum-Netzwerk Sie gerade arbeiten (in unserem Beispiel das *Main Network*).

Glückwunsch! Sie haben gerade Ihre erste Ethereum-Wallet eingerichtet.

Netzwerke wechseln

Wie Sie auf der MetaMask-Account-Seite sehen, können Sie zwischen mehreren Ethereum-Netzwerken wählen. Standardmäßig versucht MetaMask, die Verbindung mit dem Hauptnetzwerk herzustellen. Andere Möglichkeiten sind öffentliche Testnetzwerke, ein Ethereum-Node Ihrer Wahl oder Nodes, die private Blockchains auf Ihrem eigenen Computer ausführen (localhost):

Ethereum-Hauptnetzwerk
: Das öffentliche Ethereum-Hauptnetzwerk. Echte ETH, echte Werte, echte Konsequenzen.

Ropsten-Testnetzwerk
: Ethereums öffentliche Test-Blockchain und Testnetzwerk. ETH in diesem Netzwerk haben keinen Wert.

Kovan-Testnetzwerk
: Öffentliche Ethereum-Test-Blockchain und Netzwerk. Verwendet das Aura-Konsensprotokoll mit Proof of Authority (föderatives Signieren). ETH in diesem Netzwerk haben keinen Wert. Das Kovan-Testnetzwerk wird nur von Parity unterstützt. Andere Ethereum-Clients verwenden das erst später vorgeschlagene Clique-Konsensprotokoll für die Proof-of-Authority-basierte Verifikation.

Rinkeby-Testnetzwerk
: Öffentliche Ethereum-Test-Blockchain und Netzwerk. Verwendet das Clique-Konsensprotokoll mit Proof of Authority (föderatives Signieren). ETH in diesem Netzwerk haben keinen Wert.

Localhost 8545
: Stellt die Verbindung mit einem Node her, der auf demselben Computer läuft wie der Browser. Der Node kann Teil einer beliebigen öffentlichen Blockchain (Haupt- oder Testnetzwerk) oder eines privaten Testnetzwerks sein.

RPC
: Erlaubt die Verbindung von MetaMask mit jedem Node, der eine Geth-kompatible RPC-Schnittstelle (*Remote Procedure Call*) besitzt. Der Node kann Teil einer beliebigen öffentlichen oder privaten Blockchain sein.

Ihre MetaMask-Wallet verwendet für alle Netzwerke, mit denen sie sich verbindet, den gleichen privaten Schlüssel und die gleiche Ethereum-Adresse. Allerdings wird das Guthaben Ihrer Ethereum-Adresse für jedes Ethereum-Netzwerk unterschiedlich sein. Ihre Schlüssel können beispielsweise die Ether und Kontrakte auf Ropsten kontrollieren, nicht aber im Hauptnetzwerk.

Test-Ether beschaffen

Ihre erste Aufgabe besteht darin, sich ein paar Ether für Ihre Wallet zu beschaffen. Sie machen das aber nicht im Hauptnetzwerk, weil echte Ether Geld kosten und die Handhabung etwas mehr Erfahrung verlangt. Daher wollen wir Ihre Wallet zuerst mit ein paar Testnet-Ethern versorgen.

Wechseln Sie mit MetaMask in das *Ropsten-Testnetzwerk*. Klicken Sie auf *Buy* und dann auf *Ropsten Test Faucet*. MetaMask öffnet eine neue Webseite, die in Abbildung 2-5 zu sehen ist.

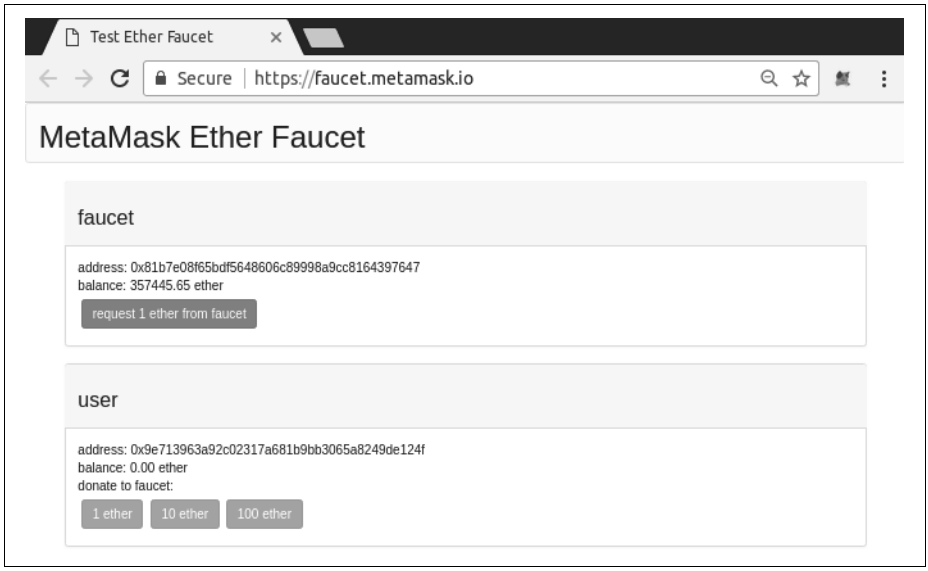

Abbildung 2-5: MetaMask-Ropsten-Test-Faucet

Sie werden bemerken, dass die Webseite bereits die Ethereum-Adresse Ihrer MetaMask-Wallet enthält. MetaMask bindet Ethereum-fähige Webseiten in Ihre MetaMask-Wallet ein und kann Ethereum-Adressen auf diesen Webseiten »sehen«. Auf diese Weise können Sie zum Beispiel eine Zahlung an einen Onlineshop senden, der eine Ethereum-Adresse ausgibt. MetaMask kann in die Webseite auch Ihre eigene Wallet-Adresse als Empfängeradresse eintragen, wenn die Webseite sie anfordert. Auf dieser Seite fragt die Faucet-Anwendung MetaMask nach einer Wallet-Adresse, an die Test-Ether gesendet werden sollen.

Klicken Sie auf den grünen Button *request 1 ether from faucet*. Es erscheint eine Transaktions-ID im unteren Teil der Seite. Die Faucet-App hat eine Transaktion, also eine Zahlung an Sie erzeugt. Die Transaktions-ID sieht wie folgt aus:

```
0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
```

Innerhalb weniger Sekunden wurde die Transaktion von den Minern im Ropsten-Netzwerk geschürft, und Ihre MetaMask-Wallet zeigt ein Guthaben von *1 Ether* an. Klicken Sie auf die Transaktions-ID, und der Browser bringt Sie zu einem *Block-Explorer*, einer Website, auf der Sie Blöcke, Adressen und Transaktionen untersuchen und visualisieren können. MetaMask nutzt den Etherscan-Block-Explorer (*https://etherscan.io/*), einen recht beliebten Ethereum-Block-Explorer. Die Transaktion, in der die Zahlung des Ropsten-Test-Faucet enthalten ist, sehen Sie in Abbildung 2-6.

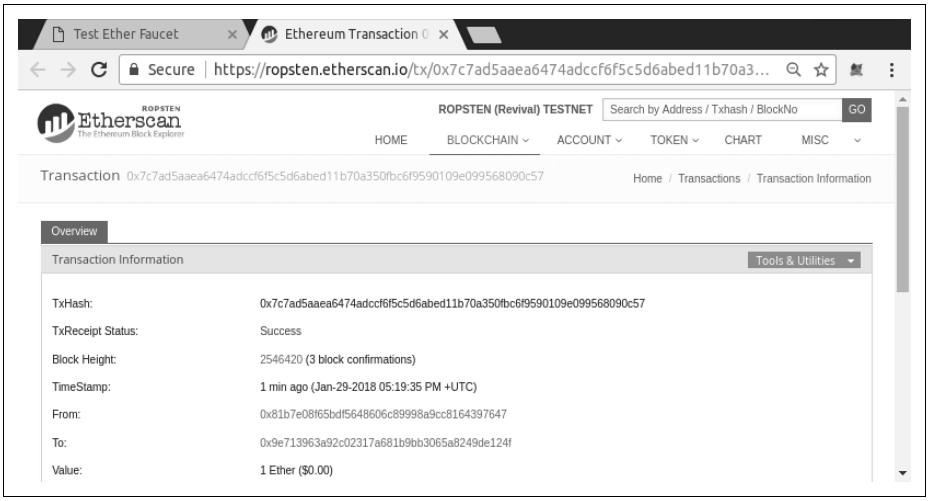

Abbildung 2-6: Etherscan-Ropsten-Block-Explorer

Die Transaktion wurde in der Ropsten-Blockchain festgehalten und kann jederzeit von jedermann eingesehen werden, indem man nach der Transaktions-ID sucht oder den Link besucht (*http://bit.ly/2Q860Wk*).

Wählen Sie den Link oder geben Sie den Transaktions-Hash auf der *ropsten.etherscan.io*-Website ein, um sie sich selbst anzusehen.

Ether aus MetaMask senden

Sobald Sie Ihren ersten Test-Ether vom Ropsten-Test-Faucet erhalten haben, können Sie versuchen, die Ether wieder zurück an das Faucet zu senden. Wie Sie auf der Ropsten-Test-Faucet-Seite sehen können, gibt es die Möglichkeit, 1 ETH an das Faucet zu »spenden« (*donate*). Diese Option ist verfügbar, um Ihre verbliebenen Test-Ether zurückzugeben, sobald Sie mit dem Testen fertig sind, damit auch

andere sie nutzen können. Obwohl Test-Ether keinen Wert haben, werden sie von einigen Leuten gehortet, was es für andere schwierig macht, die Testnetzwerke zu nutzen. Das Horten von Test-Ethern ist verpönt!

Glücklicherweise horten wir keine Test-Ether. Klicken Sie den orangefarbenen *1 ether*-Button an, um eine Transaktion zu erzeugen, die dem Faucet 1 Ether zahlt. MetaMask bereitet die Transaktion vor und öffnet ein Fenster mit der Bestätigung (siehe Abbildung 2-7).

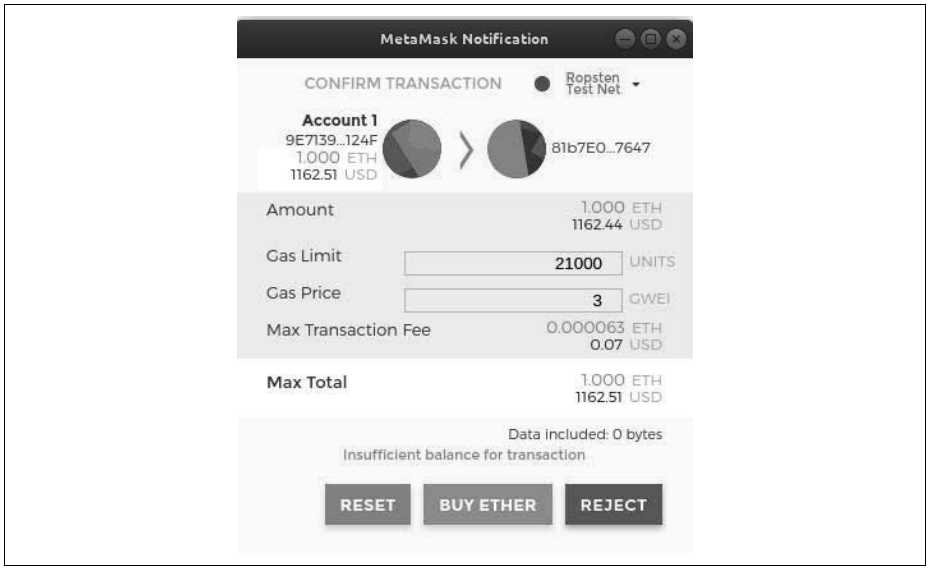

Abbildung 2-7: 1 Ether an das Faucet senden

Ups! Wie Sie sehen, können Sie die Transaktion nicht abschließen, weil MetaMask behauptet, Ihr Guthaben würde nicht ausreichen. Auf den ersten Blick ist das verwirrend: Sie besitzen 1 ETH und wollen 1 ETH senden, also warum behauptet MetaMask, Ihr Guthaben würde nicht ausreichen?

Die Antwort liegt in den Kosten für das Gas. Jede Ethereum-Transaktion verlangt die Zahlung einer Gebühr, die von den Minern für die Validierung der Transaktion erhoben wird. Die Gebühren werden bei Ethereum in einer virtuellen Währung namens *Gas* erhoben. Sie bezahlen dieses Gas als Teil der Transaktion in Ether.

Gebühren werden auch in den Testnetzwerken erhoben. Ohne Gebühren würde sich das Testnetzwerk anders verhalten als das Hauptnetzwerk, und das wäre für eine Testplattform ungeeignet. Gebühren schützen die Testnetzwerke auch vor DoS-Angriffen und schlecht konstruierten Kontrakten (z. B. Endlosschleifen), genau so, wie sie das im Hauptnetzwerk tun.

Wenn Sie eine Transaktion senden wollen, berechnet MetaMask den durchschnittlichen Gaspreis erfolgreicher Transaktionen mit 3 GWei (was für Gigawei steht).

Wei ist die kleinste Stückelung von Ether, wie bereits in »Ether-Währungseinheiten« auf Seite 15 diskutiert. Das Gaslimit ist als Preis einer grundlegenden Transaktion festgelegt, die bei 21.000 Gaseinheiten liegt. Der maximale ETH-Betrag, den Sie zu bezahlen haben, beträgt also 3 * 21.000 GWei = 63.000 GWei = 0,000063 ETH. (Beachten Sie, dass der durchschnittliche Gaspreis variiert, da er hauptsächlich durch die Miner bestimmt wird. In einem späteren Kapitel werden Sie sehen, wie man das Gaslimit erhöhen/reduzieren kann, um sicherzustellen, dass Ihre Transaktion Vorrang genießt, wenn es nötig ist.)

Eine Transaktion von 1 ETH kostet also 1,000063 ETH. MetaMask rundet das verwirrenderweise auf 1 ETH *ab*, wenn es die Gesamtsumme ausgibt, doch der tatsächlich benötigte Betrag ist 1,000063 ETH, und Sie besitzen nur 1 ETH. Klicken Sie auf *Reject*, um diese Transaktion abzubrechen.

Besorgen wir uns also weitere Test-Ether! Klicken Sie wieder auf den grünen Button *request 1 ether from the faucet* und warten Sie ein paar Sekunden. Keine Sorge, das Faucet verfügt über viele Ether und gibt Ihnen weitere, wenn Sie das wünschen.

Sobald Ihr Guthaben 2 ETH beträgt, können Sie es erneut versuchen. Wenn Sie diesmal den orangefarbenen Spenden-Button *1 ether* anklicken, reicht Ihr Guthaben aus, um die Transaktion abzuschließen. Klicken Sie auf *Submit*, wenn MetaMask das Zahlungsfenster öffnet. Nach Abschluss der Transaktion sehen Sie ein Guthaben von 0,999937 ETH, weil Sie 1 ETH an das Faucet gesendet und dafür 0,000063 ETH für Gas bezahlt haben.

Die Transaktionshistorie einer Adresse untersuchen

Sie wissen nun, wie man mit MetaMask Test-Ether sendet und empfängt. Ihr Wallet hat bisher zwei Zahlungen empfangen und eine gesendet. Sie können sich all diese Transaktionen mit dem Block-Explorer *ropsten.etherscan.io* ansehen. Kopieren Sie dazu Ihre Wallet-Adresse und fügen Sie sie in das Suchfeld des Block-Explorers ein, oder Sie lassen die Seite direkt durch MetaMask öffnen. Gleich neben dem Account-Icon sehen Sie bei MetaMask einen Button mit drei Punkten. Klicken Sie ihn an, um sich ein Menü mit Account-bezogenen Optionen anzeigen zu lassen (siehe Abbildung 2-8).

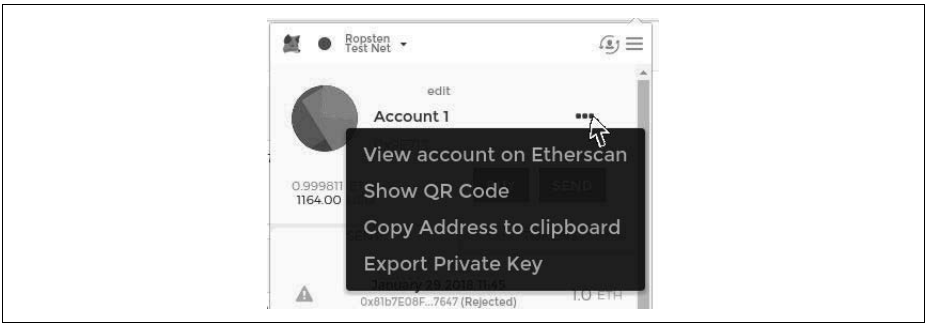

Abbildung 2-8: MetaMask Account-Kontextmenü

Wählen Sie *View account on Etherscan*, um eine Webseite im Block-Explorer zu öffnen, die Ihnen die Transaktionshistorie des Accounts wie in Abbildung 2-9 darstellt.

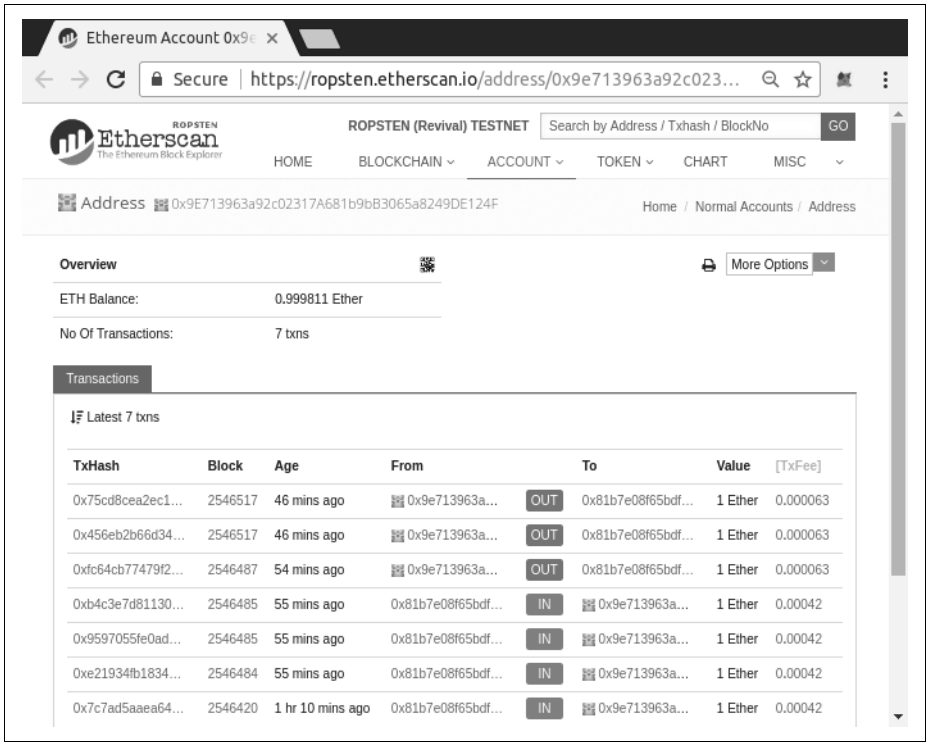

Abbildung 2-9: Adresstransaktionshistorie bei Etherscan

Hier sehen Sie die gesamte Transaktionshistorie Ihrer Ethereum-Adresse. Sie umfasst alle Transaktionen in der Ropsten-Blockchain, bei denen Ihre Adresse als Sender oder Empfänger auftaucht. Klicken Sie die Transaktionen an, um sich weitere Details anzusehen.

Sie können sich die Transaktionshistorie jeder beliebigen Adresse betrachten. Sehen Sie sich beispielsweise die Transaktionshistorie der Ropsten-Test-Faucet-Adresse an (Tipp: die *sender*-Adresse in der ältesten Zahlung an Ihre Adresse). Sie können alle Test-Ether sehen, die vom Faucet an Sie und andere Adressen gesendet wurden. Jede Transaktion, die Sie sich anschauen, führt zu weiteren Adressen und weiteren Transaktionen. Innerhalb kürzester Zeit gehen Sie in diesem Labyrinth vernetzter Daten unter. Öffentliche Blockchains enthalten riesige Mengen an Informationen, die sich programmatisch untersuchen lassen, was zukünftige Beispiele noch zeigen werden.

Der Weltcomputer

Sie haben nun eine Wallet angelegt und Ether gesendet und empfangen. Bisher haben wir Ethereum als Kryptowährung betrachtet, doch Ethereum ist viel, viel mehr. Tatsächlich ist die Kryptowährungsfunktion bei Ethereums Rolle als dezentralisierter Weltcomputer nur von untergeordneter Bedeutung. Ether ist dafür gedacht, für laufende *Smart Contracts* zu bezahlen, also für Computerprogramme, die auf einem emulierten Computer laufen, der als *Ethereum Virtual Machine* (EVM) bezeichnet wird.

Die EVM ist ein globaler Singleton, d.h., er wird als eine globale Instanz eines Computers betrieben, der überall läuft. Jeder Node im Ethereum-Netzwerk betreibt eine lokale Kopie der EVM, um Smart Contracts zu validieren. Die Ethereum-Blockchain hält den sich ändernden *Zustand* dieses Weltcomputers fest, während Transaktionen und Smart Contracts verarbeitet werden. Wir werden das wesentlich detaillierter in Kapitel 13 diskutieren.

Externally Owned Accounts (EOAs) und Kontrakte

Die Art von Account, den Sie in der MetaMask-Wallet angelegt haben, wird *Externally Owned Account*, kurz EOA (deutsch etwa »Account in externem Besitz«) genannt. Solche EOAs sind Accounts, die über einen privaten Schlüssel verfügen. Der Besitz eines privaten Schlüssels bedeutet die Kontrolle über den Zugriff auf Guthaben und Kontrakte. Sie werden sich nun denken können, dass es eine weitere Art Account gibt. Diese andere Account-Art heißt *Kontrakt-Account*. Ein Kontrakt-Account enthält Smart-Contract-Code, den ein einfacher EOA nicht haben kann. Darüber hinaus hat ein Kontrakt-Account keinen privaten Schlüssel. Vielmehr »gehört« er der Logik des Smart-Contract-Codes und wird von dieser auch kontrolliert: dem Softwareprogramm, das in der Ethereum-Blockchain festgehalten wurde, als der Kontrakt-Account erzeugt und durch die EVM ausgeführt wurde.

Kontrakte besitzen Adressen, genau wie EOAs. Kontrakte können auch Ether senden und empfangen, genau wie EOAs. Ist das Ziel einer Transaktion aber eine Kontraktadresse, wird dieser Kontrakt in der EVM *ausgeführt*, wobei die Transaktion und die Transaktionsdaten als Input verwendet werden. Neben Ether können Transaktionen auch Daten enthalten, die angeben, welche spezifische Funktion des Kontrakts ausgeführt werden soll und welche Parameter an diese Funktion zu übergeben sind. Auf diese Weise können Transaktionen Funktionen innerhalb der Kontrakte *aufrufen*.

Beachten Sie, dass ein Kontrakt-Account keinen privaten Schlüssel besitzt und daher keine Transaktion *initiieren* kann. Nur EOAs können Transaktionen initiieren, doch Kontrakte können auf Transaktionen *reagieren*, indem sie andere Kontrakte aufrufen und so komplexe Ausführungspfade aufbauen. Ein typischer

Anwendungsfall ist ein EOA, der eine Anforderungstransaktion an eine Multisignatur-Smart Contract-Wallet schickt, um ein paar ETH an eine andere Adresse zu überweisen. Ein typisches DApp-Programmiermuster besteht darin, das Kontrakt A Kontrakt B aufruft, um einen gemeinsamen Zustand für die Nutzer von Kontrakt A zu erhalten.

In den nächsten Abschnitten werden Sie Ihren ersten Kontrakt schreiben. Sie lernen dann, wie Sie diesen Kontrakt mit Ihrer MetaMask-Wallet und den Test-Ethern aus dem Ropsten-Testnetzwerk anlegen, finanzieren und nutzen.

Ein einfacher Kontrakt: ein Test-Ether-Faucet

Ethereum besitzt viele verschiedene Hochsprachen, die alle zur Entwicklung von Kontrakten verwendet werden können und die alle Bytecode erzeugen. Die prominentesten und interessantesten finden Sie in »Einführung in Ethereum-Hochsprachen« auf Seite 129. Die mit Abstand dominierende Hochsprache zur Entwicklung von Smart Contracts ist aber Solidity. Solidity wurde von Dr. Gavin Wood, dem Mitautor dieses Buchs, entwickelt und ist mittlerweile die am häufigsten genutzte Sprache bei Ethereum (und darüber hinaus). Wir werden Solidity verwenden, um unseren ersten Kontrakt zu schreiben.

Für das erste Beispiel (siehe Beispiel 2-1) entwickeln wir einen Kontrakt, der ein *Faucet* kontrolliert. Sie haben bereits ein Faucet genutzt, um sich Test-Ether im Ropsten-Testnetzwerk zu beschaffen. Ein Faucet ist eine relativ einfache Sache: Es gibt Ether an jede Adresse aus, die danach fragt, und kann regelmäßig wieder aufgefüllt werden. Sie können ein Faucet als Wallet implementieren, das von einem Menschen oder einem Webserver gesteuert wird.

Beispiel 2-1: Faucet.sol: Faucet implementierender Solidity-Kontrakt

```
// Unser erster Kontrakt ist ein Faucet!
contract Faucet {

    // Ether an jeden ausgeben, der darum bittet
    function withdraw(uint withdraw_amount) public {

        // Limit für Abhebungen
        require(withdraw_amount <= 100000000000000000);

        // Betrag an die Adresse senden, die ihn angefordert hat
        msg.sender.transfer(withdraw_amount);
    }

    // Eingehende Zahlungen erlauben
    function () public payable {}

}
```

Sie finden alle Codebeispiele im *code*-Unterverzeichnis des GitHub-Repositories zu diesem Buch (*https://github.com/ethereumbook/ethereumbook/*). Den *Faucet.sol*-Kontrakt finden Sie in: `code/Solidity/Faucet.sol`

Das ist ein sehr einfacher Kontrakt, viel einfacher geht es nicht mehr. Der Kontrakt ist außerdem *mangelhaft* implementiert, da er eine Reihe schlechter Praktiken verwendet und Sicherheitslücken aufweist. Wir werden uns all diese Mängel in späteren Abschnitten ansehen. Doch für den Moment wollen wir uns Zeile für Zeile ansehen, was der Kontrakt macht und wie er funktioniert. Sie werden schnell bemerken, dass viele Solidity-Elemente denen bekannter Programmiersprachen wie JavaScript, Java, oder C++ ähneln.

Die erste Zeile ist ein Kommentar:

```
// Unser erster Kontrakt ist ein Faucet!
```

Kommentare sind für Menschen gedacht und im ausführbaren EVM-Bytecode nicht enthalten. Üblicherweise stehen sie in einer Zeile vor dem Code, den man erklären will, manchmal auch in der gleichen Zeile. Kommentare beginnen mit zwei Schrägstrichen (Slashs): `//`. Alles vom ersten Schrägstrich bis zum Ende der Zeile wird als Leerzeile betrachtet und ignoriert.

Mit der nächsten Zeile beginnt unser eigentlicher Kontrakt:

```
contract Faucet {
```

Diese Zeile deklariert ein `contract`-Objekt. Das ist vergleichbar mit der `class`-Deklaration anderer objektorientierter Sprachen. Die Definition des Kontrakts umfasst alle Zeilen zwischen den geschweiften Klammern ({}), die einen *Geltungsbereich* (Scope) definieren. Viele andere Programmiersprachen verwenden geschweifte Klammern in gleicher Weise.

Als Nächstes deklarieren wir die erste Funktion des `Faucet`-Kontrakts:

```
function withdraw(uint withdraw_amount) public
    {
```

Die Funktion heißt `withdraw` und erwartet als Argument einen vorzeichenlosen Integer-Wert (`uint`) namens `withdraw_amount`. Sie ist als öffentliche Funktion deklariert, d.h., sie kann von anderen Kontrakten aufgerufen werden. Die Funktionsdefinition folgt zwischen geschweiften Klammern. Die erste Zeile der `withdraw`-Funktion legt ein Limit für Abhebungen fest:

```
require(withdraw_amount <=
    100000000000000000);
```

Sie verwendet die fest eingebaute Solidity-Funktion `require`, um zu prüfen, ob der `withdraw_amount` kleiner oder gleich 100.000.000.000.000.000 Wei ist (der Ether-

Basiseinheit, siehe Tabelle 2-1), was 0,1 Ether entspricht. Wird die `withdraw`-Funktion mit einem `withdraw_amount`-Wert über diesem Limit aufgerufen, bricht die `require`-Funktion die Ausführung des Kontrakts ab und schlägt mit einer Ausnahme (*Exception*) fehl. Beachten Sie, dass Anweisungen bei Solidity mit einem Semikolon abgeschlossen werden müssen.

Dieser Teil des Kontrakts bildet die Hauptlogik unseres Kontrakts. Er kontrolliert den Fluss von Mitteln, indem er eine Grenze für Abhebungen festlegt. Das ist ein sehr einfacher Kontrollmechanismus, gibt Ihnen aber eine erste Vorstellung von der Leistungsfähigkeit einer programmierbaren Blockchain: dezentralisierte, Geld kontrollierende Software.

Nun folgt die eigentliche Abhebung:

```
msg.sender.transfer(withdraw_amount);
```

Hier passiert eine Reihe interessanter Dinge. Das `msg`-Objekt ist einer der Inputs, auf den alle Kontrakte zugreifen können. Es repräsentiert die Transaktion, die die Ausführung dieses Kontrakts angestoßen hat. Das Attribut `sender` ist die Senderadresse der Transaktion. Die Funktion `transfer` ist eine fest eingebaute Funktion, die Ether aus dem aktuellen Kontrakt an die Adresse des Senders überträgt. Rückwärts gelesen, bedeutet das, `transferiere` an den `sender` der `msg`, der die Ausführung dieses Kontrakts angestoßen hat. Die `transfer`-Funktion erwartet als einziges Argument einen Betrag. Wir verwenden den `withdraw_amount`-Wert, der als Parameter an die weiter oben deklarierte `withdraw`-Funktion übergeben wurde.

In der nächsten Zeile folgt die schließende geschweifte Klammer, die das Ende der Definition unserer `withdraw`-Funktion anzeigt.

Wir deklarieren nun noch eine weitere Funktion:

```
function () public payable {}
```

Diese Funktion ist eine sogenannte *Fallback-* oder *Standardfunktion*, die aufgerufen wird, wenn die den Kontrakt anstoßende Transaktion keine der im Kontrakt deklarierten Funktionen (oder gar keine Funktion) angegeben hat oder wenn sie keine Daten enthält. Kontrakte können eine solche Standardfunktion (ohne Namen) enthalten. Sie wird üblicherweise zum Empfangen von Ether genutzt. Aus diesem Grund ist sie als öffentlich (`public`) und »zahlbar« (`payable`) definiert, d. h., sie akzeptiert Ether für den Kontrakt. Sie macht nichts anderes, als Ether zu akzeptieren, was die leere Definition in den geschweiften Klammern ({}) anzeigt. Wenn wir ein Transaktion ausführen, die Ether an die Kontraktadresse sendet, als wäre sie eine Wallet, wird sie von dieser Funktion verarbeitet.

Direkt auf unsere Fallback-Funktion folgt die abschließende geschweifte Klammer, die die Definition des `Faucet`-Kontrakts beendet. Das war's!

Den Faucet-Kontrakt kompilieren

Nachdem unser erster Beispielkontrakt fertig ist, brauchen wir einen Solidity-Compiler, um den Solidity-Code in EVM-Bytecode umzuwandeln, der durch eine EVM in der Blockchain ausgeführt werden kann.

Der Solidity-Compiler kommt als eigenständiges Executable als Teil verschiedener Frameworks und gebündelt in integrierten Entwicklungsumgebungen (*Integrated Development Environments*, IDEs). Um die Dinge einfach zu halten, verwenden wir eine der beliebteren IDEs namens *Remix*.

Nutzen Sie Ihren Chrome-Browser (mit der vorhin installierten MetaMask-Wallet) und navigieren Sie zur Remix-IDE auf *https://remix.ethereum.org*.

Wenn Sie Remix das erste Mal laden, öffnet es einen Beispielkontrakt namens *ballot.sol*. Wir brauchen ihn nicht, deshalb wir schließen ihn durch Anklicken des x in der Ecke des Tabs (siehe Abbildung 2-10).

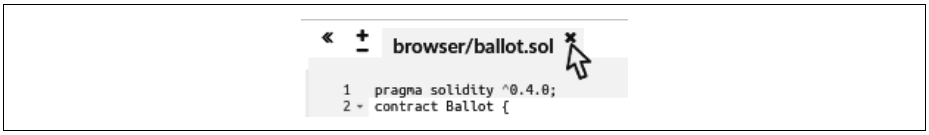

Abbildung 2-10: Den Standard-Beispiel-Tab schließen

Öffnen Sie nun einen neuen Tab, indem Sie das runde Pluszeichen in der oberen rechten Werkzeugleiste anklicken (siehe Abbildung 2-11). Nennen Sie die neue Datei *Faucet.sol*.

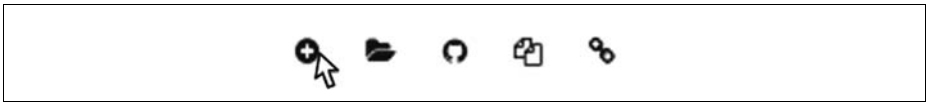

Abbildung 2-11: Klicken Sie das Pluszeichen an, um einen neuen Tab zu öffnen.

Sobald der neue Tab geöffnet ist, kopieren Sie den Beispielcode aus *Faucet.sol* und fügen ihn in den Tab ein (siehe Abbildung 2-12).

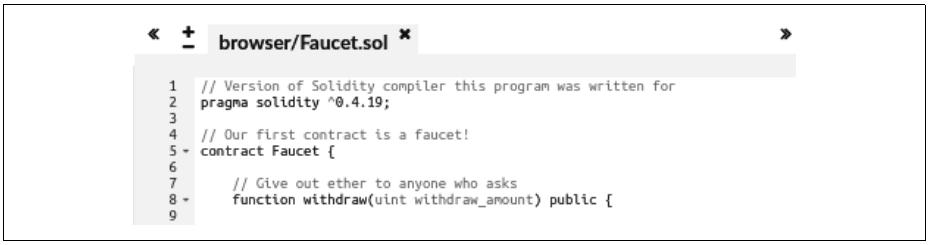

Abbildung 2-12: Faucet-Beispielcode in neuen Tab kopieren

Sobald Sie den *Faucet.sol*-Kontrakt in die Remix-IDE geladen haben, kompiliert die IDE den Code automatisch. Geht alles gut, erscheint rechts unter dem *Compile-*

Tab eine grüne Box namens *Faucet*, was die erfolgreiche Kompilierung bestätigt (siehe Abbildung 2-13).

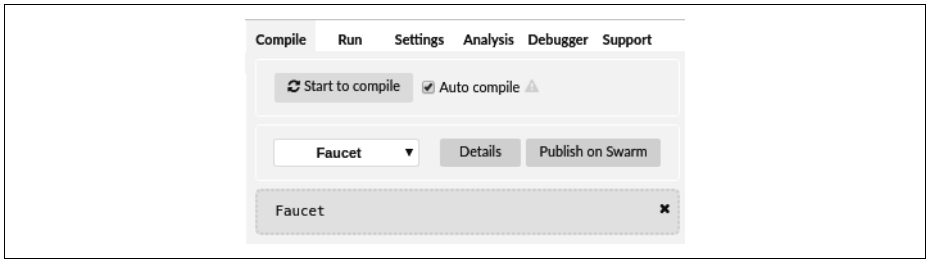

Abbildung 2-13: Remix hat den Faucet.sol-Kontrakt erfolgreich kompiliert.

Geht etwas schief, liegt das üblicherweise daran, dass die Remix-IDE eine von 0.4.19 abweichende Version des Solidity-Compilers nutzt. In diesem Fall verhindert unsere Pragma-Direktive die Kompilierung von *Faucet.sol*. Um die Compilerversion zu ändern, wechseln Sie in den *Settings*-Tab, legen die Version mit 0.4.19 fest und versuchen es erneut.

Der Solidity-Compiler hat die *Faucet.sol* nun in EVM-Bytecode kompiliert. Falls es Sie interessiert, der Bytecode sieht wie folgt aus:

```
PUSH1 0x60 PUSH1 0x40
    MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST
    PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN STOP PUSH1
    0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI PUSH1
    0x0 CALLDATALOAD PUSH29
    0x100000000000000000000000000000000000000000000000000000000 SWAP1 DIV
    PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI JUMPDEST
    STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT
    JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD
    SWAP1 SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8
    0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0
    DUP1 REVERT JUMPDEST CALLER PUSH20
    0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH2 0x8FC DUP3 SWAP1 DUP2
    ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1 DUP4 SUB
    DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP POP ISZERO ISZERO PUSH1 0xB6
    JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6
    0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE
    LOG4 0x5f PUSH24 0x7541F409787592C988A079407FB28B4AD000290000000000
```

Sind Sie glücklich darüber, eine Hochsprache wie Solidity nutzen zu können, statt direkt in EVM-Bytecode programmieren zu müssen? Wir auch!

Den Kontrakt in der Blockchain registrieren

Wir haben jetzt also einen Kontrakt, den wir in Bytecode kompiliert haben. Nun müssen wir den Kontrakt in der Ethereum-Blockchain »registrieren«. Wir werden das Ropsten-Testnetzwerk für unseren Testkontrakt nutzen, weshalb das die Blockchain ist, an die wir ihn senden.

Die Registrierung eines Kontrakts in der Blockchain verlangt die Erzeugung einer speziellen Transaktion, deren Ziel die Adresse `0x0000000000000000000000000000000000000000` ist, die auch *Nulladresse* (Zero Address) genannt wird. Die Nulladresse ist eine spezielle Adresse, die der Ethereum-Blockchain mitteilt, dass Sie einen Kontrakt registrieren wollen. Glücklicherweise übernimmt die Remix-IDE das für Sie und sendet die Transaktion an MetaMask.

Wechseln Sie zuerst in den *Run*-Tab und wählen Sie *Injected Web3* im *Environment*-Drop-down-Menü. Das verbindet die Remix-IDE mit der MetaMask-Wallet und über MetaMask mit dem Ropsten-Testnetzwerk. Sobald das geschehen ist, erscheint Ropsten als Umgebung (unter *Environment*). Im *Account*-Auswahlfeld erscheint außerdem die Adresse Ihrer Wallet (siehe Abbildung 2-14).

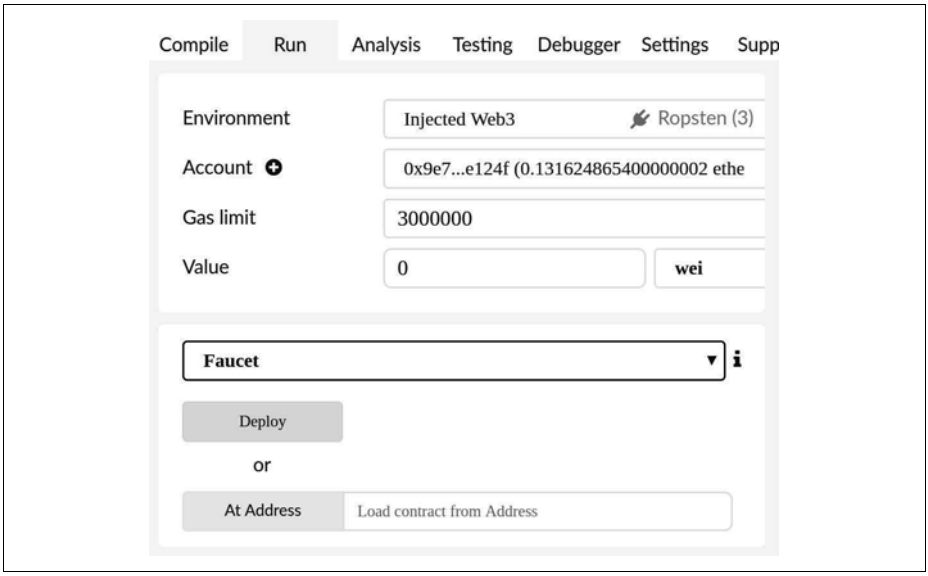

Abbildung 2-14: Run-Tab der Remix-IDE mit ausgewählter Injected-Web3-Umgebung

Rechts unter den *Run*-Einstellungen, die Sie gerade bestätigt haben, ist der `Faucet`-Kontrakt bereit, erzeugt zu werden. Klicken Sie auf den *Deploy*-Button (siehe Abbildung 2-14).

Remix konstruiert eine spezielle »Erzeugungstransaktion«, und MetaMask fordert Sie auf, diese zu bestätigen (siehe Abbildung 2-15). Sie werden bemerken, dass die Kontrakterzeugungstransaktion keine Ether enthält, aber 258 Byte Daten (den kompilierten Kontrakt) umfasst und 10 GWei Gas verbraucht. Klicken Sie auf *Submit*, um ihn zu genehmigen.

Nun müssen Sie warten. Es kann zwischen 15 und 30 Sekunden dauern, bis der Kontrakt auf Ropsten geschürft wurde. Remix scheint nicht viel zu tun, aber gedulden Sie sich.

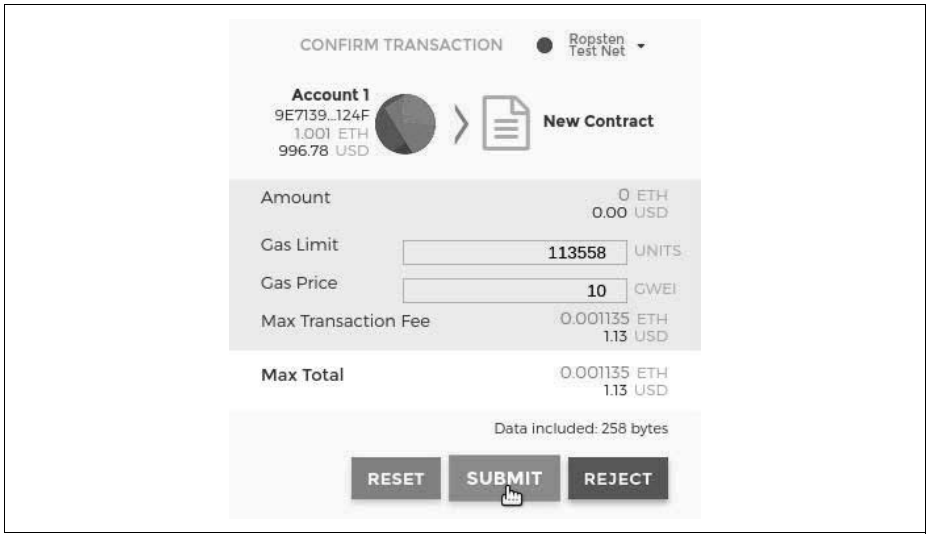

Abbildung 2-15: MetaMask mit Kontrakterzeugungstransaktion

Sobald der Kontrakt erzeugt ist, erscheint er unter dem *Run*-Tab (siehe Abbildung 2-16).

Abbildung 2-16: Der Faucet-Kontrakt LEBT!

Beachten Sie, dass der `Faucet`-Kontrakt nun eine eigene Adresse aufweist: Remix zeigt sie als *Faucet at 0x72e…c7829* (Ihre Adresse, die aus zufälligen Buchstaben und Ziffern besteht, ist eine andere). Das kleine Zwischenablagesymbol zur Rechten erlaubt Ihnen, die Kontraktadresse in Ihre Zwischenablage zu kopieren. Wir nutzen diese Option im nächsten Abschnitt.

Interaktion mit dem Kontrakt

Fassen wir zusammen, was Sie bisher gelernt haben: Ethereum-Kontrakte sind Geld kontrollierende Programme, die in einer als EVM bezeichneten virtuellen Maschine ausgeführt werden. Sie werden durch eine spezielle Transaktion erzeugt, die den Bytecode des Kontrakts in der Blockchain festhält. Sobald sie in der Blockchain eingetragen sind, besitzen sie (genau wie Wallets) eine Ethereum-Adresse.

Wenn jemand eine Transaktion an die Kontraktadresse sendet, wird der Kontrakt (mit der Transaktion als Eingabe) in der EVM ausgeführt. An Kontraktadressen gesendete Transaktionen können Ether, Daten oder beides enthalten. Enthalten sie Ether, werden diese dem Kontraktkonto »gutgeschrieben«. Enthalten sie Daten, können diese eine bekannte Funktion des Kontrakts festlegen, die mit entsprechenden Argumenten aufgerufen wird.

Die Kontraktadresse in einem Block-Explorer ansehen

Wir haben unseren Kontrakt in der Blockchain festgehalten und können sehen, dass er eine Ethereum-Adresse besitzt. Wir wollen uns mit dem Block-Explorer auf *ropsten.etherscan.io* ansehen, wie ein solcher Kontrakt aussieht. In der Remix-IDE kopieren Sie die Adresse des Kontrakts, indem Sie das Zwischenablagesymbol neben seinem Namen anklicken (siehe Abbildung 2-17).

Abbildung 2-17: Kontraktadresse aus Remix kopieren

Lassen Sie Remix offen, wir kehren später dorthin zurück. Wechseln Sie mit Ihrem Browser auf *ropsten.etherscan.io* und fügen Sie die Adresse in das Suchfeld ein. Sie sehen die Historie der Ethereum-Adresse des Kontrakts (siehe Abbildung 2-18).

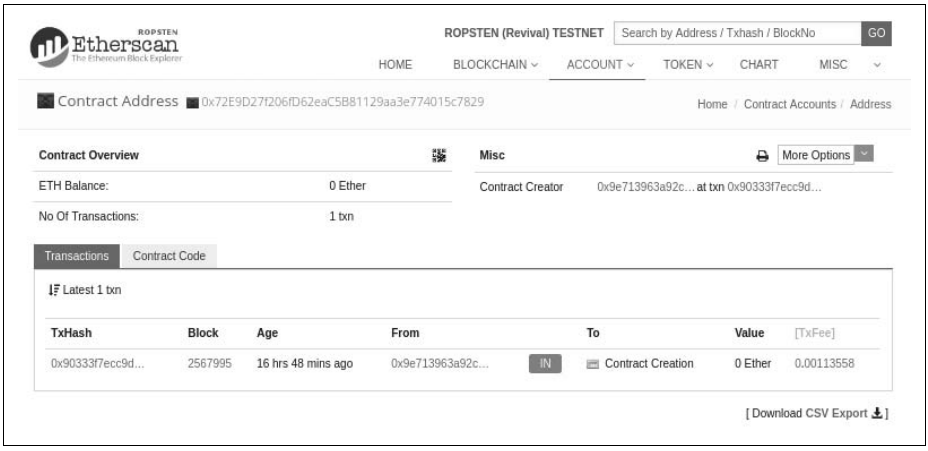

Abbildung 2-18: Ansicht der Faucet-Kontraktadresse im Etherscan-Block-Explorer

Den Kontrakt finanzieren

Bisher enthält der Kontrakt nur eine Transaktion in seiner Historie: die Kontrakterzeugungstransaktion. Wie Sie sehen können, besitzt der Kontrakt auch keine Ether (Kontostand null). Das liegt daran, dass wir in der Erzeugungstransaktion keine Ether an den Kontrakt gesendet haben, auch wenn wir das hätten tun können.

Unser Faucet braucht Geld! Wir werden also zuerst MetaMask nutzen, um Ether an den Kontrakt zu senden. Die Adresse des Kontrakts sollte immer noch in Ihrer Zwischenablage liegen (falls nicht, kopieren Sie sie noch einmal aus Remix). Öffnen Sie MetaMask und senden Sie ihm 1 Ether, genau wie an jede andere Ethereum-Adresse (siehe Abbildung 2-19).

Abbildung 2-19: 1 Ether an die Kontraktadresse senden

Wenn Sie den Etherscan-Block-Explorer nach einer Minute neu laden, sehen Sie eine weitere Transaktion für die Kontraktadresse und ein aktualisiertes Guthaben von einem Ether.

Erinnern Sie sich an die unbenannte, öffentliche, »zahlungsfähige« Standardfunktion in unserem *Faucet.sol*-Code? Sie sah wie folgt aus:

```
function () public payable {}
```

Wenn Sie eine Transaktion an die Kontraktadresse senden, ohne die aufzurufende Funktion festzulegen, wird diese Standardfunktion aufgerufen. Da wir sie als zahlungsfähig (`payable`) deklariert haben, hat sie den Ether akzeptiert und im Konto des Kontrakts festgehalten. Ihre Transaktion hat die Ausführung des Kontrakts in

der EVM angestoßen und dessen Kontostand aktualisiert. Sie haben Ihr Faucet finanziert!

Abhebungen aus unserem Kontrakt

Als Nächstes wollen wir etwas von unserem Faucet abheben. Um sich etwas auszahlen zu lassen, müssen wir ein Transaktion aufbauen, die die `withdraw`-Funktion aufruft und als Argument die abzuhebende Menge (`withdraw_amount`) übergibt. Um die Dinge für den Moment einfach zu halten, lassen wir Remix diese Transaktion für uns erzeugen, und MetaMask wird sie uns zur Bestätigung vorlegen.

Kehren Sie zu Remix zurück und sehen Sie sich den Kontrakt im *Run*-Tab an. Sie sehen eine rote Box namens *withdraw* sowie ein Feld mit `uint256 withdraw_amount` (siehe Abbildung 2-20).

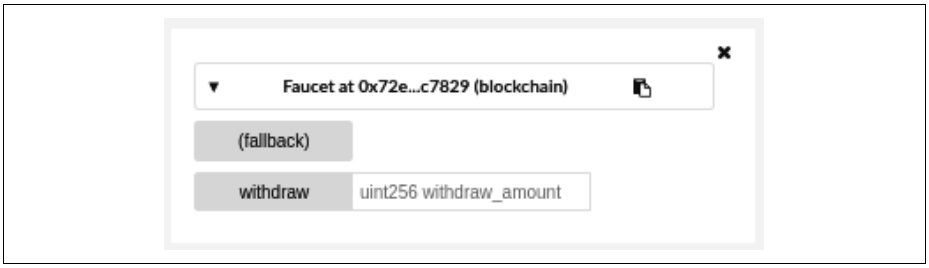

Abbildung 2-20: Die withdraw-Funktion von Faucet.sol in Remix

Das ist die Remix-Schnittstelle des Kontrakts. Sie erlaubt uns den Aufbau von Transaktionen, die die im Kontrakt definierten Funktionen aufrufen. Wir werden einen Wert für `withdraw_amount` eingeben und den *withdraw*-Button anklicken, um eine Transaktion zu erzeugen.

Zuerst wollen wir den Wert für `withdraw_amount` bestimmen. Wir wollen versuchen, 0,1 Ether abzuheben, was der maximalen Menge unseres Kontrakts entspricht. Denken Sie daran, dass bei Ethereum alle Währungseinheiten intern in Wei angegeben werden und unsere `withdraw`-Funktion den Auszahlungsbetrag `withdraw_amount` ebenfalls in Wei erwartet. Die gewünschte Menge von 0,1 Ether entspricht 100.000.000.000.000.000 Wei (einer 1, gefolgt von 17 Nullen).

Durch die Limits von JavaScript kann eine so große Zahl wie 10^17 von Remix nicht verarbeitet werden. Daher verwenden wir Anführungszeichen, damit Remix den Wert als String empfangen und dann als `BigNumber` verarbeiten kann. Lassen wir die Anführungszeichen weg, kann die Remix-IDE den Wert nicht verarbeiten und bricht mit der Fehlermeldung *Error encoding arguments: Error: Assertion failed* ab.

Geben Sie "100000000000000000" (mit Anführungszeichen) in das `withdraw_amount`-Feld ein und klicken Sie den *withdraw*-Button an (siehe Abbildung 2-21).

Abbildung 2-21: Klicken Sie in Remix auf withdraw, um eine Auszahlungstransaktion zu erzeugen.

MetaMask öffnet ein Transaktionsfenster, das Sie bestätigen müssen. Klicken Sie auf *Submit*, um den withdrawal-Aufruf an den Kontrakt zu senden (siehe Abbildung 2-22).

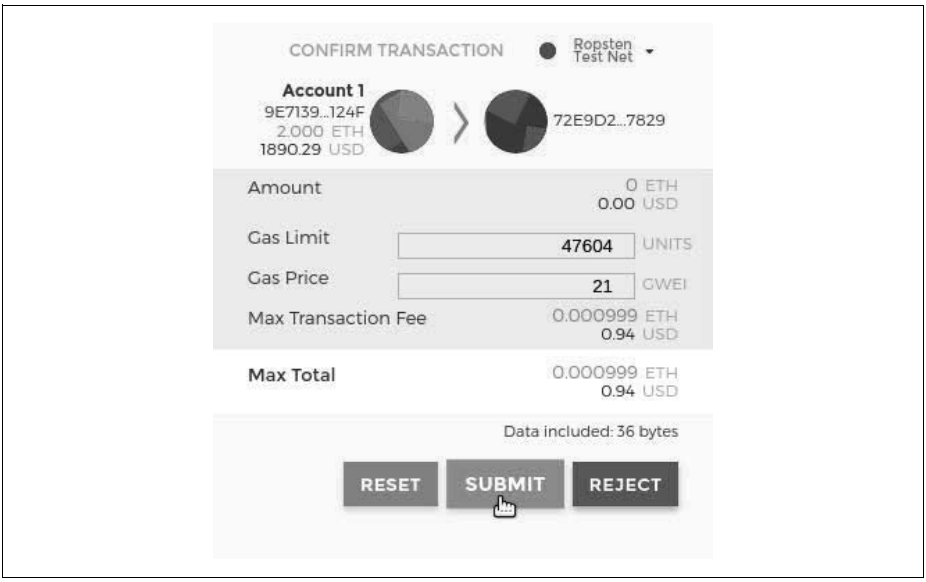

Abbildung 2-22: MetaMask-Transaktion zum Aufruf der withdraw-Funktion

Warten Sie eine Minute und laden Sie dann den Etherscan-Block-Explorer neu, um sich die neue Transaktion in der Historie für die `Faucet`-Kontraktadresse anzusehen (siehe Abbildung 2-23).

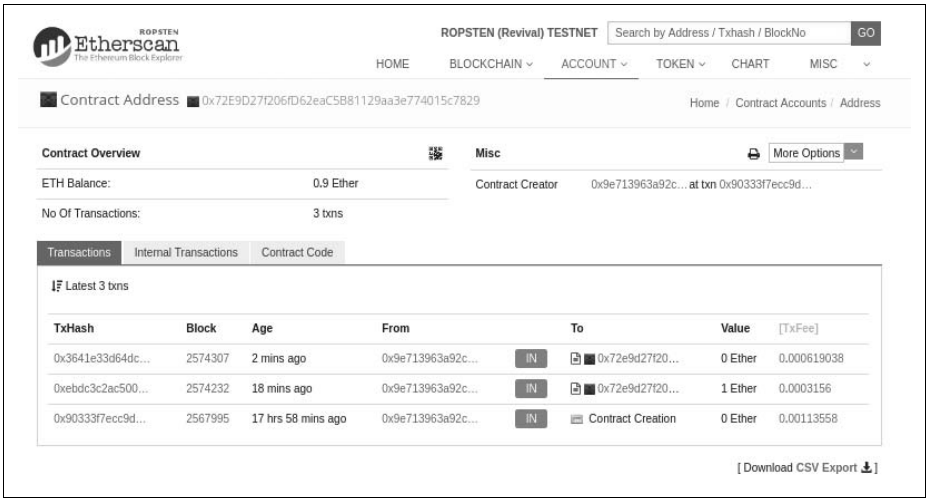

Abbildung 2-23: Etherscan mit der die withdraw-Funktion aufrufenden Transaktion

Wir sehen nun eine neue Transaktion mit der Kontraktadresse als Ziel und einem Wert von 0 Ether. Das Guthaben des Kontrakts liegt nun bei 0,9 Ether, weil uns wie gewünscht 0,1 Ether gesendet wurden. Doch wir sehen keine »OUT«-Transaktion in der Historie der Kontraktadresse.

Wo ist die ausgehende Überweisung hin? Auf der History-Seite der Kontraktadresse ist ein neuer Tab namens *Internal Transactions* erschienen. Da die Überweisung der 0,1 Ether vom Kontraktcode ausging, handelte es sich um eine interne Transaktion, eine sogenannte *Nachricht* (Message). Klicken Sie den Tab an, um sie sich anzusehen (siehe Abbildung 2-24).

Diese »interne Transaktion« wurde durch den Kontrakt in dieser Codezeile (der `withdraw`-Funktion in *Faucet.sol*) gesendet:

```
msg.sender.transfer(withdraw_amount);
```

Fassen wir zusammen: Sie haben eine Transaktion in Ihrer MetaMask-Wallet angestoßen, die Dateninstruktionen enthielt, die die Funktion `withdraw` mit einem `withdraw_amount`-Argument von 0,1 Ether aufgerufen hat. Diese Transaktion hat die Ausführung des Codes innerhalb der EVM ausgelöst. Während die EVM die `withdraw`-Funktion des `Faucet`-Kontrakts ausgeführt hat, wurde zuerst `require` aufgerufen und sichergestellt, dass der angeforderte Betrag kleiner oder gleich der maximal erlaubten Abhebung von 0,1 Ether ist. Dann wurde die `transfer`-Funktion aufgerufen, um die Ether zu senden. Die Ausführung der `transfer`-Funktion hat eine interne Transaktion erzeugt, die 0,1 Ether des Kontraktguthabens an Ihre Wallet-Adresse überwiesen hat. Diese Transaktion erscheint im *Internal Transactions*-Tab in Etherscan.

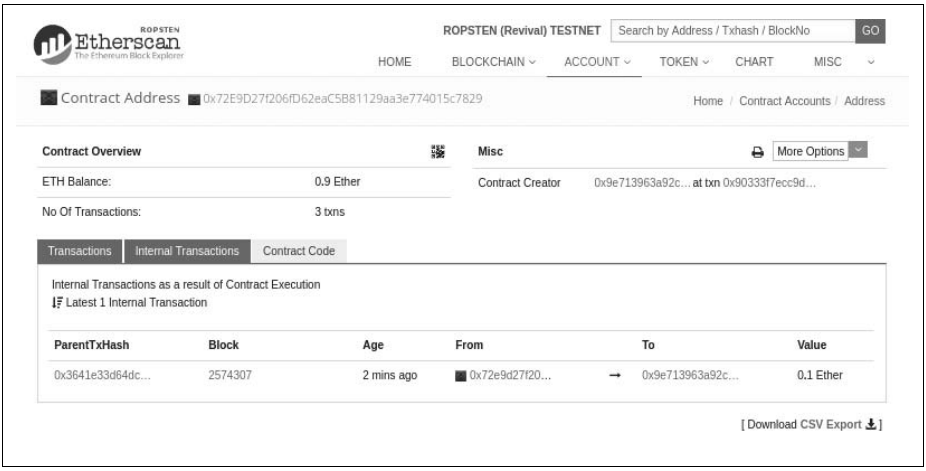

Abbildung 2-24: Etherscan mit interner Transaktion, die Ether vom Kontrakt transferiert

Fazit

In diesem Kapitel haben Sie eine Wallet mit MetaMask eingerichtet und über ein Faucet mit Test-Ether aus dem Ropsten-Testnetzwerk aufgefüllt. Sie haben Ether über die Ethereum-Adresse Ihrer Wallet empfangen und diese dann an die Ethereum-Adresse des Faucets geschickt.

Dann haben Sie einen Faucet-Kontrakt in Solidity entwickelt. Mithilfe der Remix-IDE haben Sie den Kontrakt in EVM-Bytecode kompiliert und dann Remix genutzt, um eine Transaktion zu erzeugen, die den `Faucet`-Kontrakt in der Ropsten-Blockchain registriert hat. Sobald er erzeugt war, verfügte der `Faucet`-Kontrakt über eine Ethereum-Adresse, der Sie Ether gesendet haben. Abschließend haben Sie eine Transaktion konstruiert, die die `withdraw`-Funktion aufrief und erfolgreich 0,1 Ether abgehoben hat. Der Kontrakt hat den Request geprüft und Ihnen 0,1 Ether in einer internen Transaktion überwiesen.

Das hört sich nicht nach viel an, doch Sie haben gerade erfolgreich mit einer Software interagiert, die Geld in einem dezentralisierten Weltcomputer verwaltet.

Wir werden in Kapitel 7 noch viele weitere Smart Contracts programmieren. Bewährte Praktiken und Sicherheitsaspekte lernen Sie in Kapitel 9 kennen.