This is a follow up post to Verifiable Precompiled Contracts to address issues that have been identified during a discussion with Casey Detrio and Alex Beregszaszi.

Modern cryptography like Elliptic Curve and ZKP is highly desired to be run inside blockchain transactions. However, this is a hard target. Blockchain client implementers are conservative about new bits of codes and this behavior is highly justified: software bugs are ubiquitous and bugs can result in direct losses or render system unusable (e.g. in-determinism can break consensus). Careful audit is required, which is labor-intensive.

The problem is amplified when compiled execution is required, which is the case since performance is critical for ZKP and many other cryptography algorithms.

To solve the problem, one may employ formal verification, which can guarantee absence of some classes of bugs, though it is labor-intensive too. Another consequence is that introduces new code to trust.

This is a difficult task, which lies on intersection of verification and compilation technologies, both of them being far from trivial.

In the post, we discuss opportunities to optimize overall efforts to sandbox compiled execution in the domain of cryptography primitives. In many cases, codes in the domain may be relatively easy to analyze, since restricted languages and formal models can be enough.

The analysis of such languages and models can be orders of magnitude easier and often allows implementation of push-button approach to verification.

Overview

Sandboxed compiled execution

It's relatively easy to sandbox interpreted execution of untrusted code: any action can be checked for undesirable consequences. In the case of compilation, such checks can be inserted too, but they will inevitably slow down the code.

Such checks can be eliminated, if a compiler can prove that the corresponding state can never be reached. It's not possible, in general, since the problem is intractable for Turing-complete languages.

However, necessary arguments can be supplied by a user, which means Formal Verification should be introduced. Formal Verification is notoriously difficult, however, there exists certain classes of programs, when the problem is decidable and even tractable. Cryptography is the problem domain, where programs often fall into such classes. This is due to specific requirements to cryptography primitives: a lot of analytical efforts have been applied to study them. Additionally, cryptography primitives should often be fast and easy to implement, including hardware implementations. That particularly means standardized parameters, which results in fixed loop bounds.

Overall, structure of practical cryptography codes should be amenable to human and mechanical analysis.

Compilation and verification phases thus can be adapted to simplify efforts to develop cryptography code and necessary proofs. In many cases, push button (i.e. fully automatic) verification can be achievable.

In cases, where push-button approach is not applicable to the whole program, one still can apply push-button approach to verify parts of necessary verification conditions.

The rest of properties can be enforced with runtime checks. If the checks are outside of busy inner loops, then performance slow down will be modest. We believe, such approach fits very good to most of the practical cryptography codes.

Trusted Computing Base

One consequence, however, is that such adaptation of compilation and verification methods increases the size of Trusted Computing Base (TCB).

In order to adopt such technology, blockchain implementers should trust it. How one can be sure that such optimizations do not break safety properties?

As have been pointed out before, Formal Verification being very difficult topic, the necessary trust can be hard to obtain.

In fact, the trust problem is arguably the key one and the center of the post. What are the ways to tackle the problem?

Full stack verification

Nowadays, more or less full stack verification, i.e. very small Trusted Computing Base, is achievable (see for example, here). One can choose enough tools to develop a software and produce proofs that the software execution results conform to the respective specification on every level: implementation, intermediate and target machine languages, sometimes, even hardware level, in some cases.

The proofs can be checked by a mechanical proof-checker. So, the only pieces of software to be trusted are:

- the proof-checker code, which is relatively small and can be inspected by humans
- final target language semantics (e.g. assembler language or H/W semantics)
- code specification, which has to be inspected in any case

Hardware is still to be trusted though. Although, there is considerable progress in H/W verification, we ignore H/W aspects for now, since we believe we don't have much choice here. E.g. proof checker has to be run on some hardware and we often have to use some hardware we have to trust, despite we know there can be vulnerabilities. We, thus, abstract away from H/W, by assuming that there exists an appropriate H/W specification.

It's hardly possible to minimize TCB further. However, proof-checker and target language semantics are to be inspected only once and are amortized. Moreover, proof-checkers are already audited by world class verification experts. The same applies to lesser degree to target language semantics.

Optimizing TCB

As usual, full stack verification can be very labor consuming. One problem is that it can lack necessary proof automation available with other approaches. Our goal remains hard, since necessary level of trust could be reached, but requires so much efforts, that it is rarely performed in practice.

We therefore treat minimal TCB/full stack verification approach as an ideal, from which one has to depart partially, given a constraint budget.

In practice, efforts to formally verify a piece of software increase trust in it, even if tools cannot be completely trusted. This is especially true for widely known and recognized tools that decades of efforts have been put into their development (like Z3, Why3, etc).

One case is that practical systems, including cryptography codes and blockchains, in most cases are implemented using regular non-certified compilers, despite it's known that such compilers cannot be fully trusted. For example. C compilers often introduce bugs when optimizing codes, relying on overflow behavior of operations. C99 standard doesn't completely specify behavior of some integer operations.

We therefore believe that tools which have a long history and are widely used are acceptable and can be trusted in practice. Given that known weaknesses are avoided (as prescribed by MISRA C standard, for example).

Improving tools

Initially, such simplifications are necessary anyway, to limit efforts to develop a prototype solution. However, in a longer run, known problems can be addressed by development of appropriate tools.

One problem is that a full stack verification solution can be limited in a ways to automate proof development. E.g. push button verification is not readily accessible. As we are targeting classes of programs that are amenable to mechanical analysis, such automation can be developed.

It's relatively easy to achieve is a class of program which do not contain loops and recursion (or which can be translated to such form). Many cryptography codes falls into such category. So, we believe, the problem is solvable, in our case.

On the other side, a verification framework, which has good proof automation suitable to our needs, may lack abilities to generate necessary proof-terms, which can be integrated into the full stack verification approach. However, some provers still can emit some proof terms, so, in principle, necessary tools can be developed, to convert the terms, so that they can be passed to the main proof-checker.

Verification tool developers also recognize the problem and work on improving their tools, so that they can be trusted to a higher degree.

Improving trust in codes specifications

Specification of a cryptography codes is to be trusted too. This is inevitable, however, efforts to audit such codes can be optimized.

There are two primary aspects, that a specification should cover in our case:

- safety properties that are to be enforced
- at least one implementation is required

Safety properties specification

The safety property specification belongs to TCB too. It will be very resource consuming, to specify and to inspect it for each submitted cryptography code.

So, a common safety property specification should be developed.

The specification can be applied to any new code in a standard way: places, where the properties can be violated are identified and necessary verification conditions are generated.

In the cryptography primitives domain, we may restrict input languages, so that such places are easy to identify. In fact, we believe such conditions can be reduced to three cases:

- array bound checks
- resource consumption checks (execution steps and memory allocation)
- non-determinism guard checks (e.g. C99 doesn't fully specify behavior of some integer operations)

Other safety properties can be enforced during compilation (e.g. compiler can check that a code accesses only a restricted set of arrays and invokes only a trusted set of functions).

The main property which remains to be specified is a gas usage formula. In general, it can be inferred too, however, a user may want to supply a tighter bounds.

Equivalence proofs

With respect to a reference implementation, there are two important cases:

- a specification is executable, so one doesn't have to prove anything and can compile the spec
- multiple implementations are desired (e.g. optimized ones), so one has to prove they conform to the specification

We assume it's natural to assume for cryptography that a specification is executable. Therefore, a typical proof should be an equivalence proof, i.e. a proof that two programs produce the same results for every possible combination of input arguments.

Often the reason to add a new code will be to implement a well known algorithm, so a reference implementation may already exist. One can translate such implementation either manually or mechanically in an accepted specification language.

The equivalence proof technique can be used to verify that the specification and the implementation agree on every possible input, so that a trust in the well known reference implementation can be transferred to trust the new formal specification too.

In fact, equivalence proofs is a important practice in modern cryptography code development, so there exist tools to aid the process (see, for example, here).

Target language assumptions

Compilation is often performed in several steps:

- an implementation in a domain specific language can be translated to a general purpose language, like C, Rust, etc
- general purpose language translated in an assembler language, which often is performed using intermediate representations too
- a program in assembler language is translated into machine code

In general, every translation step can introduce a bug, so verified compilers should be used, if full stack verification is required. There exist certified compilers, like CompCert, however, there can be licensing and performance limitations, which may be unacceptable.

We therefore make an additional simplifying assumption, which is arguably acceptable for the problems, we are targeting to solve. We assume that our final target is not H/W, but a (sub)program in a subset of an assembly language.

The assumption is not critical, as the same approach can be used to incorporate it, though at higher costs.

One reason to introduce such assumption is that we don't have a control over the final code deployment, so we have to stop at some point, anyway.

Thus, two additional things to be trusted (in addition to a proof checker):

- Assembler language specification(s)

- Assembler and linker tools, language bindings, etc

We believe these are reasonable costs, since it's often the case that practical cryptography algorithms are written in an assembler language (one case is to avoid bugs which can be introduced by optimizing C compilers).

C language as a target

Assembler code, especially, heavily-optimized is difficult to call human-readable. Probably, it's the reason why many critical code is still written in C, that applies to cryptography codes too, despite known problems with C standard and compilers (like undefined behavior in some cases, bugs when optimizing code which relies on overflow behavior, etc).

We therefore consider C as an acceptable or even required final target language. it can be difficult to gain trust, when people cannot understand code.

Certain measures, however, should be applied to prevent known problems. Such measures have been already developed (e.g. MISRA C standard). We need to adapt it to our case, that means target language should be a carefully chosen subset of C (e.g. C99 language standard), which avoids known pitfalls.

Since there are plenty of mature C compilers, including a certified one, we believe that using C as a final target language helps to increase overall trust.

Practical considerations

Verification framework examples

Full stack verification

A notable example of full stack approach is VST, based on Coq proof assistant:

- VST and Verifiable C
- CompCert certified compiler

There is a similar solution, based on HOL system - CakeML.

Lighter frameworks

There are lighter verification frameworks, which lack the explicit proof-checker, e.g. one has to trust proof-solver. However, the approaches are more popular and arguably more flexible. They often allows to use external provers and SMT solvers, which can discharge many verification conditions automatically. Therefore, proof development is greatly simplified, so that even push button verification is possible in certain cases.

- MSFT infrastructure around Z3 SMT-solver:
- Z3
- Boogie
- Dafny, Spec#, F*
- Z3
- Boogie
- Dafny, Spec#, F*
- Why3 framework
- Viper infrastructure

How formal verification can be simplified

fixed-gas fragment

We need proofs to eliminate runtime checks, which hinder performance. However, such checks are critical inside busy loops mostly.

In practice, inner loops of cryptography primitives often have fixed bounds, since they are implementing cryptographic algorithms with standardized parameters, like bit length, etc. It’s often required to make high performance implementation possible. So, it’s quite common in practical cryptography codes.

If loop bounds are fixed, then such loops can be unrolled, which means the analysis of such programs is greatly simplified (decidable and even tractable in many practical cases).

simple linear-gas fragment

Many cryptography algorithms are linear with respect to input size, e.g. hash functions or encryption/decryption, i.e. there is no fixed upper bound on execution steps. However, they often split input in chunks and call fixed-gas code for each chunk.

So, despite having a variable bound loop, its structure is simple and may be amenable to a fully automated analysis.

polynomial-gas and other difficult cases

It’s critical to optimize inner loops, outer loops are more forgiving from performance perspective. Therefore, a practical strategy can be to recognize fixed-gas and simple linear-gas fragments and to eliminate safety checks for such fragments only.

A user can spend efforts to produce proofs for harder fragments, on his/her discretion, if he/she finds this important to increase performance.

Resource constraints

One of the critical problems is to ensure that a compiled code do not exhaust resources, like processor time and memory (either in form of stack allocation or dynamic memory allocation).

We require that there exists worst-case execution bound formula and code can be proved to respect the bounds. Memory constraints can be enforced in a similar way.

A straightforward way to implement resource bound constraints is to put resource accounting instructions and checks into code directly. Then it’s easy to prove that it’s respect bounds. However, such code can be slow.

We can treat the problem in a similar way as array bound check elimination - such checks can be eliminated, if it can be proved that it’s safe to do so.

For example, if there is a block of code without brunch instructions, then it’s enough to put resource check and necessary accounting at the beginning of the block only. A slight modification allows to handle unconditional forward branches too.

Similarly, if there is a conditional forward branch instruction, one can check for a maximum possible resource usage, considering every branch.

Loops with fixed bounds can be unrolled, so can be treated the same way.

Simple linear-gas loops, when there is a simple metric decreasing after each loop iteration can also be handled relatively easy.

One can leave runtime checks for polynomial-gas and other more tricky cases.

Compositional proofs

It’s natural to split large programs in sub-components. If a program to be verified calls other modules, such modules can be replaced with corresponding module specifications. As a result, verification is split in several sub-tasks, which can be much easier to do, since each part can be much simpler then the whole.

Such approach matches the above mentioned check elimination strategy quite well.

For example, if a method A calls method B and our goal is to prove that necessary properties cannot be violated, so that a compiler can be instructed to remove the corresponding runtime checks. If it’s proved that the method B cannot violate the properties, given any possible input, then during verification of the method A, one doesn’t need to take care about method B call.

However, results of calling the method B can affect the method A in other ways, so a proper specification of B is required still.

Verification and compilation interaction

Summarizing above, we propose the following implementation strategy.

To ensure sandboxed compiled execution of untrusted code, such code is instrumented with corresponding runtime guard checks

:

- array bound checks
- non-determinism checks (e.g. when behavior is not completely specified)
- resource accounting and resource constraint checks

To improve performance of compiled code, such checks need to be eliminated, so corresponding verification conditions are generated.

Verifier tries to recognize automatically analyzable fragments and discharge such conditions automatically. If it’s successful, the corresponding checks are eliminated.

If it’s not possible, the compiler can emit a warning, so that a user can take appropriate actions, like supply necessary hints, lemmas, restructure code, suppress the warning, etc.

Gas-formula inference

In some cases, gas-formula can be inferred automatically, given a user template, e.g. a fixed-gas fragment or simple linear-gas fragment like a + b *

, where a

and b

are parameters to be inferred and

is some input variable or an expression depending on input variables.

We assume that final versions of top-level functions should have an explicit gas usage formula. However, such inference can be helpful during development and is acceptable for auxiliary functions, which are not invoked directly by end users.

Target program class

Since we are going to optimize efforts, we should restrict input language to simplify development, but so that target class of programs is still expressible.

In the case cryptography primitives, we assume that the following restrictions are natural:

- integer, bitwise and boolean operations only, no floating point operations
- that may include cryptography and vector extensions of some processors
- that may include cryptography and vector extensions of some processors
- exceptions are not allowed (no null de-references, overflows, divide by zero, etc)
- necessary guards are inserted, if an instruction can throw an exception
- necessary guards are inserted, if an instruction can throw an exception
- deterministic results are specified for every case
- necessary guards are inserted, if an instruction behavior is not fully specified
- necessary guards are inserted, if an instruction behavior is not fully specified
- no pointer arithmetic
- can access arrays only
- array bound checks are inserted
- can access arrays only
- array bound checks are inserted
- can read and write a restricted set of memory regions:

- read input parameters
- write output result
- read and write own stack memory
- read and write own dynamically allocated memory, if allowed
- read input parameters
- write output result
- read and write own stack memory
- read and write own dynamically allocated memory, if allowed
- memory allocation
- allocate stack memory (known bound per stack frame)
- optionally, can allocate static memory (during initialization, known bound)
- optionally, dynamically allocated memory can be allowed (checked for resources or known bound)
- resource checks are inserted
- resource checks are inserted
- allocate stack memory (known bound per stack frame)
- optionally, can allocate static memory (during initialization, known bound)
- optionally, dynamically allocated memory can be allowed (checked for resources or known bound)
- resource checks are inserted
- resource checks are inserted
- recursion and mutual recursion is prohibited (with possible exemption for tail recursion)
- can invoke:
- own subroutines
- a restircted set of functions and procedures:
- verified precompiles, mini-precompiles
- allowed trusted set of system calls
- verified precompiles, mini-precompiles
- allowed trusted set of system calls
- own subroutines
- a restircted set of functions and procedures:
- verified precompiles, mini-precompiles
- allowed trusted set of system calls
- verified precompiles, mini-precompiles
- allowed trusted set of system calls

Program kinds

NB

The idea that lighter variants like mini-precompiles and opcodes can be defined was originally suggested byCasey Detrio during our discussion.

Here we describe how the idea can be detailed in the context of the verified precompiled contracts framework.

There can be several kind of newly defined code, some of them requiring special treatment.

One reason is to reduce disk space usage, i.e. some common operations can be reused instead of being inlined by several precompiled contracts.

The other reason is that one may want to reduce invocation costs for some type of operations, e.g. avoid copying memory buffers, etc.

opcodes

Opcodes are considered as a part of VM, so very strict requirements and harder restrictions are applied. We assume they are not 'user-defined'.

A typical set of restrictions can be:

- known worst case execution bounds (fixed gas), which is rather small
- can invoke only a trusted set of system calls, designated to be invoked by opcodes
- can allocate memory only using specially designated subroutines
- no recursion
- fixed loop bounds
- higher level verification requirements (e.g. full stack verification)
- all necessary verification should be decidable and tractable
- guard checks

should be eliminatable

- guard checks

should be eliminatable

In exchange, invocation of opcodes is optimized. An example can be EC curve operations, big integer operations, etc. Opcodes can be used by regular smart contracts, so that their performance can be improved. As there may be growing interest to add new opcodes, a verification and compilation framework can be extended to support verifiable opcode development.

mini-precompiles

Mini-precompiles are similar to opcodes, however, they can be user-defined. The intention is to have subroutines, which can be shared, to reduce disk space and memory usage (e.g. to avoid inlining parts of code, which can be shared between functions). We assume that invocation costs are less than for 'full' precompiled contracts, in exchange for stricter requirements:

- known worst-case execution bounds (fixed-gas)
- simple linear-gas bounds can be allowed too (e.g. a simple loop, decreasing linear metric)
- simple linear-gas bounds can be allowed too (e.g. a simple loop, decreasing linear metric)
- can only call other mini-precompiles (plus specially designated EVM subroutines)
- cannot allocate memory dynamically (it's expected that a caller will provide necessary memory buffers)
- all necessary verification should be decidable and tractable typically (e.g. with longer timeouts)
- guard checks

should be eliminatable

- guard checks

should be eliminatable

For example, mini-precompiles can be core inner cycles of hash algorithms or encryption/decryption algorithms.

precompiled contracts

These are usual precompiled contracts with the regular invocation costs.

- can call mini-precompiles and other precompiled contracts
- verification can be semi-decidable or intractable, in general
- can be allowed to allocate memory buffers (subject to resource constraint checks)
- guard checks

are allowed to remain

- explicit gas pricing formula is required
- can be enforced with runtime checks, e.g. partially, if a verifier is able to discharge some verification conditions
- can be enforced with runtime checks, e.g. partially, if a verifier is able to discharge some verification conditions

Safety properties and conditions

Which conditions and properties should be enforced to allow safe/sandboxed execution of untrusted code?

In the context of execution of cryptography in blockchain transactions, we assume the following safety properties:

- other software components cannot be affected, other than in an allowed way
- can read only specified memory regions (input parameters, own memory)
- can write only specified memory regions (own memory, output result)
- can invoke only specified set of routines
- can read only specified memory regions (input parameters, own memory)
- can write only specified memory regions (own memory, output result)
- can invoke only specified set of routines
- deterministic execution
- all steps produce deterministic results
- resource usage can be enforced within pre-specified bounds
- either predicted before execution or accounted for during execution and aborted, if necessary
- either predicted before execution or accounted for during execution and aborted, if necessary
- all steps produce deterministic results
- resource usage can be enforced within pre-specified bounds
- either predicted before execution or accounted for during execution and aborted, if necessary
- either predicted before execution or accounted for during execution and aborted, if necessary

Resources:

- execution steps as measured by some metric (e.g. gas)
- dynamic memory allocation
- stack allocation or stack depth

Most of access properties can be enforced during compilation:

- can only access regions of memory that are passed as parameters or during initialization
- pointers can be forbidden or limited
- only static set of subroutines can be allowed

One can restrain programs from explicit dynamic memory allocation, i.e. necessary memory can be provided either by a caller or by system (in pre-specified amounts).

Stack allocation bounds can be enforced by:

- forbidding recursion (with the possible exception of tail-recursion)
- stack limits for functions

Therefore, maximum possible stack size can be calculated before code invocation.

As a consequence, only verification conditions corresponding to array access bounds and execution step constraints should be proved.

NB

As non-deterministic cases and exceptions are required to be properly handled, corresponding verification conditions may be required to be proved too.

NB

If the execution model is relaxed, more properties are to be proved (like pointers cannot escape, etc).