Thanx @zilm, @jrhea and @mkalinin for discussion and comments.

Executing user defined code during blockchain transactions offer tremendous opportunities. There is growing interest to privacy-preserving cryptography in smart contracts, like Zero-Knowledge Proofs (e.g. zkSNARKs, Bulletproof, zkSTARKs). However, ZKP is CPU-hungry, while safe execution of untrusted computation-heavy code is a serious problem. It's relatively easy to "sandbox" untrusted code, using interpreters (e.g. Ethereum VM), but it is inevitably slow. Compilation could be much faster (e.g. EWASM with JIT/AOT compilers), but it's notoriously difficult to implement in a safe way (e.g.JIT bombs).

Performance-critical code often has to be implemented as core functionality extensions, which should be trusted (e.g. precompiled contracts in EVM). Managing such extensions requires social consensus, which may be difficult to reach. This inevitably slows down innovation.

We propose to solve the problem - fast execution of untrusted computation-heavy code is a safe way - employing machine-checkable proofs. A user who wants to run performance critical code in a compiled form can supply proofs that the code is indeed safe, i.e. doesn't violate safety properties (like affecting other contracts behavior, exhausting resources, etc).

It can be difficult to implement, in general. However, we are limiting ourselves with the domain of cryptography primitives (like hash functions, MAC, HMAC, elliptic curves, pairings, etc). Thus, we can carefully restrict expressiveness of code specification language, so that its analysis and verification ideally can be done automatically. Given that core cryptography primitives can often fit CPU caches, when loops are unrolled, it's a reachable goal.

The approach can be extended to other performance-critical code, which users may want to run on-chain, e.g. asset pricing, numerical optimization, etc. A more expressive language and a more complex verification technology may be beneficial or even required. We therefore propose a modular approach, which components can be replaced with heavier and more expressive, if needed.

However, our main target is to allow easier introduction of modern cryptographic protocols into Ethereum VM smart contracts. Therefore, we make certain assumptions, to simplify further reasoning.

Motivating examples

zkSNARK friendly hashes

Privacy is a critical issue and there is growing interest to ZKP approaches like zkSNARKs, etc. However, zkSNARKs can be quite expensive to run in EVM. E.g. Nightfall paper reports numbers about 2-3M gas cost per transaction. Nightfall is using SHA2-256 hash function, which is cheap to evaluate in EVM, but is not zkSNARK friendly, leading to many constraints created for a proof.

There are zkSNARK friendlier functions, likeMiMC and Poseidon, requiring two orders of magnitude less constrains (e.g. 50x and 100x less, see, for example, comparison here). Unfortunately, the functions are expensive to compute in EVM, requiring orders of magnitude more gas than SHA2-256 (e.g. 150x and 1000x the same paper).

The problem is that SHA2-256 is implemented as a highly optimized precompiled contract in EVM, so it's assigned with a discounted gas cost. While, MiMC and Poseidon are to be implemented using interpreted EVM instructions (and available precompiled contracts). While, when implemented in a compiled language, they can be computed much faster, though still slower than SHA2-256. For example, measurements from here reports MiMC and Poseidon are about 2.5x and 30x slower than SHA2-256.

So, if MiMC and Poseidon can be implemented as precompiled contracts, they can be assigned with much lesser gas costs.

Elliptic Curve cryptography

EVM have precompiled contracts for bn256 elliptic curve (ECADD, ECMUL, pairing check). However, there is interest in adding faster and more secure elliptic curves, like BLS (pairing-friendly) and secp256k1 (e.g. for non-pairing applications), see, for example, here.

Usage scenarios

There can be several ways how verifiable precompiled contracts can implemented from organizational point of view. Let's consider a couple of scenarios. Our goal is to illustrate how the idea can be used to solve practical problems, rather than to propose a complete approach.

Fully-mechanized

A user-defined precompiled contract (UDPC) can be defined with a special operation or contract, during a transaction execution. Once participants verify the UDPC, compile it and reach consensus that it's safe to be run in a compiled form,

smart contracts can invoke the UDPC. As being compiled, UDPCs run much faster, UDPC gas can be made much cheaper (proportional to performance gains).

It will be much easier and cheaper to execute privacy-preserving protocols on-chain, since an implementer can choose zkSNARK friendly function and use faster implementations of faster elliptic curves (e.g. BLS). Even, though using new hash functions like Poseidon or MiMC can be risky, it can be acceptable for some smart contract developers.

There are several problems associated with the scenario. it can be hard to implement in general. However it's much easier for restricted languages, which are limited in their expressiveness (but enough to code cryptographic primitives).

Since, UDPCs are to be verified and compiled on-chain, both verification and compilation phases should be reasonably fast and resource bounded. As with smart contracts, resource consumption of verification and compilation should be predictable, so that the procedures be deterministic.

If users are free to define their own operations, the nodes executing transactions are required to keep lots of binaries. Therefore, introducing a new UDPC incurs costs to other participants. Though, new precompiled contracts can be beneficial to some participants, costs are incurred to everybody. The problem can be relaxed, if defining a UDPC incurs significant cost. However, defined UDPC can be used by many users, so it's a free-rider problem.

It becomes a (public) economy problem then, so we won't discuss it further in details. Just note that it should be resolved somehow, if the scenario is desirable. For example, participants can vote for a precompile (with deposits). The access to a UDPC can be restricted for those who have paid for it.

Partially-mechanized

Since running compilation and verification phases on-chain can be tricky, a UDPC submission can be accepted via a social consensus to extend EVM core functionality. The consensus can filter out UDPC which do not seem to be valuable to the community.

However, the procedure of accepting new UDPC can be greatly simplified, if UDPC verification and compilation is mechanized. This will reduce human efforts to review code and implement it in a fast, consistent and safe manner.

To reach that, a UDPC submission is augmented with gas usage formula, formal proofs of safety (including proof that the formula is valid). Mechanical proof checker can be used to check the proofs of safety properties. The UDPC specification can be compiled to a reference implementation then (e.g. in C).

There can be also several compilers of UDPC specification to other targets (C++, Rust, LLVM, WASM, Java, etc).

With the scenarios, after social consensus accepts the UDPC, EVM implementers need to run proof checker and compile the code. Compilers to non-C targets and bindings to invoke the resulted code should be developed too, but the effort is amortized over many UDPCs.

The scenario is much less risky, since it just simplifies decisions and efforts to be made by a governance committee and implementers.

Hybrid approach

There can be a hybrid approach, where a social consensus is implemented over Byzantine Agreement protocol. I.e. anyone can propose a UDPC, however, it cannot be invoked immediately, even if it's well-formed and can be proved to be safe.

Implementers should first implement it in their software (that can be as simple as verify and compile the UDPC to a dynamic library) and users who run the client, should upgrade their software (that can be implemented as the dynamic library loading).

After a quorum is reached, a decision to accept the UDPC can be made and rest of clients have certain period of time, to upgrade their software, before the UDPC is enabled in EVM and can be actually invoked by smart contracts.

A committee may veto the decision too.

UDPC Gas-pricing

As UDPCs are to be specified via a formal language, UDPC gas can be defined and assigned for each specification language instruction. As we are targeting cryptography applications, most instructions will be simple: integer and boolean operations, load/store instructions, branches.

A suite of cryptographic primitives can be implemented and compiled to C.

Then performance of the primitives can be measured and gas pricing formula can be deduced, using statistical methods (e.g. group instructions by type and cryptographic primitive, count them and regress on measured execution time). The

UDPC gas pricing formula can be correlated with EVM gas, by comparing with EVM cryptographic primitives implementations, like SHA256, etc.

Roles and Concepts

Let’s define core roles and concepts more formally.

A Submitter

wants to evaluate some code inside EVM, in a compiled form. We call it UDPC

- User-Defined Precompiled Contract, by analogy with EVM core precompiled contracts.

An Acceptor

won’t allow executing untrusted code in such a way, however, the Acceptor

agrees to do that, if the code is actually safe, i.e. certain safety criteria are met. The criteria are dubbed Safety properties

. They can be split in two groups: code execution is sandboxed (cannot affect other smart contracts, etc) and code consumes limited amount of resources (CPU cycles, memory, etc). For a compiled code, resource consumption bounds should be known in advance, as it’s more tricky to abort its execution.

To convince the Acceptor

, the Submitter

provides an argument that the code is safe, along with the code itself. We dubbed the argument Proof

, as we are aiming mechanically-checkable proofs.

The Acceptor

checks the Proof

, i.e. verifies that the UDPC

meets the Safety Properties

. If everything is okay, the Acceptor

allows execution of the UDPC

in a compiled form.

In a heterogeneous system, when there are multiple implementations of EVM on different processing architectures, the problem is more involved. Implementers

should implement the UDPC

, so that it can be run on their particular architecture in a compiled form. It can be a non-trivial problem and consume lots of Implementer

’s resources. That’s why, in practice, an Implementer

is often a stakeholder in the acceptance process, i.e. the Acceptor

role is a consensus of a committee.

So, in general, Safety properties

should include a property that Implementer

’s efforts will be reasonable and justified. If implementation is a fully mechanical process (i.e. compilation and linking of a UDPC

), a malicious Submitter

can in general construct UDPC

code in a way, that it will exhaust computer resources during the compilation phase (e.g. JIT bomb). Since these are resources too, we extend Security properties

with a criteria that Implementer

’s efforts are reasonable too.

Mechanizing Actors

The above model holds when both humans and computer can be actors. Currently, the precompiled contracts governance is not mechanized and acceptance of a UDPC

submission can take serious efforts of many people. A consensus can be problematic to reach, since implementations of cryptographic primitives in a consistent and performant way over multiple languages and processor architectures can be a serious problem.

We therefore aim to (partially) replace human decisions with formal procedures, i.e. make actors more mechanized. While it can be very resource-consuming in general, the modern state of software verification of cryptographic primitives allows for a relatively simple solutions.

Mechanizing Acceptor

means a mechanical Proof checker

procedure should be defined, which in turn, means formalizing Safety properties

, UDPC

specification language and Proof

. In general, Acceptor

need not be fully mechanical, we aim to lower human efforts involved, though fully mechanical Acceptor

can be useful too.

As Submitter

is required to supply Proof

, constructing it can be a problem too. It cannot be fully-mechanized in general. However, if the specification language is restricted enough, proof-construction procedure can be fully mechanized. We believe it’s the best strategy in the case of cryptographic primitive precompiled contracts. Though, it can still be beneficial to have semi-automated procedures (shorter proofs, code maintenance, etc).

Mechanizing Implementor

is one of the most important tasks. As we are aiming to have a restricted UDPC

specification language, translating it to a C, Java, Rust, LLVM, WASM or other popular targets is not a problem. Optimizations are also required. Cryptography primitives are often implemented with assembler, to maximize performance. Though, this work shows that mechanically generated code can be competitive with manually optimized C and asm code. To accommodate for optimizations we distinguish UDPC

specification and implementations. There can be several UDPC

implementations with different control flow graphs, reflecting optimizations performed. They can be proved to conform to the UDPC

specification (which we assume to be executable too). So, Implementer

can measure performance and choose most suitable. Or create own implementation.

Verification technology notes

While cryptographic algorithms are often specified in a parametric way, to accommodate varying needs, in practice, users typically use particular combinations of parameters, which are often standardized.

So, practical cryptographic primitive cores are typically implemented with loops with fixed bounds. While many cryptographic primitives can accept variable-length inputs, a fixed-size input core procedures are typically invoked repeatedly inside of an outer loop.

Since, EVM enforces gas limits, such restricted languages are pretty reasonable, in general too (seeVyper, for example).

This means, there are many opportunities to define a restricted language, which is expressive enough to specify practical cryptographic primitives, while being amenable to fully mechanized analysis.

Simply speaking, the fixed-bound loops can be unrolled, leading to a finite acyclic control-flow graph. Such a graph has a finite amount of possible execution traces (paths from an initial to a final state), each consisting of finite steps (without conditional branches). Thus, assertions can be checked in a finite time.

While in theory, such graphs can be huge, the cryptographic primitives should be fast and they have lots of repeated activity, so optimizations of such analysis is possible and already implemented in many SAT/SMT solvers and model checkers.

Conclusion

The goal of the post is to provide a high level overview of verifiable precompiled contracts idea. A more detailed technically involved WIP write up can be found here.