Armors, the security ecosystem for smart contracts

Armors Lab

Jun 18 2018

Abstract

Armors bring bullet proof security to blockchain ecosystems, thus restore the balance between digital autonomy and cybersecurity to blockchain as the future cyber-societal OS.

Lately many bigname hacking events has been posing a serious threat to the growth of a healthy and sustainable blockchain community. As is widely accepted that the future belongs to mass collaboration by decentralization, security flaws and oversights, lacking of protection, governance, oversight and regulation on different parts in blockchain has proven to be one of the biggest problems of the whole blockchain industry. Recently researchers from the University of Singapore released a paper showing that 45% of Ethereum contracts are vulnerable.[1]

Armors recognized the missing parts and has positioned itself as the security guard of blockchain infrastructure and applications. Through a crowd sourced exploit database, a bug bounty program/platform and a system of algorithmic static code analysis, Armors provide the most up-to-date smart contract scanning and early alert engine, including patented Digital Immunity System (DIS), Security Nerve System (SNS) and Anomaly Migration System (AMS). At the same time, Armors ecosystem also provide compile time and run time protection through its dynamic AI and machine learning driven engine on potential security breaches. Through creating an open and free white-hat community, Armors aims to build a vibrant and innovative ecosystem.

We also discuss other attacks as well as case studies. In the end, we elaborate how Armors will use its static analyzing and dynamic monitoring to effectively detect, prevent and migrate those attacks, by building an ecosystem which incentize different roles to provide full-ranged blockchain security coverage.

1 Introduction

Throughout human civilization, people are struggling to reach a perfect balance of freedom and security. All human organization structures revolves different perspectives over consensus among a specific crowd, including governments, companies, communes, etc. Blockchain itself, as the future cyber-societal OS, has the inherent problem as well. Originally, blockchain is hailed as the technical holy grail to find the best way to organize mass human collaboration and profit distribution by the digital anarchists. Meanwhile, more freedom means more responsibility. In the days where a robust security protocol is missing and most people are still lacking knowledge, skills and awareness of cybersecurity, incessant hacking activities and smart contract vulnerabilities poses great doubts over the feasibility of blockchain.

Since blockchains feature a distributed protocol for a set of network computing nodes to reach a consensus on the global state, smart contracts —the stateful executable objects hosted on blockchains carry billions of dollars worth of coins and cannot be updated or patched once deployed, thus creating huge spaces for hackers. Smart contracts extend the idea of a blockchain to a universal computing cluster for decentralized execution of Turing-complete applications. Contracts are programs that run on blockchains: their code and state is stored on the blockchain, and they can update their states. Smart contracts have been popularized by the Ethereum and the late EOS. Recently, sophisticated applications of smart contracts have arisen, especially in the area of token management due to the development of ERC20 token standard. This standard provides a universal interface of various tokens, enabling decentralized exchanges and complex wallets. Today, over a million smart contracts operate on the Ethereum network and more is to come as EOS mainnet came online.

At Armors, we leverage our crowd sourced platform and AI driven DIS, SNS and AMS to act together to form a complete cyber-security ecosystem. Our decentralized AI system detects patterns of the attacker behaviors as well as smart contract code vulnerabilities, sends alarms and forestall potential suspicious attacks and delivers information throughout our ecosystem, meanwhile maintaining the autonomy and anonymousness of roles in the blockchain.

2 Blockchain Security

- Overview

Smart contracts can automate many different processes and operations between human and software, with the most obvious being payment and the actions pending on payment. But those capabilities extend to many external organizational activities. According to The Chamber of Digital Commerce, "Smart Contracts: 12 Use Cases for Business & Beyond", twelve smart contract use cases are presented across a range of industries and topics, such as:

- Land title recording
- Mortgages
- Digital identity
- Supply chain
- Financial data processing
- Over-the-counter (OTC) derivatives
- Auto insurance
- Public notary service
- Securities
- Trade finance
- Drug discovery
- Clinical trials

More than 1,000 decentralized applications (DApps) have been deployed on the Ethereum network since 2017, based on research from the analytics startup Alethio.

According to Nick Szabo, who first presented to public the concept of smart contract: [2]

Protocols for smart contracts should be structured in such a way as to make their contracts (a) robust against naive vandalism, and (b) robust against sophisticated, incentive compatible (rational) breach.

A vandal can be a strategy or sub-strategy of a game whose utility is at least partially a function of one's own negative utility; or it can be a mistake by a contracting party to the same effect. "Naive" simply refers to both lack of forethought as to the consequences of a breach, as well as the relatively low amount of resources expended to enable that breach. Naive vandalism is common enough that it must be taken into consideration. A third category, (c) sophisticated vandalism (where

the vandals can and are willing to sacrifice substantial resources), for example a military attack by third parties, is of a special and difficult kind that doesn't often arise in typical contracting, so that we can place it in a separate category and ignore it here. The distinction between naive and sophisticated strategies has been computationally formalized in algorithmic information theory.

For the first category, we use 1) pre-compiling 2) formal verification 3) cross-compiling. For the second category, we provide dynamic runtime monitoring and check. We will detail our findings as follows.

Blockchain is a decentralization technology which still unavoidably involves some centralized parts, including exchange, wallet etc. Those are within the traditional definition of cybersecurity and falls out of Armor's first priority concern and therefore discussion of this whitepaper.

- Product

  According to Deloitte's research in '17 [1], a complete blockchain security framework would need to be secure, vigilant and resilient.

Figure 1: Blockchain Security Strategy and Armors Product Scope

To achieve security, Armors DIS provide a suite of cross compilers, currently including three languages, Haskell [3] and F* [4], targeted to mainstream smart contract platforms Ethereum, EOS, Counterparty

[1] `https://www2.deloitte.com/content/dam/Deloitte/ie/Documents/Technology/IE_C_BlockchainandCyberPOV_0417.pdf`

and Rootstock. Haskell is targeted to mainstream developers familiar with functional programming paradigms, type theory and category theory. Cardano project is using Haskell as main development language to help security and improve effectiveness of it academic peer reviews. One step further, we encourage developers to use F*, to have formal verification at cross-compiler level. F* is a functional programming language inspired by ML and aimed at program verification. Its type system includes dependent types, monadic effects, and refinement types. This allows expressing precise specifications for programs, including functional correctness and security properties. The F* typechecker aims to prove that programs meet their specifications using a combination of SMT solving and manual proofs.

To achieve vigilance, Armors provide a static vulnerability scanning during compiling time. Armors SNS provides a practical framework for automatic formal verification of smart contracts using abstract interpretation and symbolic model checking. SNS takes as input the smart contracts written in specified languages and parses the source code into Abstract Syntax Trees (ASTs). Based on the generated semantic data structure, it cross compiles smart contracts into a low-level intermediate representation (IR), such as LLVM bitcode, EVM bytecode, eWASM bytecode, JVM bytecode etc. (in our case: LLVM), encoding the execution semantics to correctly reason about the contract behavior. It then performs static analysis atop the IR to determine the points at which the verification predicates (as specified in the policy) must be asserted. Finally, SNS feeds the modified IR to a verification engine that leverages constrained horn clauses (CHCs) to quickly ascertain the safety of the smart contract.

To achieve resilience, we provide Anomaly Migration System (AMS), a decentralized back-up mechanism. This will periodically migrates the state of the users and contracts. In cases of emergency, the saved mirror will be used to restore the user's state and data and rollback the effects of malicious operations. The smart contract author can specify the backup timing to be either periodic or scenario-based.

3 Solution

3.1 Digital Immunity System (DIS)

Armors provide three main programming languages that can be cross compiled into major smart contract bytecodes on EVM, eWASM etc.

- Haskell

  Basically, Haskell is a computer programming language using complex mathematical logic in coding —that then makes the language not same like the others. There are a lot of cryptographic codes in writing blockchain protocols, which need extreme degree fault tolerance structure. Haskell has proved to be a good choice in Cardano community.

  First above all, Haskell is a functional language with advanced typing, which itself could help to forestall and prevent most vulnerabilities stemming from malpractice or inadvertent programming, i.e. the first type of weakness defined by Szabo. In particular, the ability to use functional programs as specifications enables statically-typed metaprogramming: programs write programs, and static type-checking guarantees that the generating process never produces invalid code. Since our focus is on generic validity properties rather than full correctness verification, it is possible to engineer type inference systems that are very effective in narrow domains. A sample of Haskell which defines a HTML++ *Document* type [5]:

  One of Haskell's great strengths is how the language can be used to restrict the kinds of damage caused by coding errors. Like many languages, Haskell has a strong type system to enforce invariants and catch some forms of undefined behavior, as well as a module system that allows for encapsulation and abstraction. More distinctively, Haskell segregates pure and impure code via the IO monad and facilitates the implementation of alternate monads (such as ST) that bound the possible side effects of computations. Thus stateless computations can be optimized and paralleled to boost the efficiency and security of the smart contracts.lo

  An interesting question is whether these features can be used to control the effects of not just buggy but outright malicious code. For instance, consider implementing a server that accepts and executes Haskell source code from untrusted network clients. Such a server should not allow one client's code to subvert another client's code or,

```
data Document Document {
     docType :: DocumentType,
     docHead :: DocumentHead,
     docBody :: DocumentBody
}

data DocumentType = DOC_TYPE_HTML_4_01_STRICT
                  | DOC_TYPE_HTML_4_01_TRANS
                  | ...
                  | DOC_TYPE_XHTML_1_1

data DocumentHead = DocumentHead {
     docTitle :: String,
     docLinks :: [Node],
     docScripts :: [Node],
     docBaseUrl :: Maybe Url,
     docBaseTarget :: Maybe Target,
     docProfile :: [Url]
}

data DocumentBody = DocumentBody {
     docBodyNode :: Node
}
```

Figure 2: Definition for the Document type.

worse yet, make arbitrary calls to the underlying operating system. Such a server is analogous to web browsers, which use language features of Java and JavaScript to confine executable content. However, because of Haskell's support for arbitrary monads, it can enable a broader range of confinement policies than most other systems.

Besides, Haskell provide the following features that can avoid most vulnerabilities from coding errors:

- Type safety. In Milner's famous phrase, well-typed programs do not go wrong.
- Referential transparency. Functions in the safe language must be deterministic. Evaluating them should not cause any side effects, but - may result in non-termination or an exception.
- Module encapsulation. Haskell provides an effective module system that is used to control access to functions and data types. In the - safe language these module boundaries are strictly enforced; a user of a module that contains an abstract data type is only able to - access or create values through the functions exported.
- Modular reasoning. Adding a new import to a module should not change the meaning of existing code that doesn't directly depend on the - imported module.
- Semantic consistency. Any valid Haskell expression that compiles both in the safe language and in the full language must have the same meaning.

- F*

  We also advocate F* as the extended smart contract programming language to enable formal verification on smart contract code. F* is a verification system for ML programs developed collaboratively by Inria and Microsoft Research. ML types are extended with logical predicates that can conveniently express precise specifications for programs (pre and post conditions of functions as well as stateful invariants), including functional correctness and security properties. The F* type checker implements a weakest-precondition calculus [6] to produce first-order logic formulas that are automatically discharged using the Z3 SMT solver [7]. The original F* implementation [**?**] has been successfully used to verify nearly 50,000 lines of code, including cryptographic protocol implementations [31, 6], web browser extensions [19, 35], cloud-hosted web applications, and key parts of the F* compiler itself [8].

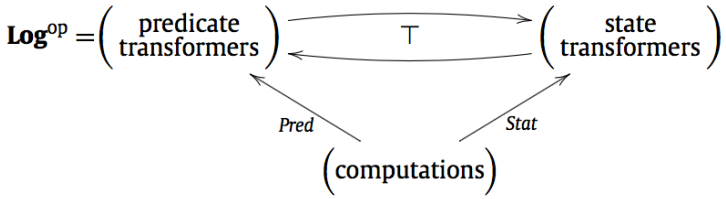
Therefore F* turns out to be a great language to write mission critical code like in a smart contract. Zen Protocol [2] is using F* as its smart contract language and we are seeing more projects are considering using such language to boost security and effectiveness. Those protocols uses F*'s powerful proof techniques both to create proofs of correctness, and to prove that contracts execute in a limited amount of time. Proofs of correctness mean that anyone can write a new contract, implementing a particular kind of asset —a call option for example— and any user or contract can immediately discover and use that asset, with confidence that it performs as it claims to.

A monad is a categorical concept that is surprisingly useful in the theory of computation. On the one hand it describes a form of computation (such as partial, non-deterministic, or probabilistic), and on the other hand it captures various algebraic structures (a.k.a. Algebraic Data Types). Monads have traditionally been used to structure effectful computations in functional programs. For example, the state monad, ST a, is the type of a computation that when evaluated in an initial heap, produces a result of type a and a final heap. In a purely functional language like Coq, ST a can be represented by the type $heap \mapsto (a \times heap)$. In a language that provides primitive support for state, e.g., ML, the type ST a can just be an abstract alias for a. As a monad, the abstract type ST comes with two operations: \$returnST : $\forall$ a. a $\mapsto$ STa\$$andbindST$: $\forall a, b. STa \mapsto (a \mapsto STb) \mapsto STb$, and these are expected to satisfy certain laws.

Technically, usual computations are maps in the Kleisli (*not khaleesi*) category of the monad, whereas the algebraic structures are described via the category of so-called Eilenberg-Moore algebras. The Kleisli approach has become common in program semantics and functional programming (notably in the language Haskell), starting with the seminal paper [9]. The algebraic structure captured by the monad exists on these programs (as Kleisli maps), technically because the Kleisli category is enriched over the category of algebras. The range of examples of monads has been extended here from computation to program logic. Hoare monads [24,29] and Dijkstra monads [28] have been defined in a systematic approach to program verification. Via these monads one describes not only a program but also the associated correctness assertions.

[2] `https://www.zenprotocol.com`

Since the Dijkstra and Hoare monads combine both semantics and logic of programs, we need to look at these two areas in a unified manner. From previous work [10] , a view on program semantics and logic emerged involving a triangle of the form:



The three nodes in this diagram represent categories of which only the morphisms are described. The arrows between these nodes are functors, where the two arrows at the top form an adjunction. The two triangles involved should commute. In the case where two up-going "predicate"and "state"functors $Pred$ and $Stat$ in (1) are full and faithful, we have three equivalent ways of describing computations. On morphisms, the predicate functor yields what is called substitution in categorical logic, but what amounts to a weakest precondition operation in program semantics. The upper category on the left is of the form Log_{op}, where Log is some category of logical structures. The opposite category op is needed because predicate transformers operate in the reverse direction, taking a post-condition to a precondition. We do not expand on the precise logical structure involved (which connectives, which quantifiers, etc. in Log) and simply claim that this 'indexed category'on the left is a model of some predicate logic. The reason is that at this stage we don't need more structure than 'substitution', which is provided by the functoriality of $Pred$.

So let $T : \mathbf{Sets} \to \mathbf{Sets}$ be an arbitrary monad. Each (fixed) Eilenberg-Moore algebra

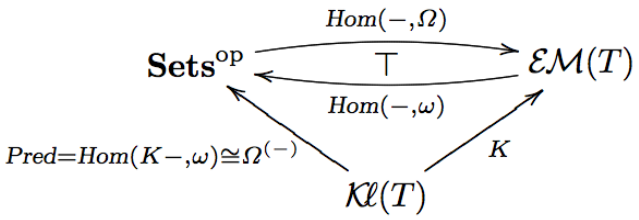
$$\omega : T(\Omega) \to \Omega$$

determines an adjunction $Sets^{op} \rightleftharpoons \mathcal{EM}(\mathcal{T})$, via functors

$$\Omega^{(-)} : Sets^{op} \to \mathcal{EM}(T)$$

and $Hom(-, \omega) : \mathcal{EM}(T) \to Sets^{op}$. It makes sense to require that the algebra ω is a cogenerator in $EM(T)$, making the unit of the adjunction injective, but this is not needed in general. The adjunction can be

generalized to strong monads on monoidal categories with equalizers, but that is not so relevant at this stage.



The induced predicate functor Pred is defined on a Kleisli map $f : X \to T(Y)$ as:

$$\Omega^Y \ni q \longrightarrow (X \xrightarrow[f]{T} (Y) \xrightarrow[T(q)]{T} (\Omega) \mapsto \Omega)$$

We define a fixed set of states S, a Dijkstra monad, namely:

$$\mathcal{D}_T(X) = Sets^{op}(Pred(S), Pres(S \times S)) = Sets(\Omega^{S \times X}, \Omega^S)$$

For D, There is a unit $\eta X : X \mapsto \mathcal{D}_T(X)$, namely $\eta X(x)(q)(s) = q(s, x)$, and a multiplication $\mu_X : (\mathcal{D}_T)^2(X) \mapsto \mathcal{D}_T(X)$ given by $\mu(H)(q) = H\lambda(t, k) \cdot k(q)(t)$.

Dijkstra monads [[6]] are the mechanism by which F* efficiently computes verification conditions, generically for all the effects of F*. While the verification condition generation algorithm is generic, for each effect one needs to define the operations used to combine weakest preconditions (return, bind, and a dozen others). This is rather subtle, because these operations are phrased in continuation-passing style and are subject to significant (meta-level) proof obligations in order to preserve the soundness of F*. Moreover, at the moment Dijkstra monads only work for the primitive effects of F* (basically the effects of ML), and we have no good way of adding new user-specified effects. Finally, the weakest precondition calculus is currently the only way by which one can reason about impure code, which means that all proofs about impure code have to be done intrinsically, when the code is defined. Proving properties extrinsically, after the fact, currently only works for pure code.

Relying on the types stated in F*, we obtain higher-order verification conditions (VCs) for programs that use a higher order store. Via a novel encoding, we will show that these higher-order VCs can be discharged by an off-the-shelf automated SMT solver (we use Z3).

- Armors will also provide a smart contract development framework to simplify and consolidate current efforts on security and efficiency. [3] It will also be included as part of the armors SDK and IDE. Armors will mostly focus on a one-stop solution for smart contract security. But other than that, Armors will also include public chain and web security in smart contract era (such as exchanges).

- Armors will build a upgradeable virtual level to effectively separate the conctract/program layer with distributed ledger/consensus level.

3.2 Security Nerve System (SNS)

Besides the native development IDE support with Haskell and F*, Armors SNS provides scanning and detection service for most mainstream blockchains. Armors smart contract scanner leverages both abstract interpretation and symbolic model checking, along with the power of constrained horn clauses to quickly verify contracts for safety. We have built a prototype of SNS for Ethereum and Counterparty blockchain platforms, and evaluated it with over 21.9K smart contracts. Our evaluation indicates that about 96.7% of contracts (including tokens worth more than $500 million) are vulnerable.

Armors SNS leverages three key observations to be both sound and salable. First, while the blockchain has execution akin to a concurrent system with task-based semantics, a transaction comprises of just one call chain starting from a publicly visible function in the smart contract. This observation helps significantly reduce the state space exploration for verifying most properties. Also, data dependence across transactions, such as read/write hazards among persistent state variables, requires analyzing $O(n^2)$ pairs of transaction inter-leavings. Second, smart contracts are both control and data driven. Thus, modeling contracts using abstract interpretation along with symbolic model checking allows SNS to soundly reason about program behavior. Abstract interpretation computes loop and function summaries over data domains, which are then used during the model checking phase

[3] `https://github.com/armors/armors-solidity`

that now operates upon a reduced state space. Lastly, CHCs provide a suitable mechanism to represent verification conditions, which can be discharged efficiently by SMT solvers.

The use of IR bitcode/bytecode also helps verification of smart contracts for existing different blockchain platforms, including Ethereum, EOS and Hyperledger Fabric, written in diverse high-level languages, such as C#, Go and Java. Note that most high-level languages have mature source code to LLVM bitcode translators already available. We leverage LLVM's rich API set to develop a Solidity to LLVM bitcode translator, which faithfully implements execution semantics for majority of the Solidity syntax for verification. Furthermore, use of IR code allow SNS to separate translation from implementation of verification checks.

SNS consists of (a) security policy builder, (b) source code parser and cross compiler, and (c) verifier. Specifically, SNS takes as input a smart contract and a policy (written in a JSON) against which the smart contract must be verified. It performs static analysis atop the smart contract code and inserts the policy predicates as assert statements at correct program points. SNS then leverages its source code translator to faithfully convert the smart contract embedded with policy assertions to LLVM bitcode. Finally, SNS invokes its verifier to determine assertion violations, which are indicative of policy violations.

- Source code formalization

  We define an abstract language that captures relevant constructs of smart contract programs. A program consists of a sequence of contract declarations. Each contract is abstractly viewed as a sequence of one or more method definitions in addition to declaration and initialization of persistent storage private to a contract, denoted by the keyword *global*. A contract is uniquely identified by *Id*, where *Id* belongs to a set of *identifiers*. This invocation of the contract's publicly visible methods is viewed as a transaction.

  The global state is defined by the tuple: $\langle\langle \mathcal{T}, \sigma\rangle, BC\rangle$ where $\langle T, \sigma\rangle$ is the block B being currently mined. BC is the list of committed blocks, and $\mathcal{T}$ denotes the multi-set of completed transactions that are not yet committed. Let $Vals \subseteq N$ be the set of values that expressions can take after evaluations. σ is the global state denoted by the function $\sigma : Id \mapsto g$ that maps contract identifiers to a valuation of the global variables, where $g \in Vals$. Note that σ is the state of the system reached after executing T in an order as specified by the miner. Finally,

each miner will add B to their respective copies of blockchain once it is validated. A transaction is defined as a stack of frames represented by γ. Each frame is further defined as: $f := \langle l, id, M, pc, \nu \rangle$, where $l \in Vals$ is the valuation of the method-local variables l, M is the code of the contract with the identifier id, pc is the program counter, and $\nu := \langle i, o \rangle$ is an auxiliary memory for storing input and output. The top frame of ν is the frame under active execution and is relevant to the currently executing transaction; it is not part of the persistent blockchain state. An empty frame is denoted by σ. A configuration c, defined as $c := \langle \nu, \sigma \rangle$, captures the state of transaction execution and denotes the small-step operational semantics. Rules for remaining sequential statements are standard. The symbol $\mapsto$ is overloaded to illustrate a transition relation for globals and blockchain states. The symbol $\leftarrow$ indicates an assignment to an lvar.

- Security policy specification

  We define vector $\langle Sub, Obj, Op, Cond, Res \rangle$. Subject $Sub \in PVars$ is the set of source variables (one or more) that need to be tracked. Object Ob $j \in PVars$ is the set of variables representing entities with which the subject interacts. Operation Op is the set of side-affecting invocations that capture the effects of interaction between the subject and the object. Op also specifies a trigger attribute, either 'pre'or 'post', indicating whether the predicates should hold before or after the specified operation. In other words, $Op := \langle f, trig \rangle$ where $f \in Func$ and $trig \in pre, post$. Condition $Cond \in Expr$ is the set of predicates that govern this interaction leading to the operation. Finally, $Res \in T, F$ indicates whether the interaction between the subject and operation as governed by the predicates is permitted or constitutes a violation.

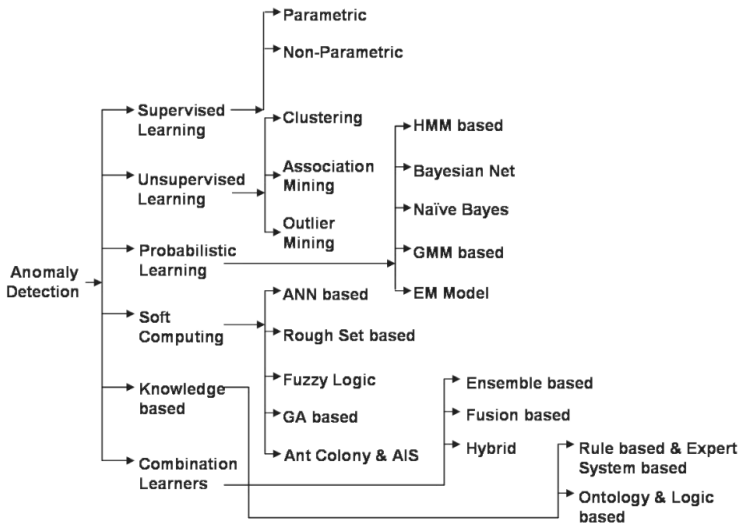
- Verification

  The proof for soundness of the translation from a smart contract to our abstract language with assertions (corresponding to policy predicates above), and finally into LLVM bitcode entails the following steps. First, we discuss that translation of the source code into our abstract language does not affect semantic behavior. Second, we argue that a conservative placement of asserts does not affect the soundness of the approach. Third, we reduce the problem of policy confirmation to a state reachability problem. Fourth, we provide a definition of state reachability in the context of a Solidity program. Fifth, we demon-

strate that by ensuring state reachability on an overapproximate version of the program, we do not miss on any program behaviors. Lastly, we argue that since our translation from this over-approximate source to LLVM bitcode is a faithful expression-to-expression translation, our overall soundness modulo the decision procedure is preserved.

SNS uses Seahorn [11] to emit verification conditions as CHCs for the translated program $\hat{P}'''$. The strength of the CHC representation enables it to interface with a variety of SMT-based solvers and off-the-shelf model checkers.

- Behavior-based proactive network anomaly detection Anomaly detection is essentially a multistep process for the mitigation of anomalies or failures when they occur. An anomaly usually attempts to compromise the defense mechanism and to degrade the performance of a network. An anomaly may arise due to various reasons. . The purpose of an anomaly detection mechanism is to analyze, understand and characterize blockchain network traffic behavior, as well as to identify or classify the abnormal traffic instances such as malicious attempts from normal instances. From a machine lerarning/AI algorithm perspective, the anomaly detection problem is a classification problem. We adopted various AI technology to have a complete real-time response to catch potential intrusion.



Our major dnomaly detection system is based on probabilistic deep

Learning algorithms such as Bayesian Belief Networks, based on the original work of Bayes and the Dempster–Shafer theory of belief, gives us a mechanism to evaluate the outcome of systems affected by randomness or other types of probabilistic uncertainty. The main characteristic of probabilistic learning is its ability to update previous outcome estimates by conditioning them with newly available evidence.

- Hidden Markov Model (HMM) A hidden Markov model (HMM) is a statistical model where the system being modeled is assumed to be a Markov process with unknown parameters.An example HMM is shown below. For blockchain security, the challenge is to estimate the hidden parameters-based on the observed parameters. Unlike a regular Markov model, where the state transition probabilities are the only parameters and the state of the system is directly observable, in a hidden Markov model, the only visible ele- ments are the variables of the system that are influenced by the state of the system and the state of the system itself is hidden. A hidden Markov model's states represent unobservable conditions of the system being modeled. In each state, a certain probability is associated with producing any of the observable system outputs and correspondingly a separate probability exists indicating the likely next states. By having different output probability distributions in each of the states and allowing the system to change states over time, the model is capable of representing nonstationary sequences. To estimate the parameters of a hidden Markov model for normal system behavior of a network, sequences of events collected from normal system operations are used as training data. Next, an expectation maximization (EM) algorithm is used to estimate the parameters. Once a hidden Markov model is trained and is used to confront test data, probability measures are used as thresholds for identification of network anomalies.
- Gaussian Mixture Model (GMM) The Gaussian Mixture Model (GMM) is a probabilistic learning model. It is a type of density model that is composed of a number of component functions, usually Gaussian. A Gaussian mixture density is a weighted sum of M component densities and is given by the equation:

$$p(\hat{x}|\lambda) = \sum_{i=1}^{M} p_i b_i(\hat{x})$$

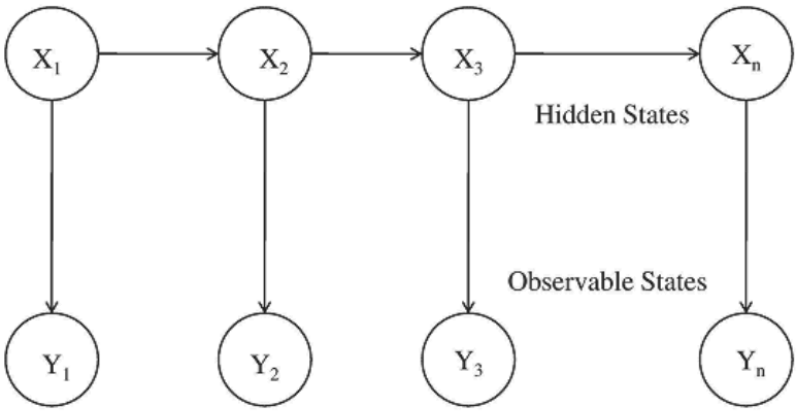


Figure 3: an example of Hidden Markov Model

where $\hat{x}$ is a D-dimensional random vector, $b_i\hat{x}, i = 1, 2, \cdots, M$, are component densities and $p_i, i = 1, 2, \cdots, M$, are mixture weights. Each component density is a D-dimensional variable Gaussian function of the form:

$$b_i\hat{x} = \frac{1}{(2\pi^{D/2}|\Sigma_i|^{1/2})} exp - \frac{1}{2}(\hat{x} - \hat{\mu_i})'\sigma_i^- 1(\hat{x} - \hat{\mu_i})$$

The mixture weights satisfy the following constraint:

$$\sum_{i=1}^{M} p_i = 1$$

The complete Gaussian mixture density is parameterized by the mean vectors, covariance matrices and mixture weights from all component densities. These parameters are collectively represented by the notation:

$$\lambda = p_i, \hat{m}u_i, \Sigma_i$$

For network anomaly detection, each classification of anomalies is represented by a GMM and is referred to by model . The GMM can have several forms depending on the choice of covariance matrices, such as (i) one covariance matrix per Gaussian component (nodal covariance), (ii) one covariance matrix for all

Gaussian components in a model (grand covariance) or (iii) a single covariance matrix shared by all models (global covariance). The covariance matrix can also be full or diagonal. Full covariance matrices are usually deemed unnecessary by the assumption of statistical independence for the mixture components. This allows simplification by using only the diagonals of the covariance matrix. A compact specification in terms of relevant features is derived for each attack type in order to support the detection of that attack. The features are extracted from the sample traffic data for statistical modeling. In the detection phase, the method attempts to identify any significant deviation for each test instance from the stored reference models. Accordingly, the recognition decision determines whether the test instance belongs to normal or any of the four attack categories, as shown below:

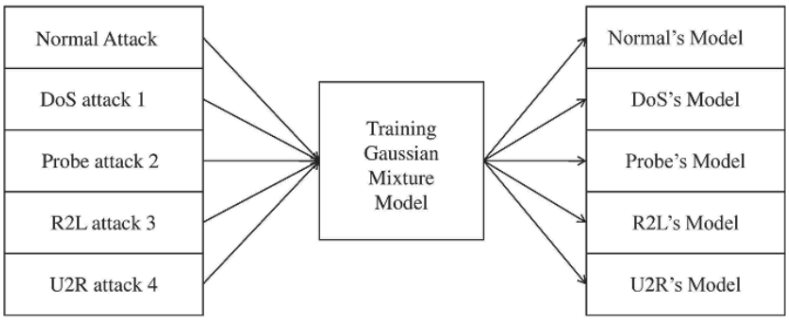


Figure 4: Training phase of the GMM model

– Multiple Bayesian Classifiers As mentioned earlier, the task of DoS detection can be formulated as a pattern classification problem, where the observed behaviors in different dimensions is to be classified as normal or attack traffic. The Bayesian Decision theory is a basic approach used in pattern recognition problems. It assumes the availability of probabilistic descriptions of the underlying features of a problem and aims to find a decision rule which would minimise the risks encountered by the decision taking process. For a two-category classification problem, let us assume we can measure an observation value x for a certain feature, and we have to decide whether the observed data point falls into the normal (w_N) or risky (w_r) category. We can proceed as follows

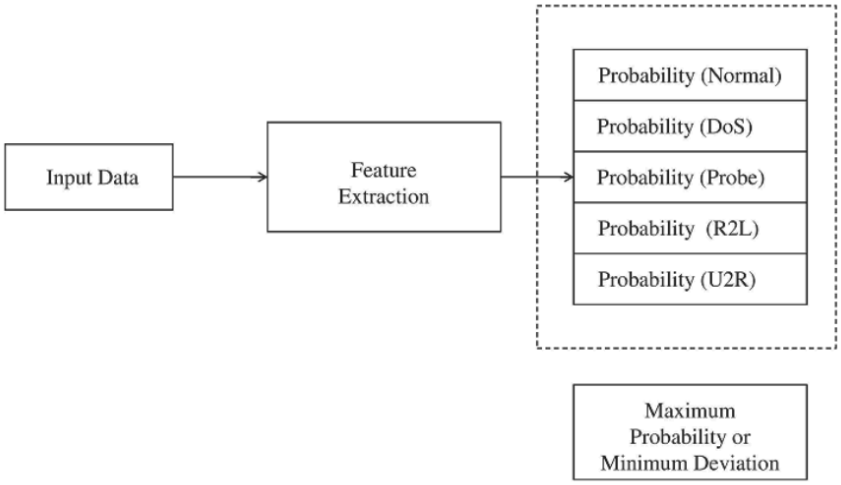


Figure 5: Detection phase of the GMM model

according to Bayesian decision theory:
decide $w_D if \mathrm{P}(\mathrm{w}_r|x) > P(w_N|x); otherwise w_N$
where $P(w_N|x)$ and $P(w_r|x)$ are the posterior probabilities. For evidence f(x) of the specific feature, and prior probabilities P(wN) and P(wD), the joint probabilities according to Bayes rule become:

$$f(w_N, x) = P(w_N|x)f(x) = f(x|w_N)P(w_N)$$
$$f(w_r, x) = P(w_r|x)f(x) = f(x|w_r)P(w_r)$$

Since the evidence f(x) is only a scale factor and can be neglected in computations, we get:
decide $\mathrm{w_r}$ if $\mathrm{f(x|w_r)P(w_r)} > \mathrm{f(x|w_N\ )P(w_N)}$; otherwise $\mathrm{w_N}$
Now we can compute the average risk R, incurred by the costs c_{ij}, the cost of deciding in favour of class i, represented by subspace r_i , when \(j) is the actual class of the data point. For our two-category classification problem, the average risk is evaluated as:

$$R = c_{DD}P(w_r)\int_{rD} fx|w_r dx + c_{NN}P(w_n)\int_{rN} f(x|w_N)dx +$$
$$c_{ND}P(w_r)\int_{rN} fx|w_r dx + c_{DN}P(w_N)int_{rD}f(x|w_N)dx$$

Above, since the first two terms represent correct decisions, we assume $c_{DD} < c_{ND} and c_{NN} < c_{DN}. Also, r = r_D + r_N$ and $\int_r f(x|w_D)dx == \int_r f(x|w_N)dx = 1$. So, the average risk is derived as:

$$R = c_{DD}P(w_D) + c_N NP(w_N) + \int_{rD}[P(w_N)(c_{DN} - c_{NN})f(x|w_N) -$$
$$P(w_D)(c_{ND} - c_{DD})f(x|w_D)]dx$$

We monitor various network behaviors to measure features for decision taking and we utilize multiple Bayesian classifiers to take individual decisions for each of the input features. The collected information is then combined in a fusion phase to yield an overall decision about the risk assessments.

3.3 Anomaly Migration System (AMS)

Smart contracts compiled with our SDK included will automatically include a snapshot mechanism to safely back up key state and information on blockchain. The timing of backup can be either periodical or ad hoc, prior to key access and requests with smart contracts. When failure is epic and rollback is desired, the owner of smart contract need to send a message to our predetermined address with digital signature of the snapshot hash. If verified, the desired rollback happen and state will be restored.

To create the snapshot file, we use the AST from DIS parsing results, modified to use memory-lean hash tables. We then use a shell script to call the appropriate functions on the block parser. We specify the maximum block number as well as the number of addresses to include in the snapshot. Next, the hash160s are converted into addresses[8], which is also written to the snapshot file. After parsing, a *snapshotToImport.txt* file is output containing the balance, hash160, and address. An example entry is 180893019187.00000000 8c1d15231afa4868330f8af694ba637b69fdc2d7 14CKu2rJN2f8 fdPrtmRLWChhXESgN5qaA7.

3.4 Plug'n'Play smart contract runtime and VM.

Our smart contract runtime and VM (ARM-VM) are compatible at op-code level with most mainstream blockchain VMs such as EVM and eWASM. ARM-VM core provides an open and extensible security middle layer. Smart contracts are executed securely in parallel by a high-percentage of the network nodes. The result of the execution of a smart-contract can be the processing of contracts, creating monetary transactions and changing the state of contracts persistent memory. ARM-VM allows security check on DIS, SNS and AMS level for Ethereum/EOS/NEO contracts to run flawlessly. ARM-VM is to emulate EVM by dynamically retargeting opcodes to a subset of LLVM bytecode. All security scanning and immunity mechanism run atop VM to maximize utility and compatibility.

ARM also provides a set of SDK and IDE to minimize the effort to create smart contracts securely. One-click ICO contract code generation is one of the examples which shows the flexibility and capacity of this toolkit.

We aim to target three types of blockchains which are predicted to boast valuation above USD 100,000 billion:

1. Public chains such as EOS, EON, etc.
2. Industrial chains such as CNN、DACC、XRP
3. Federated chains used by governments, corporations and conglomerates.

4 Token Logic

- Armors provides a crowd sourced system for smart contract audit, bug bounty and monitoring. The core concept of the token logic is "audit is mining". Here audit is categorized into three classes: developers providing scripts and tests on exploits and fuzzing analysis, miners who provides blockchain validation and computing resources, autit committee validating the security and availability of code submitted. Combining three layers and roles, Armors aims to create an ecosystem which provides secure, incentivized and democratic security audit.
- Roles in Armors ecosystem
    1. Companies
    2. Miners doing formal verification;

3. Miners doing smart contract operating & monitoring;
4. Armors Committee;
5. Whitehat hackers and smart contract developers;

- Ecosystem overview
  - Smart Contract Code Audit: The Armors smart contract audit will be conducted through formal verification. Armors platform will perform the formal verification computations from random rounds of the "formal verification miner node". The decentralized system will automatically verify the results. The voting process eliminates human intervention and avoids the problem of malicious fraud by individual nodes. The node will obtain ARM tokens as compensation based on its POW computation.
  - Smart Contract DApps Operation Monitoring: For any DApp or smart contract, project teams can purchase the smart contract monitoring and alert service on Armors platform. Armors models smart contract behaviors and raise alerts when abnormal patterns are recognized.

    The service is charged monthly or yearly. Armors will decenteralize this process and provide incentives to miners rewarding ARM gas. The monitoring of the same abnormal event will only take effect when more than 2 different random miners raise alert, to avoid malicious mines.
  - Smart Contract Security Community: In this community, whitehat hackers and developers can obtain ARM tokens by contributing by reporting security vulnerabilities, contributing smart contract code, etc. When there is a dispute over the results, the Standing Committee conducts multiple rounds of random selection of members to vote for decisions.
  - Armors Ecosystem Fund: Armors will work with partners in our ecosystem to organize a fund to compensate partners when their blockchain security is under threat. If the partner's blockchain security is compromised, the Armors platform will organize committee members to conduct multiple rounds of back-to-back voting, reaching consensus on the level of impact for this breach, and issue token compensations based on the reached consensus.
  - Since Armors provides services on different blockchains, ARM Token itself will also be distributed on different public chains, thus covering DApps on different infastructure systems.

5 Ecosystem

- Roles As our outlook for the future of Dapps and blockchain above, requirements for blockchain technology gaining more importance than ever, especially for business users. However, not all businesses are "blockchain technology" ready, severely lacking of blockchain R&D, continuous integration, dev-ops and tech maintenance capabilities. Within Armors ecosystem design, the first priority is the needs of those business users.

  DApp developers, mostly software engineers who write smart contracts and develop DApps based on product requirement documentations. White hat hacker mainly refers to those who grasp the hacking and petentration technology. They perform security intrusion test based on non-malicious usage of hack for hardware and software technology systems. Armors committee is a group of security specialists responsible for the entire security ecosystem. They are from Armors community, mostly responsible for security of transactions between DApp developers, white hats, and DApp users. At the same time, they will also receive rewards for their work. The Armors stakeholders are groups that hold Armors equity certificates (currently is Armors tokens) and hold rights to Armors ecosystem. Those stakeholders have a certain amount of equity in the ecosystem, and at the same time, they will be entitled to claims for assets in DApp development, security upgrade supports and committee election in the ecosystem. The equity of the stakeholders can be normally used freely. After usage, those corresponding claims will be frozen for a period of time, during which claims cannot be traded or reused. After the frozen timeframe, those claims will be reactivated, such that can be used again.

  The design of this mechanism mainly reflects that Armors pays more attention to long-term development of community relationship among Armors stakeholders. The Armors officials, refers to the agents in the entire Armors ecosystem, responsible for Armors Dapp market development & maintenance, community operation, technical infrastructure, research and development and general support for organizations. Armors security committee is the standing team whose main job is to build mutual trust among transactions.

- Role List
  - Fund Initiator: Armors Foundation.

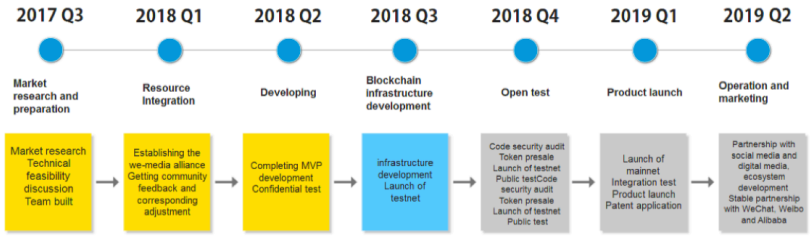
- Fund Type: Charity.
- Fundraising: Limited-partnership, donation.
- Partners: Any stakeholders in the Armors Ecosystem, in particular those seeking security solutions. Using the Armors services. Benefits are measured by the number of shares in the fund.
- Purpose: Whitehat hackers who provide security clues claim for the rights of the Armors ecosystem, security engineers by providing fixes for those threats. The different roles within the ecosystem are incentivatized to work together to provide full coverage for blockchain security issues.
- Fund life cycle: Long-term;
- Fund Manager: Armors Committee.

6 Token Allocation

Armors look forward to issue *10 billion* ARM token. As of publication of this document, Armors has the valuation of *6,666* BTC. We are looking forward to raise 10% (i.e. *666* BTC) on the post-money basis. Current converting rate between BTC/ARM is 1,500,000. We provide the option of *linear* lock-up scheme. After ICO, on each single day 1/365 tokens become available to sell. As discussed in previous section, Armors token are used widely in our ecosystem, including:

- Used for settlements in smart contract transactions in the community market;
- Incentives distributed to contributors in the open source community;
- Used in Armors paid API;
- Use to establish a security fund between armors and ecological partners;
- Incentives for whitehat hackers after submitting vulnerabilities;
- Armors committees for daily operations;
- Used for cross-chain transactions, when DApps on different blockchains use Armors services;
- Other Armors paid-for services, such as code audit services;

7 Roadmap



8 Conclusion

We present the design and implementation of Armors —a complete blockchain security software framework and ecosystem, for dynamic scanning and safeguarding smart contracts. We leverages type theory, Dijkstra monad, abstract interpretation and symbolic model checking, along with the power of CHCs to prevent and detect potential vulnerabilities. We are the first blockchain security firm providing service and development SDKs for smart contracts, aiming to solve one of the top problems of the entire blockchain landscape.

References

[1] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[2] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[3] Review by: A. R. Turquette. Language, metalanguage, and formal systemby haskell b. curry. *Journal of Symbolic Logic*, 18, 09 1953.

[4] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. *SIGPLAN Not.*, 51(1):256–270, January 2016.

[5] William Robertson and Giovanni Vigna. Static enforcement of web application integrity through strong typing. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, pages 283–298, Berkeley, CA, USA, 2009. USENIX Association.

[6] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*. ACM Press, 2013.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[8] Pierre-Yves Strub, Nikhil Swamy, Cedric Fournet, and Juan Chen. Self-certification. *ACM SIGPLAN Notices*, 47(1):571, jan 2012.

[9] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93, 1991.

[10] Bart Jacobs. Measurable spaces and their effect logic. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, jun 2013.

[11] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification*, pages 343–361. Springer International Publishing, 2015.