Bihu Smart Contract Formal Verification

Daejun Park, Yi Zhang, Manasvi Saxena, Iustin Iordache, and Grigore Rosu

Runtime Verification, Inc.

February 28, 2018

Contents

1 Introduction

This report presents a formal verification of Bihu KEY contracts that will be deployed on the Ethereum mainnet in February 2018, upon request from the Bihu (https://bihu.com/) team represented by Dr. Bin Lu (Gulu) and Mr. Huafu Bao.

Bihu is a blockchain-based ID system, and KEY is the utility token for the Bihu ID system and community. The KEY ERC20 token contract had already been deployed in the Ethereum mainnet[1].

The smart contracts requested to be formally verified are the ones for operating the KEY tokens. The Solidity source code of those contracts that are the target of our formal verification is provided here:

`https://github.com/bihu-id/bihu-contracts/tree/f9a7ab65181cc204332e17df30406612d5d350ef/src`

and their informal specification is described here:

`https://docs.google.com/document/d/1-PilHhInQxGod7FZNbtfv2bbgV1045ROT5TO3WLhDOE`

Our formal verification artifact for the Bihu KEY contracts is publicly available at:

`https://github.com/runtimeverification/verified-smart-contracts/tree/master/bihu`

[1] `https://etherscan.io/address/0x4cd988afbad37289baaf53c13e98e2bd46aaea8c#code`

1.1 Formal Verification Methodology

Our methodology for formal verification of smart contracts is as follows. First, we formalize the high-level business logic of the smart contracts, based on a typically informal specification provided by the client, to provide us with a precise and comprehensive specification of the functional correctness properties of the smart contracts. This high-level specification needs to be confirmed by the client, possibly after several rounds of discussions and changes, to ensure that it correctly captures the intended behavior of their contracts. Then we refine the specification all the way down to the Ethereum Virtual Machine (EVM) level, often in multiple steps, to capture the EVM-specific details. The role of the final EVM-level specification is to ensure that nothing unexpected happens at the bytecode level, that is, that only what was specified in the high-level specification will happen when the bytecode is executed. To precisely reason about the EVM bytecode without missing any EVM quirks, we adopted KEVM[2], a complete formal semantics of the EVM, and instantiated the K-framework[3] reachability logic theorem prover[4] to generate a correct-by-construction deductive program verifier for the EVM. We use the verifier to verify the compiled EVM bytecode of the smart contract against its EVM-level specification. Note that the Solidity compiler is *not* part of our trust base, since we directly verify the compiled EVM bytecode. Therefore, our verification result *does not depend on the correctness of the Solidity compiler*.

For more details, resources, and examples, we refer the reader to our Github repository for formal verification of smart contracts, publicly available at:

`https://github.com/runtimeverification/verified-smart-contracts`

1.2 Scope

The target contracts of our formal verification are the following, where we took the Solidity source code from Bihu's Github repository, `https://github.com/bihu-id/bihu-contracts`, commit `f9a7ab65`:

- `KeyRewardPool.sol`
- `WarmWallet.sol`

More specifically, we formally verified the functional correctness of the following two functions:

- `KeyRewardPool.collectToken()`
- `WarmWallet.forwardHotWallet()`

1.3 Resources

We use the K-framework (`kframework.org`) and its verification infrastructure throughout the formal verification effort, as mentioned in Section 1.1. All of the formal specifications are mechanized within the K-framework as well. Therefore, some background knowledge about the K-framework would be necessary for reading and fully understanding the formal specifications and reproducing the mechanized proofs. We refer the reader to existing resources for background knowledge about the K-framework and its verification infrastructure as follows (URL links are embedded in the text):

- K framework
  - Download and install
  - K tutorial
  - K editor support

[2]`https://github.com/kframework/evm-semantics`
[3]`http://www.kframework.org`
[4]`http://fsl.cs.illinois.edu/index.php/Semantics-Based_Program_Verifiers_for_All_Languages`

- KEVM: an executable formal semantics of the EVM in K
  - Jellowpaper: reader-friendly formatting of KEVM
  - KEVM technical report
- K reachability logic prover
  - eDSL: domain-specific language for EVM-level specifications

1.4 Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

The formal verification results presented in this report only show that the target contract behaviors meet the formal (functional) specifications. Moreover, the correctness of the generated formal proofs assumes the correctness of the specifications and their refinement, the correctness of KEVM, the correctness of the K-framework's reachability logic theorem prover, and the correctness of the Z3 SMT solver. The presented result makes no guarantee about properties not specified in the formal specification. Importantly, the presented formal specification considers only the behaviors *within* the EVM, *without* considering the block/transaction level properties or off-chain behaviors, meaning that the verification result does *not* completely rule out the possibility of the contract being vulnerable to existing and/or unknown attacks.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large number of funds.

2 Code Review Analysis

Overall, we found that the code we were asked to formally verify was carefully written and met the intended specification. During the verification effort, we found several minor issues with the code, which we reported to the Bihu developers and they promptly addressed or took under advisement. We describe these briefly below (URL links are embedded in the text 'Line N'):

KeyRewardPool.sol

- We noticed that the parentheses on Line 31 were unnecessary. The Bihu developers have fixed the problem in the commit `a2ffbb5`.
- We suggested moving Line 54 before the assignment `var _key = key;`. By doing that, if the require condition on line 54 is not satisfied, the gas cost for the assignment can be saved. The issue was fixed in the commit `a2ffbb5`.
- We noticed that Line 72 could potentially cause overflow and advised Bihu to use

```
uint canExtract = canExtractThisYear + (total - remainingTokens);
```

The Bihu developers have fixed the problem in the commit `8492fe7`.

- We showed that the check at Line 76 was unnecessary since the condition is always false. The Bihu developers preferred to keep the code, though, as sanity check. We think it is acceptable as long as the gas consumption is not a big concern.
- The transferTokens function, unfortunately, is not formally verifiable as is. That is because there is no way to know if the `_token` parameter is indeed an address holding a conforming ERC20 token. Moreover, this function can be quite dangerous. We recommended increase care when using it, since it can call an arbitrary contract under this contract's permission. Note that the type casting `ERC20(_token)` checks *nothing* at runtime, and the client has to check if the given `_token` contract is indeed a legitimate ERC20 token and its transfer does nothing harmful.

WarmWallet.sol

- We noticed that Line 4 was redundant. Indeed, `ds-auth/auth.sol` is never directly used in `WarmWallet.sol`, but is imported and used by `ds-token/token.sol`. The generated EVM bytecode stays unchanged after removing this import. The issue was fixed in the commit `8492fe7`.
- We found that the mixed use of the type names `uint` and `uint256` (the first is just an alias to the second) on Lines 21 - 22 lowers code readability. There has been a related discussion in the OpenZeppelin community[5]. The issue was fixed in the commit `8492fe7`.
- We noticed that there was no overflow protection on Line 72. We suggested using

  ```
  require((_time - lastWithdrawTime) > 24 hours);
  ```

  The issue was fixed in the commit `8492fe7`. Note that our suggested code will be safe as long as the current time (Unix epoch time in seconds) can be represented within 256 bits, i.e., until the next $\sim 3 \times 10^{69}$ years, even without using the Math library.
- On Lines 70-72, the likelihood that the first `require` throws is much lower than the likelihood that the second one throws. If true, it may be more beneficial to move the second require after the first, in terms of gas consumption.

Protection for Invalid Call Data Although we have not found a specific exploit, we wanted to ensure that the Bihu developers are aware of the tricky behaviors of how the function arguments (especially, the address values) are extracted (decoded) at the EVM level, especially when the call data is not valid.

- **Case 1: The address values consisting of more than 20 bytes**

  While a valid address is 20-byte long, it is encoded as a 32-byte long data, being padded on the higher-order (left) side with zero-bytes such that the length is (a multiple of) 32 bytes.

  For example, an address `0x1234567890123456789012345678901234567890` is encoded as the call data `0x0000000000000000000000001234567890123456789012345678901234567890`.

  When a function is called with this encoded call data, the call data is decoded by simply truncating the first 12 bytes, without checking if the higher 12 bytes are all zeros. This means that if a client (e.g., an exchange) makes a mistake when encoding a transaction, putting non-zero bits in the higher-order side, (e.g., accidentally shifting the bits to the left), then the function silently truncates and takes the remaining bytes as the address input value, instead of throwing an exception (or reverting the execution).

[5] `https://github.com/OpenZeppelin/zeppelin-solidity/issues/226`

For example, if `WarmWallet.setOwner` is called with an invalid address encoding, e.g., `0x00000000000000000000000012345678901234567890123456789012345678900` (shifted left by one), then it sets the owner to `0x2345678901234567890123456789012345678900` truncating the left-most 1, which may be an invalid address whose private key is not known, meaning that the wallet may become nonfunctional (you cannot change `owner`, `withdrawer`, or `withdrawLimit`, and cannot run `pauseStart` or `pauseEnd`).

- **Case 2: The address values consisting of less than 20 bytes**

  The similar silent behavior happens when an address is mistakenly encoded in less than 20 bytes. Most notably, the short address attack[6] exploits such a behavior. Some workarounds are proposed, but they are not fully satisfiable[7].

We recommend the clients (e.g., exchanges) of this contract do the checks for both cases off-chain before issuing transactions.

Compilation and Build System There were many warnings when compiling the original source code with the latest Solidity compiler (version `0.4.18+commit.9cf6e910.Linux.g++`). We suggested using the newer version of the DappSys library to fix all the warnings. The issue was acknowledged by the Bihu developers. However, it turned out that it is not trivial to update the library since Bihu has already deployed the `KEY` token contract using the old version of the DappSys library's `DSToken` contract (commit `bb98ff4`), which depends on other DappSys library contracts that are used by the Bihu contracts.

Another aspect of the old version of `DSToken` to consider is the use of `assert()` instead of `require()` for input validation. Since the Byzantium network upgrade, `require()` is recommended for light-weight validation because it has better gas behavior than `assert()` in case of failure. The latest version of `DSToken` uses `require()` for input validation.

3 Formal Specification and Verification of `KeyRewardPool.collectToken`

In this section, we present a formal specification (and its refinements) of `KeyRewardPool.collectToken`. As explained in Section 1.1, we refine the initial high-level specification into low-level ones in multiple steps. We have the following four (refined) specifications, where each subsequent specification refines the previous one:

1. $\mathsf{collectToken}_{\mathsf{spec}}$: high-level definitional specification
2. $\mathsf{collectToken}_{\mathsf{code}}$: high-level constructive specification
3. $\mathtt{collectToken}_{\mathtt{solidity}}$: Solidity-level functional specification
4. $\mathtt{collectToken}_{\mathtt{evm}}$: EVM-level functional specification

$\mathsf{collectToken}_{\mathsf{spec}}$ is the high-level specification written with the purpose of communication with the client to ensure that it correctly captures the intended behavior of their contract. While $\mathsf{collectToken}_{\mathsf{spec}}$ is rather a mathematical definition, $\mathsf{collectToken}_{\mathsf{code}}$ refines it to make the computation steps explicit, being more constructive. Since $\mathsf{collectToken}_{\mathsf{code}}$ employs simply the real arithmetic for the computation steps, $\mathtt{collectToken}_{\mathtt{solidity}}$ refines it to capture the unsigned integer arithmetic of Solidity (and thus EVM). Finally, $\mathtt{collectToken}_{\mathtt{evm}}$ refines $\mathtt{collectToken}_{\mathtt{solidity}}$ further down to the EVM level to capture EVM-specific details.

Note that the specification refinement is critical for formal verification because of the inherent gap between the (high-level) code written by developers and the (low-level) code that actually runs on the EVM. Moreover, we split the refinement process into multiple (small) steps, which makes it easier to prove the soundness of each refinement step.

[6] `http://vessenes.com/the-erc20-short-address-attack-explained`
`https://blog.coinfabrik.com/smart-contract-short-address-attack-mitigation-failure`
[7] `https://github.com/OpenZeppelin/zeppelin-solidity/issues/261`

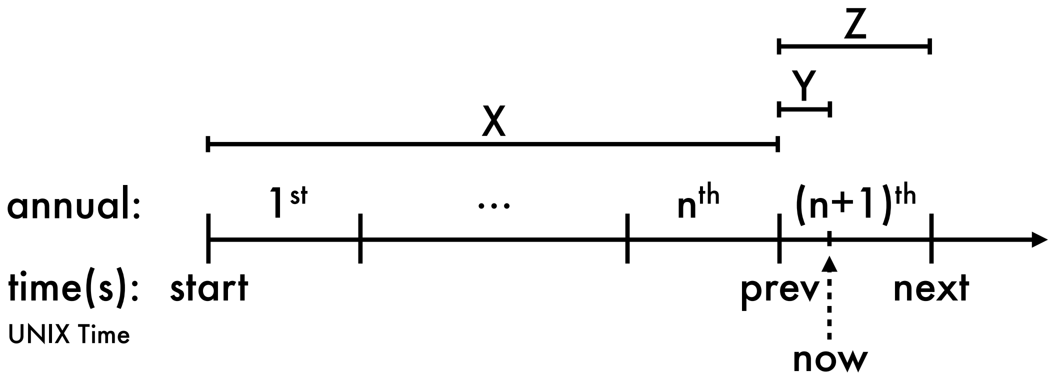

Figure 1: Illustration of important variables of $\mathsf{collectToken_{spec}}$

3.1 $\mathsf{collectToken_{spec}}$: High-Level Definitional Specification

We present $\mathsf{collectToken_{spec}}$, a formal high-level specification of the `collectTokens` function, the main function of the `KeyRewardPool` contract, derived from the following informal English specification provided to us by the client[8].

> "KeyRewardPool contract is responsible for releasing key tokens in the reward pool (initially about 45 billion tokens). The reward pool has a start time. From the start time, every 365 days, we define it as an annual. In each year, a total of 10% of the remaining amount is released. In each year, the token is released linearly."

Figure 1 illustrates the important variables of $\mathsf{collectToken_{spec}}$. We formalize each English sentence, starting from the second one.

> "The reward pool has a start time."

Let start denote the start time as shown in Figure 1.

> "From the start time, every 365 days, we define it as an annual."

Suppose n full-years (i.e., *annuals*) have passed since start, where the time periods of each annual are denoted by $1^{\text{st}}, \cdots, n^{\text{th}}$, and $(n+1)^{\text{th}}$, respectively, in Figure 1. Let now be the time (as seconds since Unix epoch) when the `collectToken` function is called, and suppose now is in the middle of the $(n+1)^{\text{th}}$ annual. Let prev (and next) be the time of the beginning (and the end, respectively) of the current $(n+1)^{\text{th}}$ annual.

> "In each year, a total of 10% of the remaining amount is released. In each year, the token is released linearly."

Let T be the total number of tokens. If T is fixed over the lifetime of the contract, the number of tokens to be released each annual can be formalized as follows. After the first annual has passed, $0.1 \times T$ tokens are released and eligible to be collected; after the second annual, additional $0.1 \times (0.9 \times T)$ tokens are released; after the third annual, yet another additional $0.1 \times (0.9^2 \times T)$ tokens are released. In general, after each n^{th} annual has passed, $0.1 \times (0.9^{n-1} \times T)$ tokens are newly released. Thus, the sum of all of the released tokens up to the end of the n^{th} annual is:

$$0.1 \cdot T + 0.1 \cdot (0.9 \cdot T) + 0.1 \cdot (0.9^2 \cdot T) + \cdots + 0.1 \cdot (0.9^{n-1} \cdot T) \;=\; T(1 - 0.9^n)$$

Let X, Y, and Z be the number of tokens that are supposed to be released for each time period (more precisely, right after the end of the period), respectively, as shown in Figure 1. Let b be the fraction of the current annual up to now, defined as:

$$b = \frac{\mathsf{now} - \mathsf{prev}}{\mathsf{next} - \mathsf{prev}}$$

[8]`https://docs.google.com/document/d/1-PilHhInQxGod7FZNbtfv2bbgV1045ROT5TO3WLhDOE`

Now we have the following equalities:

$$
\begin{aligned}
Y &= Z \times b \\
Z &= T \times 0.9^n \times 0.1 \\
X &= T(1 - 0.9^n) \\
X + Z &= T(1 - 0.9^{n+1})
\end{aligned}
$$

When the `collectToken` function is called, it collects all of the tokens that have been released (but not yet collected) up to that point. Let C be the number of tokens that have already been collected until now since start. The `collectToken` function, when being called at now, collects newly the following amount of tokens:

$$X + Y - C$$

Let C' be the number of the collected tokens after this `collectToken` call. Then we have $C' = X + Y$.

Note that T may change in between calls to the `collectToken` function. We have two cases:

- Case 1. If T has *increased* since the last `collectToken` call, the number of the newly collected tokens (more precisely $X + Y$) will *increase* as it will collect the new percentage of tokens that have not been collected in the past.
- Case 2. If T has *decreased* since the last `collectToken` call, the number of the newly collected tokens (more precisely $X + Y$) will *decrease*, meaning that more tokens have already been collected, as a percentage, than previously allowed. In this case, however, if $X + Y < C$, do not collect any token, meaning that $C' = C$.

Also, note that n and b can be derived directly from start and now, as follows:

$$
\begin{aligned}
n &= \left\lfloor \frac{\mathsf{now} - \mathsf{start}}{31536000} \right\rfloor \\
b &= \frac{\mathsf{now} - \mathsf{prev}}{31536000} = \frac{\mathsf{now} - \mathsf{start} \bmod 31536000}{31536000}
\end{aligned}
$$

where 31,536,000 is the number of seconds in a year ($= 365 \times 24 \times 3600$).

3.2 $\mathsf{collectToken}_{\mathsf{code}}$: High-Level Constructive Specification

We present $\mathsf{collectToken}_{\mathsf{code}}$, a constructive definition of $\mathsf{collectToken}_{\mathsf{spec}}$, in a form of pseudo-code, as shown in Figure 2. The arithmetic operations used in $\mathsf{collectToken}_{\mathsf{code}}$ are the purely mathematical ones, with no overflow nor rounding errors.

Lemma 1 bridges the gap between $\mathsf{collectToken}_{\mathsf{code}}$ and $\mathsf{collectToken}_{\mathsf{spec}}$.

Lemma 1. *If the inputs of `collectToken` in Figure 2 satisfies the following equations:*

$$
\begin{aligned}
\mathit{balance} &= T - C \\
\mathit{collectedTokens} &= C \\
\mathit{rewardStartTime} &= \mathit{start} \\
\mathit{now} &= \mathit{now}
\end{aligned}
$$

then the following holds for the outputs of the function:

$$
\begin{aligned}
\mathit{balance}' &= T - C' &= T - (X + Y) \\
\mathit{collectedTokens}' &= C' &= X + Y
\end{aligned}
$$

```
/* @input:   balance                                                    // $T - C$
 *           collectedTokens                                            // $C$
 *           rewardStartTime                                            // start
 *           now                                                        // now
 *
 * @output: balance'                                                    // $T - C'$
 *           collectedTokens'                                           // $C'$
 *
 * @pre-condition: now > rewardStartTime
 */
procedure collectToken() {
  total := collectedTokens + balance                                    // $T$
  yearCount := floor(days(now - rewardStartTime) / 365)                 // $n$
  fractionOfThisYear := (days(now - rewardStartTime) % 365) / 365       // $b$

  remainingTokens := total * (0.9 ^ yearCount)
  totalRewardThisYear := remainingTokens * 0.1                          // $Z$
  canExtractThisYear := totalRewardThisYear * fractionOfThisYear        // $Y$
  canExtract := canExtractThisYear + (total - remainingTokens)
                                     - collectedTokens                  // $Y + X - C$

  collectedTokens' := collectedTokens + canExtract                      // $C'$
  balance' := balance - canExtract                                      // $T - C'$
}
```

Figure 2: $\mathsf{collectToken}_{\mathsf{code}}$: All of the arithmetic operations are the purely mathematical ones, with no overflow nor rounding errors.

Proof. Immediate from the following equalities for the intermediate values, by definition of $\mathsf{collectToken}_{\mathsf{spec}}$ and $\mathsf{collectToken}_{\mathsf{code}}$.

$$\begin{aligned}
\mathsf{total} &= T \\
\mathsf{yearCount} &= n \\
\mathsf{fractionOfThisYear} &= b \\
\mathsf{remainingTokens} &= \mathsf{total} - X \\
\mathsf{totalRewardThisYear} &= Z \\
\mathsf{canExtractThisYear} &= Y \\
\mathsf{canExtract} &= X + Y - C
\end{aligned}$$

□

3.3 $\texttt{collectToken}_{\texttt{solidity}}$: Solidity-Level Functional Specification

We convert $\mathsf{collectToken}_{\mathsf{code}}$ to $\texttt{collectToken}_{\texttt{solidity}}$ by replacing the pure mathematical operations with the unsigned integer arithmetic operations of Solidity, as shown in Figure 3. $\texttt{collectToken}_{\texttt{solidity}}$ serves as the functional correctness specification of `collectToken`. Note that we use different fonts to distinguish different specification variables: the sans serif font for the $\mathsf{collectToken}_{\mathsf{code}}$ variables, and the `teletype` font for those of $\texttt{collectToken}_{\texttt{solidity}}$.

Note that there is a gap between $\mathsf{collectToken}_{\mathsf{code}}$ and $\texttt{collectToken}_{\texttt{solidity}}$, especially due to the integer division rounding errors. We next analyze the rounding errors to bridge the gap between them.

First, let us define the integer division in terms of the mathematical one with the floor operation as follows:

$$x \ / \ y \overset{\texttt{def}}{=} \left\lfloor \frac{x}{y} \right\rfloor$$

```
/* @input:  uint balance
 *          unit collectedTokens
 *          unit rewardStartTime
 *          unit now
 *
 * @output: unit balance'
 *          unit collectedTokens'
 *
 * @pre-condition: now > rewardStartTime
 */
procedure collectToken() {
  unit total := collectedTokens + balance
  unit yearCount := days(now - rewardStartTime) / 365
  unit fractionOfThisYear365 := days(now - rewardStartTime) % 365

  unit remainingTokens := power(total, 90, 100, yearCount)
  unit totalRewardThisYear := remainingTokens * 10 / 100
  unit canExtractThisYear := totalRewardThisYear * fractionOfThisYear365 / 365
  unit canExtract := canExtractThisYear + (total - remainingTokens)
                                         - collectedTokens

  collectedTokens' := collectedTokens + canExtract
  balance' := balance - canExtract
}

// return (conceptually): acc * ((base_n / base_d) ^ exp)
function power(acc, base_n, base_d, exp) {
  if exp == 0 {
    return acc
  } else {
    return power(acc * base_n / base_d, base_n, base_d, exp - 1)
  }
}
```

Figure 3: `collectToken`$_{\texttt{solidity}}$: All of the arithmetic operations are the unsigned integer arithmetics, where an exception is thrown when an overflow occurs.

where $/$ is the integer division. Since $r - 1 < \lfloor r \rfloor \leq r$ (for $r \in \mathbb{R}$), we have:

$$\frac{x}{y} - 1 < x \ / \ y \leq \frac{x}{y}$$

Let us introduce the epsilon (ϵ) notation, a small non-deterministic number within the range $[0, 1)$, i.e., $0 \leq \epsilon < 1$, so that we can represent the above inequalities in a simpler form as follows:

$$x \ / \ y = \left\lfloor \frac{x}{y} \right\rfloor = \frac{x}{y} - \epsilon$$

We start to analyze the error bound of $\texttt{remainingTokens} = \texttt{power(total,90,100,}n\texttt{)}$ for $n \geq 0$. First, we have:

$$\begin{aligned}
\texttt{power(total,90,100,0)} &= \texttt{total} \\[2em]
\texttt{power(total,90,100,1)} &= \texttt{total} \times 90 \ / \ 100 \\
&= \lfloor \texttt{total} \times 0.9 \rfloor \\
&= \texttt{total} \times 0.9 - \epsilon_1 \\[2em]
\texttt{power(total,90,100,2)} &= (\texttt{total} \times 90 \ / \ 100) \times 90 \ / \ 100 \\
&= \lfloor \lfloor \texttt{total} \times 0.9 \rfloor \times 0.9 \rfloor \\
&= (\texttt{total} \times 0.9 - \epsilon_1) \times 0.9 - \epsilon_2 \\
&= \texttt{total} \times 0.9^2 - \epsilon_1 \times 0.9 - \epsilon_2 \\[2em]
\texttt{power(total,90,100,3)} &= ((\texttt{total} \times 90 \ / \ 100) \times 90 \ / \ 100) \times 90 \ / \ 100 \\
&= \lfloor \lfloor \lfloor \texttt{total} \times 0.9 \rfloor \times 0.9 \rfloor \times 0.9 \rfloor \\
&= (\texttt{total} \times 0.9^2 - \epsilon_1 \times 0.9 - \epsilon_2) \times 0.9 - \epsilon_3 \\
&= \texttt{total} \times 0.9^3 - \epsilon_1 \times 0.9^2 - \epsilon_2 \times 0.9 - \epsilon_3
\end{aligned}$$

where $/$ is the integer division. Thus, in general, for any $n \geq 0$, we have:

$$\texttt{power(total,90,100,}n\texttt{)} = \texttt{total} \times 0.9^n - \epsilon_1 \times 0.9^{n-1} - \cdots - \epsilon_{n-1} \times 0.9 - \epsilon_n$$

By the definition of ϵ, we have:

$$\begin{aligned}
\texttt{total} \times 0.9^n \geq \texttt{power(total,90,100,}n\texttt{)} &> \texttt{total} \times 0.9^n - 0.9^{n-1} - \cdots - 0.9 - 1 \\
&= \texttt{total} \times 0.9^n - \frac{1 - 0.9^n}{0.1} \\
&> \texttt{total} \times 0.9^n - 10
\end{aligned}$$

Since $\texttt{total} = T$, we have

$$T \times 0.9^n - 10 < \texttt{remainingTokens} = \texttt{power(total,90,100,}n\texttt{)} \leq T \times 0.9^n \tag{1}$$

Next, we analyze the error bound of $\texttt{totalRewardThisYear} = \texttt{remainingTokens} \times 10 \ / \ 100$. By the definition of the integer division, we have:

$$\texttt{remainingTokens} \times 0.1 - 1 < \texttt{remainingTokens} \times 10 \ / \ 100 \leq \texttt{remainingTokens} \times 0.1$$

By the equation 1, we have:

$$\begin{aligned}\texttt{totalRewardThisYear} = \texttt{remainingTokens} \times 10 \texttt{ / } 100 &\le \texttt{remainingTokens} \times 0.1 \\ &\le T \times 0.9^n \times 0.1 \\ &= Z\end{aligned}$$

and

$$\begin{aligned}\texttt{totalRewardThisYear} = \texttt{remainingTokens} \times 10 \texttt{ / } 100 &> \texttt{remainingTokens} \times 0.1 - 1 \\ &> (T \times 0.9^n - 10) \times 0.1 - 1 \\ &= T \times 0.9^n \times 0.1 - 2 \\ &= Z - 2\end{aligned}$$

That is,

$$Z - 2 < \texttt{totalRewardThisYear} \le Z \tag{2}$$

Similarly, the error bound of `canExtractThisYear` is analyzed as follows. We have:

$$\texttt{totalRewardThisYear} \times b - 1 < \texttt{canExtractThisYear} \le \texttt{totalRewardThisYear} \times b$$

By the equation 2, we have:

$$\begin{aligned}\texttt{canExtractThisYear} &\le \texttt{totalRewardThisYear} \times b \\ &\le Z \times b \\ &= Y\end{aligned}$$

and

$$\begin{aligned}\texttt{canExtractThisYear} &> \texttt{totalRewardThisYear} \times b - 1 \\ &> (Z - 2) \times b - 1 \\ &= Z \times b - 2b - 1 \\ &> Z \times b - 3 && \text{(since } 0 \le b < 1) \\ &= Y - 3\end{aligned}$$

That is,

$$Y - 3 < \texttt{canExtractThisYear} \le Y \tag{3}$$

Finally, we analyze the error bound of `canExtract`. By the equations 1 and 3, we have:

$$(Y - 3) - T \times 0.9^n < \texttt{canExtractThisYear} - \texttt{remainingTokens} < Y - (T \times 0.9^n - 10)$$

Since

$$\begin{aligned}\texttt{canExtract} &= \texttt{canExtractThisYear} + (\texttt{total} - \texttt{remainingTokens}) - \texttt{collectedTokens} \\ &= \texttt{canExtractThisYear} + (T - \texttt{remainingTokens}) - C\end{aligned}$$

we have:

$$(X + Y - C) - 3 < \texttt{canExtract} < (X + Y - C) + 10 \tag{4}$$

Thus, the number of collected tokens calculated by the `collectToken` function may be up to 10 more than, or 3 less than the mathematical definition, due to the integer division rounding errors. The accumulated rounding error is constant-bounded, thus stable, no matter how large n is. The following lemma formulates this fact.

Lemma 2. *If the inputs of* `collectToken` *in Figure 3 satisfies the following equations:*

$$\begin{aligned} \mathtt{balance} &= T - C \\ \mathtt{collectedTokens} &= C \\ \mathtt{rewardStartTime} &= \mathsf{start} \\ \mathtt{now} &= \mathsf{now} \end{aligned}$$

then the following holds for the outputs of the function:

$$\begin{array}{rcccl} (T - C') - 10 & < & \mathtt{balance'} & < & (T - C') + 3 \\ C' - 3 & < & \mathtt{collectedTokens'} & < & C' + 10 \end{array}$$

Proof. By the equations 1, 2, 3, and 4. □

Another important property of `collectToken` is the monotonicity. That is, the number of collected tokens increases as time goes on. In other words, the following should always hold:

$$\mathtt{canExtractThisYear} + (\mathtt{total} - \mathtt{remainingTokens}) \geq \mathtt{collectedTokens}$$

While it is clear that the above holds over the real arithmetic, it is not clear whether the above holds over the integer arithmetic (due to the rounding errors).

Lemma 3. *Suppose two* `collectToken` *function (as shown in Figure 3) calls are made at times* t *and* t'*,respectively, where* $t < t'$*. Let* r *and* r' *be the output values of* $\mathtt{canExtractThisYear} + (\mathtt{total} - \mathtt{remainingTokens})$ *for each function call at* t *and* t'*, respectively. Assume that* `total` *does* not *descrese between* t *and* t'*. Then,* $r \leq r'$*.*

Proof. We only need to show in the case `total` is fixed, from which it is trivial to show in the case `total` increases. Also, if t and t' are in the same annual, it is trivial to show, since `remainingTokens` is fixed and `canExtractThisYear` is monotone within the annual. Thus, we only need to show for the case t and t' are in the different annual. Specifically, it is sufficient to show when t is the last second of the $n + 1^{\text{th}}$ annual, and t' is the first second of the $n + 2^{\text{th}}$ annual, for arbitrary $n \geq 0$. Then, we have:

$$\begin{aligned} r &= \mathtt{power}(T\mathtt{,90,100,}n) \times 10 \ / \ 100 \times 31535999 \ / \ 31536000 + (T - \mathtt{power}(T\mathtt{,90,100,}n)) \\ &< \mathtt{power}(T\mathtt{,90,100,}n) \times 10 \ / \ 100 + (T - \mathtt{power}(T\mathtt{,90,100,}n)) \\ &= T - \mathtt{power}(T\mathtt{,90,100,}n) + \mathtt{power}(T\mathtt{,90,100,}n) \times 10 \ / \ 100 \\ &= T - \mathtt{power}(T\mathtt{,90,100,}n) + \mathtt{power}(T\mathtt{,90,100,}n) \times 0.1 - \epsilon \\ &= T - \mathtt{power}(T\mathtt{,90,100,}n) \times 0.9 - \epsilon \end{aligned}$$

$$\begin{aligned} r' &= \mathtt{power}(T\mathtt{,90,100,}n) \times 10 \ / \ 100 \times 0 \ / \ 31536000 + (T - \mathtt{power}(T\mathtt{,90,100,}n+1)) \\ &= T - \mathtt{power}(T\mathtt{,90,100,}n+1) \\ &= T - \mathtt{power}(T\mathtt{,90,100,}n) \times 90 \ / \ 100 \\ &= T - (\mathtt{power}(T\mathtt{,90,100,}n) \times 0.9 - \epsilon) \\ &= T - \mathtt{power}(T\mathtt{,90,100,}n) \times 0.9 + \epsilon \end{aligned}$$

By the definition of ϵ, we conclude: $r \leq r'$. □

Now we can show that `collectToken` is monotonic even over the integer arithmetic.

Corollary 1. *If* `balance` *does not decrease since the last* `collectToken` *function call, the following always hold:*

$$\mathtt{canExtractThisYear} + (\mathtt{total} - \mathtt{remainingTokens}) \geq \mathtt{collectedTokens}$$

Proof. By Lemma 3, and the fact that `collectedTokens'` is set to (less than or) equal to $\mathtt{canExtractThisYear} + (\mathtt{total} - \mathtt{remainingTokens})$. □

```
pragma solidity ^0.4.18;

contract KeyRewardPool is DSMath{
  uint public collectedTokens;
  uint public balance;
  uint constant public yearlyRewardPercentage = 10;

  // @notice call this method to extract the tokens
  function collectToken(uint nowTime, uint rewardStartTime) public returns(bool){
    require(nowTime > rewardStartTime);

    uint total = add(collectedTokens, balance);
    uint remainingTokens = total;
    uint yearCount = yearFor(nowTime, rewardStartTime);

    for(uint i = 0; i < yearCount; i++) {
      remainingTokens =  div(mul(remainingTokens, 100 - yearlyRewardPercentage), 100);
    }
    uint totalRewardThisYear =  div(mul(remainingTokens, yearlyRewardPercentage), 100);

    // the reward will be increasing linearly in one year.
    uint canExtractThisYear = div(mul(totalRewardThisYear, (nowTime - rewardStartTime)  % 365 days), 365 days);
    uint canExtract = canExtractThisYear + (total - remainingTokens);
    canExtract = sub(canExtract, collectedTokens);

    if(canExtract > balance) {
      canExtract = balance;
    }

    collectedTokens = add(collectedTokens, canExtract);
    balance = sub(balance, canExtract);
    return true;
  }
  function yearFor(uint nowTime, uint rewardStartTime) public constant returns(uint) {
    return nowTime < rewardStartTime
      ? 0
      : sub(nowTime, rewardStartTime) / (365 days);
    }
}
```

Figure 4: Modified `collectToken` source code

3.4 $\texttt{collectToken}_{\texttt{evm}}$: EVM-Level Functional Specification

Here we present $\texttt{collectToken}_{\texttt{evm}}$, a refinement of $\texttt{collectToken}_{\texttt{solidity}}$, that captures EVM-specific details. $\texttt{collectToken}_{\texttt{solidity}}$ is a Solidity-level specification, intentionally omitting EVM-specific details such as gas consumption, data layout in storage, ABI encoding, and byte representation of the program. However, reasoning about the low-level details is critical because many security vulnerabilities are related to the EVM quirks.

We refine $\texttt{collectToken}_{\texttt{solidity}}$ to EVM-level variant $\texttt{collectToken}_{\texttt{evm}}$, which captures all of the detailed behaviors that can happen when the code is compiled and executed at the EVM level. That includes laying out the contract state variables in the EVM storage, encoding the program and the call data in bytes, and specifying additional information such as gas consumption.

We verified a mathematically equivalent variant of the `collectToken` function, as shown in Figure 4. Due to time constraints (we started working on this verification project on February 13, 2018), we have *not* verified the external call to `key.balanceOf` and `key.transfer`. The differences are as follows:

- Instead of calling `time()` to get the current time, `now` and `rewardStartTime` are given as input to the `collectToken` function.

- We declare `balance` as a global variable rather than calling `key.balanceOf` function. In the end, we directly update `balance` instead of calling `key.transfer` function.
- We change:

  ```
  uint canExtract = canExtractThisYear + total - remainingTokens;
  ```

  to:

  ```
  uint canExtract = canExtractThisYear + (total - remainingTokens);
  ```

  in order to avoid potential unnecessary overflow.
- We omit to log the `TokensWithdrawn` event.

In order to verify the `collectToken` function, we present the top-level spec (top-level$_{\text{spec}}$), together with the spec for the loop invariant (loop-invariant$_{\text{spec}}$). Specifically, we provide the specification template parameters from which the full specifications are derived by instantiating the template (Figure 7) and focus on explaining the EVM-specific detailed behaviors. For more details of the specification template and template parameters, refer to the eDSL in Section 1.3.

3.4.1 top-level$_{\text{spec}}$

top-level$_{\text{spec}}$ provides the functional specification at the EVM level, describing the pre- and post-conditions of the `collectToken` function. Below is the EVM specification template parameters for top-level$_{\text{spec}}$.

```
[topLevel]
k: #execute => (RETURN RET_ADDR:Int 32 ~> _)
output: _
memoryUsed: 0 => _
callData: #abiCallData("collectToken", #uint256(NOW), #uint256(START))
wordStack: .WordStack => _
localMem: .Map => .Map[ RET_ADDR := #asByteStackInWidth(1, 32) ] _:Map
pc: 0 => _
gas: GASCAP => _
log: _
refund: _ => _
storage:
    #hashedLocation("Solidity", {_COLLECTEDTOKENS}, .IntList) |-> (COLLECTED => COLLECTED +Int VALUE)
    #hashedLocation("Solidity", {_BALANCE},         .IntList) |-> (BAL       => BAL       -Int VALUE)
    _:Map
requires:
    andBool 0 <=Int COLLECTED  andBool COLLECTED <Int (2 ^Int 256)
    andBool 0 <=Int BAL        andBool BAL       <Int (2 ^Int 256)
    andBool 0 <=Int START      andBool START     <Int (2 ^Int 256)
    andBool 0 <=Int (COLLECTED +Int BAL) andBool (COLLECTED +Int BAL) *Int 3153600 <Int (2 ^Int 256)
    andBool 0 <Int (NOW -Int START) andBool (NOW -Int START) <Int (2 ^Int 256)
    andBool #accumulatedReleasedTokens(BAL, COLLECTED, START, NOW) >Int COLLECTED +Int 3
    andBool #accumulatedReleasedTokens(BAL, COLLECTED, START, NOW) <Int (BAL +Int COLLECTED) -Int 10
    andBool GASCAP >=Int (293 *Int ((NOW -Int START) /Int 31536000)) +Int 43000
ensures: VALUE ==Int @canExtractThisYear(COLLECTED +Int BAL, NOW, START)
                     +Int BAL -Int @remainingTokens(COLLECTED +Int BAL, NOW, START)
```

- `k` specifies that the execution eventually reaches the `RETURN` instruction, meaning that the program will successfully terminate.

- `memoryUsed` specifies that the initial memory consumption is initially 0. During the execution, `collectTokens` will write values into the memory. However, the exact amount of used memory by the end of the execution is irrelevant.

- `callData` specifies the call data using the `#abiCallData` eDSL notation. Refer to the resources in Section 1.3 for complete details.

- `wordStack` specifies that the local stack is initially empty. By the end of the execution, the stack may not be empty, but that is not relevant.

- `localMem` specifies that the local memory is empty in the beginning, but in the end, it will store the return value `true`, represented as `1`.

- `pc` specifies the program counter starting from 0.

- `gas` specifies the maximum gas amount, `GASCAP`, ensuring that the program does not consume more gas than the limit. Here we give a loose upper bound which is specified in `requires`.

- `log` specifies that no log is generated during the execution.

- `refund` specifies that a refund may be issued. Note that, however, we have not specified the refund detail since it is not essential for the functional correctness.

- `storage` specifies that the value of `collectedTokens` is `COLLECTED` and the value of `balance` is `BAL`. Other entries are not relevant (and could be arbitrary values). `_COLLECTEDTOKENS` and `_BALANCE` are the eDSL notations (called program-specific template parameters) that represent the position index of the corresponding variables. Refer to the resources in Section 1.3 for complete details.

- `requires` specifies the pre-conditions of the function.
  - The first 3 lines specify the range of symbolic values based on their types.
  - Line 4 specify the range of the total number of tokens. It should be sufficiently small to avoid multiplication overflow. Note that 3153600 is the seconds in an year divided by 10, i.e., $\frac{365\times24\times3600}{10}$.
  - Line 5 specifies the total seconds that have passed since `rewardStartTime` should be in the proper range of 256-bit unsigned integers.
  - Lines 6 and 7 specify the accumulated number of the released token until `now` since `rewardStartTime`, `#accumulatedReleasedTokens(BAL, COLLECTED, START, NOW)`[9], should be marginally greater than `collectedTokens` and smaller than the total number of tokens, considering the (bounded) rounding errors due to the integer division (See 3.2).
  - Line 8 specifies the loose upper bound on the gas cost, which depends on the input.

- `ensures` specifies that the number of newly collected token, `canExtract`, is correct, i.e., it is the same with that of $\texttt{collectToken}_{\texttt{solidity}}$. We use two macros `@canExtractThisYear` and `@remainingTokens` to succinctly specify that, defined as follows[10]:

```
rule @canExtractThisYear(TOTAL, NOW, START)
  => ((#roundpower(TOTAL, 90, 100, (NOW -Int START) /Int 31536000) *Int 10 /Int 100)
     *Int ((NOW -Int START) %Int 31536000)) /Int 31536000  [macro]

rule @remainingTokens(TOTAL, NOW, START)
  => #roundpower(TOTAL, 90, 100, (NOW -Int START) /Int 31536000)  [macro]
```

[9] `#accumulatedReleasedTokens` corresponds to $X + Y$ in $\mathsf{collectToken}_{\mathsf{spec}}$

[10] `#roundpower` is a macro corresponding to the `power` in the $\texttt{collectToken}_{\texttt{solidity}}$.

3.4.2 loop-invariant$_{\text{spec}}$

Below is the specification of the `for` loop inside the `collectTokens` function.

```
[loopInvariant]
k: #execute => #execute
output: _
memoryUsed: MU
callData: _
wordStack: (CANEXTRACT : CANEXTRACTTHISYEAR : TOTALREWARDTHISYEAR : INDEX : YEARCOUNT :
            REMAINING : TOTAL : RETURNVAL : START : NOW : RPC : FUNID : .WordStack)
        => (CANEXTRACT : CANEXTRACTTHISYEAR : TOTALREWARDTHISYEAR : YEARCOUNT : YEARCOUNT
            : #roundpower(REMAINING, 90, 100, YEARCOUNT -Int INDEX) : TOTAL : RETURNVAL :
            START : NOW : RPC : FUNID : .WordStack)
localMem: _
pc: 498 => 545
gas: GASCAP => GASCAP -Int (293 *Int (YEARCOUNT -Int INDEX)) -Int 26
log: _
refund: _
storage: _
requires: andBool 0 <=Int MU         andBool MU                <Int  (2 ^Int 256)
          andBool 0 <=Int INDEX      andBool INDEX             <=Int YEARCOUNT
          andBool 0 <=Int YEARCOUNT  andBool YEARCOUNT         <Int  (2 ^Int 256)
          andBool 0 <=Int REMAINING  andBool REMAINING *Int 90 <Int  (2 ^Int 256)
          andBool GASCAP >=Int ((293 *Int (YEARCOUNT -Int INDEX)) +Int 26)
```

Notable differences with the previous ones are as follows:

- `memoryUsed` specifies that the local memory is never used (no read/write) inside the loop. Indeed, the local variables are stored in the stack instead of the memory in this EVM bytecode.
- `wordStack` specifies all of the elements in the local stack before and after the execution of the loop. In the end, the loop index becomes `YEARCOUNT`, and the `remainingTokens`'s value becomes `#roundpower(REMAINING, 90, 100, YEARCOUNT -Int INDEX)`, which is the most important loop invariant.
- `pc` specifies the program counters for the loop head and the end of the loop.
- `gas` specifies the gas consumption which depends on the number of loop iterations.
- `requires` specifies the proper range of the symbolic values. Especially, `REMAINING` should be sufficiently small to avoid multiplication overflow.

3.4.3 Lemmas

We mechanize Corollary 1, which is trusted by the verifier, as follows:

```
rule      @canExtractThisYear(COLLECTED +Int BAL, NOW, START)
     +Int ((COLLECTED +Int BAL)
     -Int @remainingTokens(COLLECTED +Int BAL, NOW, START)) >=Int COLLECTED => true
  requires #shouldReleaseSofar(BAL, COLLECTED, START, NOW) >Int COLLECTED +Int 3
```

3.4.4 Proof

We took the modified source code, inlined the `DSMath` contract and compiled it to the EVM bytecode using Remix Solidity IDE (of the version `soljson-v0.4.20+commit.3155dd80`). The inlined source code can be found at `KeyRewardPool.modified.inlined.sol`.

```
function forwardToHotWallet(uint _amount) public notPaused onlyWithdrawer returns (uint) {
    require(_amount > 0);
    uint _time = time();
    require(_time > (lastWithdrawTime + 24 hours));

    uint amount = _amount;
    if (amount > withdrawLimit) {
        amount = withdrawLimit;
    }

    lastWithdrawTime = _time;
    return amount;
    // key.transfer(hotWallet, amount);
}
```

Figure 5: Modified `forwardToHotWallet` source code

The verification proof of the `collectToken` function can be reproduced by using the verifier. For the detailed instruction, refer to: `https://github.com/runtimeverification/verified-smart-contracts/tree/master/bihu`.

4 Formal Specification and Verification of `WarmWallet.forwardToHotWallet`

We verified a mathematically equivalent variant of the `forwardToHotWallet` function, as shown in Figure 5. Due to time constraints, we have *not* verified the external call to `key.transfer`. Instead, we slightly modified the source code to simply return the amount to transfer instead of actually transferring it. The following lemma formulates the correctness property that we verified.

Lemma 4. *`WarmWallet.forwardToHotWallet` succeeds in calling `key.transfer(hotWallet, amount)` with updating `lastWithdrawTime` to `now`, where `amount` = $\min(\texttt{\_amount}, \texttt{withdrawLimit})$, if and only if the following hold:*

$$\begin{aligned} \texttt{paused} &= 0 \\ \texttt{msg.sender} &= \texttt{withdrawer} \\ \texttt{\_amount} &> 0 \\ \texttt{\_time} &> (\texttt{lastWithdrawTime} + (24 \times 3600) \bmod 2^{256}) \end{aligned}$$

If at least one of the above does not *hold, it throws reverting the state to the original without draining the remaining gas (i.e., `revert`).*

Lemma 4 is fully mechanized in and automatically proved by the K reachability logic theorem prover. Figure 6 shows the specification template parameters from which the full specifications are derived by instantiating the template (Figure 7). The specification makes use of some advanced features of our DSL for EVM specifications. Refer to the eDSL in Section 1.3 for complete details.

We explain two sub-cases in the following.

Success Case 1: `_amount` $\leq$ `withdrawLimit`. Below is part of the EVM-level specification for the success case 1.

```
[forwardToHotWallet-success-1]
+requires:
     andBool AMOUNT <=Int WITHDRAWLIMIT // if (amount > withdrawLimit) { ... }
    ensures  RET_VAL ==Int AMOUNT
```

```
[forwardToHotWallet]
callData: #abiCallData("forwardToHotWallet", #uint256(AMOUNT))
gas: 22000 => _
log: _
refund: _
requires:
     andBool 0 <=Int AMOUNT             andBool AMOUNT              <Int (2 ^Int 256)
     andBool 0 <=Int WITHDRAWER         andBool WITHDRAWER          <Int (2 ^Int 160)
     andBool 0 <=Int WITHDRAWLIMIT      andBool WITHDRAWLIMIT       <Int (2 ^Int 256)
     andBool 0 <=Int LASTWITHDRAWTIME   andBool LASTWITHDRAWTIME    <Int (2 ^Int 256)
     andBool 0 <=Int PAUSED             andBool PAUSED              <Int (2 ^Int 256)

[forwardToHotWallet-success]
k: #execute => (RETURN RET_ADDR:Int 32 ~> _)
output: _
localMem: .Map => ( .Map[ RET_ADDR := #asByteStackInWidth(RET_VAL, 32) ] _:Map )
storage:
    #hashedLocation("Solidity", {_WITHDRAWER},       .IntList) |-> WITHDRAWER
    #hashedLocation("Solidity", {_WITHDRAWLIMIT},    .IntList) |-> WITHDRAWLIMIT
    #hashedLocation("Solidity", {_LASTWITHDRAWTIME}, .IntList) |-> ( LASTWITHDRAWTIME => NOW )
    #hashedLocation("Solidity", {_PAUSED},           .IntList) |-> PAUSED
    _:Map
+requires:
     andBool PAUSED ==Int 0                                     // notPaused
     andBool CALLER_ID ==Int WITHDRAWER                         // onlyWithdrawer
     andBool AMOUNT >Int 0                                      // require(_amount > 0);
     andBool NOW >Int LASTWITHDRAWTIME +Word (24 *Int 3600)     // require(_time > (lastWithdrawTime + 24 hours));

[forwardToHotWallet-success-1]
+requires:
     andBool AMOUNT <=Int WITHDRAWLIMIT // if (amount > withdrawLimit) { ... }
    ensures  RET_VAL ==Int AMOUNT

[forwardToHotWallet-success-2]
+requires:
     andBool AMOUNT >Int WITHDRAWLIMIT // if (amount > withdrawLimit) { ... }
    ensures  RET_VAL ==Int WITHDRAWLIMIT

[forwardToHotWallet-failure]
k: #execute => #revert
output: _ => .WordStack
localMem: .Map => _
storage:
    #hashedLocation("Solidity", {_WITHDRAWER},       .IntList) |-> WITHDRAWER
    #hashedLocation("Solidity", {_WITHDRAWLIMIT},    .IntList) |-> WITHDRAWLIMIT
    #hashedLocation("Solidity", {_LASTWITHDRAWTIME}, .IntList) |-> LASTWITHDRAWTIME
    #hashedLocation("Solidity", {_PAUSED},           .IntList) |-> PAUSED
    _:Map
+requires:
     andBool (
             PAUSED ==Int 1 <<Int (20 *Int 8)                   // notPaused
      orBool CALLER_ID =/=Int WITHDRAWER                        // onlyWithdrawer
      orBool AMOUNT <=Int 0                                     // require(_amount > 0);
      orBool NOW <=Int LASTWITHDRAWTIME +Word (24 *Int 3600)    // require(_time > (lastWithdrawTime + 24 hours));
             )
```

Figure 6: Specification of `forwardToHotWallet`

It specifies that in the success case `forwardToHotWallet` transfers `_amount` if `_amount` is less than or equal to `withdrawLimit`.

Success Case 2: `_amount` > `withdrawLimit`. Below is the EVM-level specification for the success case 2.

```
[forwardToHotWallet-success-2]
+requires:
     andBool AMOUNT >Int WITHDRAWLIMIT // if (amount > withdrawLimit) { ... }
    ensures  RET_VAL ==Int WITHDRAWLIMIT
```

It specifies that in the success case `forwardToHotWallet` transfers `withdrawLimit` if `_amount` is greater than `withdrawLimit`.

4.1 Proof

We took the modified source code, inlined the `DSToken` contract interface and compiled it to the EVM bytecode using Remix Solidity IDE (of the version `soljson-v0.4.20+commit.3155dd80`). The inlined source code can be found at `WarmWallet.modified.inlined.sol`.

The verification proof of the `forwardToHotWallet` function can be reproduced by using the verifier. For detailed instructions, refer to: `https://github.com/runtimeverification/verified-smart-contracts/tree/master/bihu`.

```
rule
  <k> {K} </k>
  <exit-code> 1 </exit-code>
  <mode> NORMAL </mode>
  <schedule> BYZANTIUM </schedule>
  <ethereum>
    <evm>
      <output> {OUTPUT} </output>
      <memoryUsed> {MEMORYUSED} </memoryUsed>
      <callDepth> CALL_DEPTH </callDepth>
      <callStack> _ => _ </callStack>
      <interimStates> _ </interimStates>
      <substateStack> _ </substateStack>
      <callLog> .Set </callLog> // for vmtest only
      <txExecState>
        <program> #asMapOpCodes(#dasmOpCodes(#parseByteStack({CODE}), BYZANTIUM)) </program>
        <programBytes> #parseByteStack({CODE}) </programBytes>
        <id> ACCT_ID </id>
        <caller> CALLER_ID </caller> // msg.sender                |    <previousHash> _ </previousHash>
        <callData> {CALLDATA} </callData> // msg.data             |    <ommersHash> _ </ommersHash>
        <callValue> {CALLVALUE} </callValue> // msg.value         |    <coinbase> _ </coinbase>
        <wordStack> {WORDSTACK} </wordStack>                      |    <stateRoot> _ </stateRoot>
        <localMem> {LOCALMEM} </localMem>                         |    <transactionsRoot> _ </transactionsRoot>
        <pc> {PC} </pc>                                           |    <receiptsRoot> _ </receiptsRoot>
        <gas> {GAS} </gas>                                        |    <logsBloom> _ </logsBloom>
        <previousGas> _ => _ </previousGas>                       |    <difficulty> _ </difficulty>
        <static> false </static> // NOTE: non-static call         |    <number> _ </number>
      </txExecState>                                              |    <gasLimit> _ </gasLimit>
      <substate>                                                  |    <gasUsed> _ </gasUsed>
        <selfDestruct> _ </selfDestruct>                          |    <timestamp> NOW </timestamp> // now
        <log> {LOG} </log>                                        |    <extraData> _ </extraData>
        <refund> {REFUND} </refund>                               |    <mixHash> _ </mixHash>
      </substate>                                                 |    <blockNonce> _ </blockNonce>
      <gasPrice> _ </gasPrice>                                    |    <ommerBlockHeaders> _ </ommerBlockHeaders>
      <origin> ORIGIN_ID </origin> // tx.origin                   |    <blockhash> _ </blockhash>
    </evm>
    <network>
      <activeAccounts> ACCT_ID |-> false _:Map </activeAccounts>
      <accounts>
        <account>
          <acctID> ACCT_ID </acctID>
          <balance> _ </balance>
          <code> #parseByteStack({CODE}) </code>
          <storage> {STORAGE} </storage>
          <nonce> _ </nonce>
        </account>
        ...
      </accounts>
      <txOrder> _ </txOrder>
      <txPending> _ </txPending>
      <messages> _ </messages>
    </network>
  </ethereum>
  requires 0 <=Int ACCT_ID    andBool ACCT_ID    <Int (2 ^Int 160)
   andBool 0 <=Int CALLER_ID  andBool CALLER_ID  <Int (2 ^Int 160)
   andBool 0 <=Int ORIGIN_ID  andBool ORIGIN_ID  <Int (2 ^Int 160)
   andBool 0 <=Int NOW        andBool NOW        <Int (2 ^Int 256)
   andBool 0 <=Int CALL_DEPTH andBool CALL_DEPTH <Int 1024
   {REQUIRES}
```

Figure 7: eDSL Specification Template