Trust and Verify: Formally Verified and Upgradable Trusted Functions

Marcus Birgersson
KTH Royal Institute of Technology
marbir@kth.se

Cyrille Artho
KTH Royal Institute of Technology
artho@kth.se

Musard Balliu
KTH Royal Institute of Technology
musard@kth.se

***Abstract*—Computation over sensitive data requires that the computation function is secure and trusted. Existing approaches either do not enforce formal verification, require the user to verify the proof, or lack secure attestation guarantees. In addition, neither addresses the issue of having users once again inspect the application after upgrading the code running in the enclave.**

We propose an approach that uses a formal specification to guarantee that the behavior of the computation function conforms to the desired functionality. By combining automated verification with attestation on a trusted execution environment, we ensure that only conformant applications are executed. At the same time, we allow updates of the computation function without changing the attestation response, as long as the formal specification still holds. We implement and evaluate the system on several functions; our results show an average overhead of only 50 %. Finally, we demonstrate the validity of the system using a real-world application, Dafny-EVM.

I. Introduction

Outsourcing computations to a cloud environment requires trust in the remote computation, because such services in general, lack support for attested computation. There is no way to get any guarantees on how the data is managed, and one must trust the provider and that the service only does what it is supposed to.

Smart contracts [39] can provide some guarantees because their transactions are publicly visible on a blockchain [43] and thus open to inspection. However, the public nature of blockchain data makes smart contracts a poor fit for the computation of sensitive data; furthermore, smart contracts cannot be directly upgraded.

In general, there are two ways of performing confidential computations in the cloud: 1) using homomorphic encryption and 2) using a Trusted Execution Environment (TEE) [36]. Since the former suffers from high computational overhead [1], the only way to practically achieve confidential computation in the cloud is by means of a TEE. In addition, many cloud services manage confidential data from multiple users [7], [8]. To ascertain the desired functionality, each user must individually verify the code running in the TEE, which can be cumbersome. If the implementation ever changes, even just slightly, each user must again perform this verification process. Thus, classical settings for multi-user computation based on TEEs suffer from the following drawbacks:

1) *Trust in provenance rather than properties.*
   Rather than trust a specific entity, by verifying a signed commit or similar, one should trust that certain properties of the code are true.
2) *Validation* of code or proofs *rather than (automated) verification* of properties, and
3) *No support for upgrades.*
   Every time the code changes in a TEE, each user must once again verify the upgraded application.

To provide a solution for these problems, we present TRUVALT (TRUst and VALidation system in a TEE), a framework to enforce verification of functions running inside an enclave. TRUVALT solves the three shortcomings above by (1) using design by contract [25] to agree upon a *specification* of properties rather than trusting a specific entity, (2) using the automated verifier Dafny to enforce these properties, and (3) treating the implementation as data to allow changes in the implementation as long as the specification of properties still holds.

In the classical setting, when several users should decide to allow their data in a multi-user computation, all users start by inspecting the code to be executed and checking the integrity of the software, typically by comparing it to a cryptographic hash value. One of the users then deploys this code to the TEE, by means of remote attestation, all of the other users can verify that the same code previously inspected is the one running in the TEE. This setup could be cumbersome, and each user must validate the benign intent of the code and also verify its correctness (or delegate this task to a trusted party).

Our contributions address these limitations: We address the first fundamental limitation by applying formal verification rather than relying on the provenance of software (with methods like software bill of materials [44]) or manual code audits. By using an automated formal verification tool, for example Dafny, the code can be formally verified to be correct in terms of a given specification. Each user can run Dafny on their copy of the code, hence the manual work of verifying correctness has decreased. However, without providing verification as a service, every user must run Dafny on their own machine.

The second limitation is addressed by ensuring that only verified code is executed. We achieve this by running Dafny inside the TEE, taking advantage of its attestation to ascertain that all code is verified. Each user now only has to validate the specifications enforced by Dafny, as well as that the Dafny binary included in the TEE is the same as the official version. Assuming that the specifications for each Dafny function are well-written, it is no longer necessary to manually inspect the *implementation*, only the *specification*. Users can be assured

that Dafny will only allow for implementations that adhere to the specifications.

The third contribution allows for automated upgrades. Normally, attestation in a TEE is tied to the executed code, so the hash value changes with each upgrade. In our method, we keep only the *specifications as code* and treat the *implementation as data*. With this method, it suffices to know *what* the code does without knowing *how* it does it. In the previous setting, when the implementation changes, even very slightly, each user must again inspect the source code to verify that what has changed does not change the functionality (or requirements) in an unwanted way. Since we treat the code as data, we can upgrade the application running inside the TEE without changing its initial state and hash, as long as the implementation still adheres to the specification.

By keeping the properties fixed, the validation of the properties does not have to be repeated. Furthermore, our framework allows code to be verified without making the implementation available. We also think that we are the first to run a verifier in a TEE.

With TRUVALT, users do not need their own TEE/verifier infrastructure, as our process ensures that only successfully verified code is used.

Our contributions in this paper are the following:

1) The verification is anchored with properties that express the requirements.
2) By using Dafny, the verification is automated and attested by means of the TEE's attestation functionality.
3) By treating code as data, we allow for implementation upgrades.

Existing works [33], [21] do not prove the correctness of the system, and the users cannot be sure what computations are carried out. Others [45], [18] use formal methods to reason about certain aspects of the application *before* it is deployed in the enclave and cannot easily be upgraded without users being involved.

This paper builds on previous work [9], where we presented a method for securely computing arbitrary aggregations on confidential data in a multi-user setting. In this work, we improve on this by allowing for upgrades to the code without requiring every user to manually inspect the new version.

The rest of the paper is organized as follows: in Section II we introduce some background concepts, and in Section III we present the overall architecture. In section IV we go through how the system is used as well as how the framework is implemented, and in Section V we analyze the security model. Section VI presents our experiments while Section VII covers the details of our different use cases. In Sections VIII and X we discuss related and future work, and Section IX concludes.

II. Background

A. Trusted Execution Environments

A trusted execution environment or *enclave* is a tamper-resistant hardware-assisted platform that protects data at use from an untrusted host. Common examples of such systems are Intel SGX[1], ARM TrustZone[2] and AMD SEV[3]. An application is deployed and executed on the specified hardware, and the computations are isolated from the host computer. The data is protected from the environment and application on the host, as all data that passes between the host and the enclave is encrypted and can only be decrypted by the enclave.

In this paper, we use Intel SGX on an Azure cloud environment, but our solution is not limited to a specific type of TEE.

1) Remote attestation: It ensures that the expected application is running on the expected hardware. This process provides a proof that contains the hash of the initial state of the currently loaded application on the TEE.

The attestation is done in two steps: first the enclave registers itself with a trusted provider in the cloud and proves its identity by a cryptographic proof using the hardware-encoded key. In the case of SGX, this is done towards Intel's attestation service. Second, a client requests the proof of what application is currently running in the enclave. The TEE generates a signed message containing the hash of the application's initial state. The client can verify the signature using Intel's attestation servers and compare the hash with the expected application. This assumes that the client that performs the attestation has access to either the binary or the source code. In the latter case, the source code must be written and compiled in a reproducible way.

2) Seals: They provide the possibility for an application that runs on a secure enclave to securely store runtime states. This data is stored on the untrusted host, but is encrypted with a key derived from the enclave. There are, in general, two ways to derive this key: either by only using the enclave's own hardware key or by combining this key with the initial state of the enclave. In the first case, any application running on the same TEE has access to the sealed data, while in the second case, only the specific application that seals the data can read it. Further, we assume that the key is derived in such a way that only the exact same application that sealed the data has access to it. Note that any seal created by an application does not modify its initial state, and hence is independent from the attestation process.

3) Gramine: Gramine is a library OS with the purpose of porting and running *unmodified code* inside a TEE.

Trusted execution environments use their own separate API for communication and execution. On top of this, some functionality is not allowed inside the enclave for security reasons. For example, an application running in an enclave should not be allowed to write plain data to the untrusted host environment or to print information to the standard output, since this can leak confidential information to the host. In general, this means that an application that is to be executed inside a trusted enclave must be developed specifically for this environment.

[1]https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html
[2]https://www.arm.com/technologies/trustzone-for-cortex-a
[3]https://www.amd.com/en/developer/sev.html

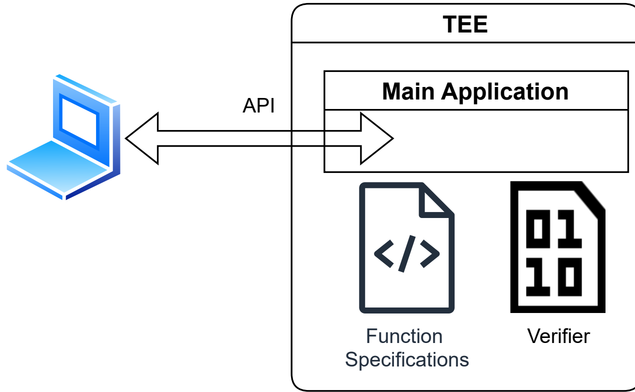


Fig. 1: The static initial state of TRUVALT. Main handles the communication to the outside, as well as connecting the other functionalities. Both the specifications and the verifier are included in the build and can be verified during attestation.

With Gramine, one can write an application as usual and then run it on an enclave using Gramine as a wrapper for the specific API functions. For example, when running a plain Python program that writes some data to disk, Gramine will automatically seal this data so that only the TEE can access it. Yet, no modifications have to be made to the Python code. This makes it possible to secure already written applications by running them in a TEE when the host is untrusted.

B. Dafny

Dafny is a software for automatically proving functional properties of code. It is written in its own “verification-aware programming language”.[4]

The developer defines specifications for functions, and Dafny verifies whether these hold for the implementation. For this, Dafny uses the Z3 theorem prover.[5] In addition to verifying conformance to the specification, Dafny can transpile and compile to code in several target languages, which will also adhere to the same specification. At the time of writing, Dafny provides support for C#, Java, JavaScript, Go, and Python.

III. Architecture

A. TRUVALT framework

Figure 1 shows three different parts of the framework; *Main Application*, *Function Specifications*, and *Verifier*. All these parts belong to the initial state of the TEE and hence are part of the attestation process. Hence, if any of these parts is changed, this will be noticed when attesting the TEE. For different applications, parts of the framework are modified. For example, the use of different verifiers might need the main application to be changed, the same for the function specifications. For a specific application, the function specifications are tailored for that purpose and can not be modified later without changing the hash of the framework. For the remainder of this paper, we will assume that the framework has been tailored for a specific application, and neither the Main, Verifier, or Function specifications will be changed. We also assume that all these parts have been developed in a reproducible way and can be inspected and compiled by any user (or trusted expert) who aspires to use the system.

1) Main: The glue that holds the different parts of the framework together. It exposes a communication API used to trigger different functionalities, such as attestation and deployment of upgradable code, as well as the execution of functions. For example, it could include one endpoint for attestation and secret sharing, one endpoint to deploy a new implementation corresponding to some present specification (which also would trigger the verification by running Dafny on the code), and one endpoint to trigger the computation. The key aspect of Main is that it enforces the verification of deployed functions.

2) Verifier: Without loss of generality, we choose Dafny as our automated verifier. Dafny is open source, does not need any interaction to verify the given code, and can transpile to a number of languages that can then be evaluated inside the TEE. For more complex projects, more advanced verifiers may be needed, but the general principles of this paper should still be valid.

When deploying the TEE, the Dafny binary is included in the build process, hence the exact version used has to stay the same. This makes it possible during attestation to verify which version of Dafny is used and hence check if the verifier has any known vulnerabilities or limitations.

3) Function Specifications: The specifications are written in Dafny syntax and can consist of anything that is supported by the Dafny language. This includes `ensure` statements (post-conditions) and helper functions that can be used to verify certain properties. The specifications are part of the initial state of the framework and cannot be changed without changing the hash during the attestation. These specifications manage what the implementation is supposed to do, hence they need to be inspected by the users to make sure that the expected functionality is correctly specified.

B. Parties

1) Developer: This party is responsible for setting up the environment, deploying the binary on the TEE, as well as making the function specifications available for inspection by any potential user. This party later on deploys the implementations for the upgradable functions, as well as potential upgrades during the application’s lifetime.

2) User: This party verifies the source code of the framework. That is, the user makes sure that the verification is enforced on the unknown implementations and that the function specifications are correct. Since the framework can be compiled in a reproducible way, the user can generate the hash of the framework and later compare this with the hash from the attestation procedure.

[4] https://dafny.org/
[5] https://github.com/Z3Prover/z3

C. *Upgradable functions*

Each function consists of two parts: the *specification* and the *implementation*. The specification defines the API of the functions. That is, the input arguments that the function requires, as well as the returned type. In addition, each specification specifies the pre- and post-conditions and potential helper functions. The implementation is the actual code that defines *how* the pre- and post-conditions are fulfilled.

The specification is part of the static aspect of the TEE application, while the implementation is loaded during runtime. The complete function is then created by merging the two parts to form a complete Dafny code file.

Dafny then verifies that the conditions hold and, only if they do, transpiles to code that can be compiled and executed in the TEE. Transpilation is the Dafny process that converts the Dafny code files into equivalent files in another language, in our case, Python. If other target languages are used, such as Java, the transpiled code is also compiled to an executable.

Listing 1 shows an example specification of the Fibonacci function[6]. The specification defines a helper function `fib` that is used by the post-condition and specifies what kind of arguments this function takes. Listing 2 shows the actual implementation. The specification is merged with the implementation during runtime; the latter can change without notifying the user. However, only implementations that conform to the specification are accepted.

Listing 1: Fibonacci specification

```
1  function fib(n: nat): nat {
2    if n == 0 then 0
3    else if n == 1 then 1
4    else fib(n - 1) + fib(n - 2) }
5  method ComputeFib(n: nat) returns (b: nat)
6    ensures b == fib(n)
7  { // Implementation body }
```

Listing 2: Fibonacci implementation

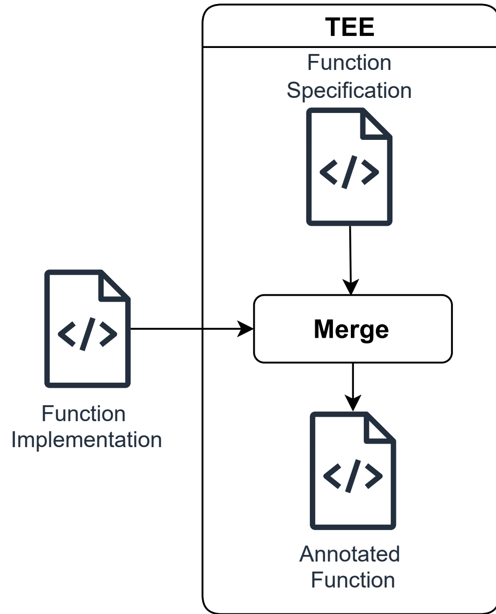
```
1    var i := 1;
2    var a := 0;
3    var b := 1;
4    while i < n
5      invariant 0 < i <= n
6      invariant a == fib(i - 1)
7      invariant b == fib(i)
8    {
9      a, b := b, a + b;
10     i := i + 1;
11   }
```

IV. Implementation and Usage

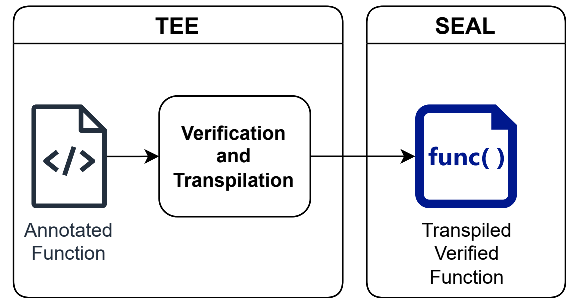
Our tool first merges the specification with the implementation and passes the result to the verifier. If the verification is successful, the code is then transpiled for later use.

Figure 2 visualizes the general workflow of TRUVALT. It is divided into two main parts: a) The deployment of the upgradable verified code (Figure 2a) to be executed on the users' data, and b) the verification and code generation of the deployed code (Figure 2b). When any user triggers a verified function, the function is loaded from the seal and evaluated on the input data.

[6]Example taken from the Dafny tutorial: https://dafny.org/dafny/OnlineTutorial/guide.html



(a) The function implementation is provided to the app and is merged with the corresponding pre-specified function specification.



(b) The annotated (merged) function is verified by Dafny. If the verification passes, Dafny transpiles the code and stores it encrypted on disk as a sealed file. If verification fails, the implementation is rejected.

Fig. 2: TRUVALT: Deployment, verification, code generation.

A. *Deployment and initial verification*

The Developer deploys the framework in a TEE-equipped cloud environment. This includes the Main part, as well as the Verifier and the Function specifications. We here assume that these have been inspected by potential users and are trusted. The attestation property of the TEE further ensures that each user can be sure that it is actually TRUVALT that is running.

Note that in this stage, the seal is empty and the aggregation functions are just specifications without any means of being executed.

B. Deployment of upgradable functions

When the initial deployment is done, the Developer sends the implementations of the upgradable functions to the enclave. In a production environment, the received implementation should be properly sanitized before the Dafny files are generated. This process should take care of any illegal characters and statements that a malicious developer might include. This is considered orthogonal to our work, and in our implementation, the input is assumed to have been sanitized.

The implementation is merged with the previously deployed pre- and post-conditions and combined into a complete Dafny code file. The enclave then executes Dafny on the deployed code file for verification of the post-conditions. If the verification succeeds, Dafny then transpiles these files to code that is stored as seals in the enclave. These seals are encrypted by the enclave and can only be decrypted by the enclave itself. Hence, the data can not be accessed by the host, and any modification would corrupt the files. In the case where verification fails, no transpilation occurs, and the functions cannot be executed. At the same time, they are persistent, so if the TEE is shut down for some reason, it can still use the previously generated implementations at the next startup without running the verifier again.

C. Upgrading functions

The upgrade process is identical to the initial deployment process in practice. The developer sends a new implementation that will form a Dafny code file and be verified by Dafny. If the verification succeeds, Dafny will generate new code for this function, and the new implementation will replace the previous version. If the verification fails, no new code will be generated, and the previous implementation will continue to be active.

D. Execution of upgradable functions

When a user sends data to the enclave to use the functions, the enclave loads the code generated by Dafny from the seal. Only if Dafny has verified the code as correct in terms of predefined pre- and post-conditions will this code exist. Since it is encrypted, only the enclave itself has the ability to read the data, and no other party would be able to modify the code without breaking it. This means that the code that is evaluated on the users' data will be correct in terms of the pre- and post-conditions that were initially deployed with the enclave.

E. Proof-of-concept implementation

A proof of concept of the proposed system was implemented using a virtual machine in Microsoft Azure's confidential computing environment. The specifications of the VM can be seen in Table I.

The TEE application is developed in Python and deployed as a TEE application by using Gramine, which makes it fairly easy to run unmodified code inside a TEE running on Intel SGX. The source code is available as an open-source project.[7]

[7]https://github.com/marbirg/trust_and_verify_upgradable_trusted_functions

TABLE I: System properties for SGX-enabled VM running on Microsoft Azure Cloud.

Property	**Value**
OS/Kernel	Linux SGX 5.4.0-1104-azure #110~18.04.1-Ubuntu SMP x86_64 GNU/Linux
CPU	Intel(R) Xeon(R) E-2288G CPU @ 3.70GHz
Size	Standard DC2ds v3
vCPUs	2
RAM	16 GiB
Python version	3.10.12
Dafny version	4.6.0.0

F. Dafny pre- and post-conditions

Dafny provides the `requires` statement that can be used by the prover for assumptions, e.g., an input array is not `null` and has a length greater than 0. Unfortunately, these prerequisites are not enforced in the exported code, hence a requirement such as `requires input_array.Length >= 200` would not be respected. Such enforcements must be written in terms of `ensures` statements.

In Listing 3, we show such an example in a `count` function. The function is supposed to count the number of occurrences in an input array `a` and return it as a multiset `c`. In addition to ensuring that the input is of a specific size, it also requires that no count can be less than a predefined number, in this case, 10. Note that these conditions might not be enough for confidentiality, depending on the type of data and application, but more such conditions can easily be added for other properties, such as the number of unique input values or non-zero values.

Hence, in the examples that we provide, the only conditions that will be enforced and verified by Dafny are the `ensures` statements.

Listing 3: Ensuring at least 200 number of participants and at least 10 entries per count. Otherwise an empty set will be returned.

```
1 method EnsuresArrayLength(a: array<int>) returns
      (c:multiset<int>)
2    requires a != null && a.Length>0
3    ensures (a.Length<200 && |c|==0) || a.Length >=
         200
4    ensures forall i :: 0 <= i < a.Length ==>
         c[a[i]] >= 10 || |c|==0
```

V. Security analysis

A. Attacker model

We assume that the service provider of the host environment is not untrusted in its core, i.e. this entity is not assumed to manipulate the firmware of the trusted enclave, or use hardware-based side channels to extract decryption keys, for example. What we do assume is that the host environment can be compromised by a malicious actor. Since this actor does not have physical access to the hosting machine, side-channel

TABLE II: Result from verification of Dafny files from DafnyBench repository. Time is given in seconds.

Files Analyzed:	771
Successfully verified:	747 (96.9%)
Mean time in enclave:	168.43
Mean time outside enclave:	2.26
Overhead:	ca 75×
Geometric mean in enclave:	174.14
Geometric mean outside enclave:	1.92
Overhead:	ca 90×

attacks or firmware manipulation are out of scope. The system does not protect against rollback attacks on the seals, but such attacks cannot affect the core functionality of the system.

The Developer is assumed to be untrusted and might try to create implementations that change the functionality of the system.

B. Rollback attacks

All data that is stored persistently on the system is stored encrypted in so-called seals. These encrypted files can be accessed by the host system, but not read due to encryption. The encryption key is derived from the TEE and the initial application state. Hence, the seal is not readable from any other TEE or any other application running on the same TEE. However, it is possible for the host system to store a copy of the seal, and at a later point in time, replace a newer version of the code with an older one.

Protection against rollback attacks is not possible with a single system, but there are works [24] that show that by combining several trusted platforms, it is possible to detect such attacks as long as all platforms are not acting maliciously. Such mechanisms could easily be added to our system and are considered orthogonal.

In our system, a rollback attack would still have a limited effect on the confidentiality of sensitive data. Such an attack would only affect the performance of the system, assuming that the newer version was optimized in some way, but it would not affect the general functionality of the system. This is the case since every version of the code that is stored as a seal has been verified by Dafny first. Hence, the old rolled-back code would still be verified according to the specification and fulfill the same properties.

VI. Controlled Experiments

We verify the correctness and performance of our framework with controlled experiments. These experiments consist of ensuring that Dafny can be reliably used as a verification tool running inside a secure enclave, as well as comparing the performance overhead of running it directly on the host system versus in the enclave.

A. Dafny verification

To confirm that Dafny running inside the enclave handles a variety of different code implementations, we use the dataset[8] from DafnyBench [23]. We have used the files that they refer to as *ground_truth*, which consists of a total of 782 Dafny source files. Of these, some cannot be verified with our version of Dafny. The remaining 771 files are each sent to be verified in the enclave. Dafny running in the enclave successfully verifies 747 files. We send each file as a JSON object to the main application using an HTTP POST request. The verifier returns a response that indicates success or failure. The time between sending the data and retrieving the result is recorded for each Dafny file. Since all network communication occurs on the same host, we believe the additional time for sending and receiving the data is negligible, and the recorded time is assumed to be the verification time. It should be noted that in this setting, Dafny only verifies that the files are correct according to the specified pre- and post-conditions. No transpilation or compilation is done.

We compare the time for verification with running the same service outside the enclave. The result is shown in Table II. As one can see, it takes ca. $90\times$ longer time to do the verification inside the enclave. This is due to the large amount of memory needed for the verification, which indicates that the enclave must cache data at the host. This operation is time-consuming, since cached data moving in and out of the enclave has to be decrypted and encrypted, respectively. In a practical setting, the verification is only done during deployment and upgrades, which should not happen very frequently; hence, we argue that this overhead is acceptable.

B. Micro benchmark

From the files taken from DafnyBench, we select a few examples to compare the runtime overhead of transpiled code. The selected samples consist of computing a sum of an array of integers, a basic find function, binary search, merge sort, and bubble sort. The files are first verified by Dafny to be correct and then transpiled to Python code. The Python code is then evaluated by triggering a specific endpoint in the web server. In the request, we submit the data that is supposed to be used as an input argument to the generated Python code. This experiment is carried out by running the service both inside the enclave and outside of it.

We evaluate each function 1000 times and record the computation time. The result can be seen in Table III. We have a geometric mean overhead of only $50\,\%$, which we believe is acceptable in regard to the increased security and confidentiality aspects given by the secure enclave.

For most functions, an array of random integer values has been generated and supplied to the functions. In the cases of `Find` and `BinarySearch`, an integer that exists in the array has also been supplied. For `BinarySearch`, the supplied integer array is sorted.

[8]https://github.com/sun-wendy/DafnyBench?tab=readme-ov-file

TABLE III: Compared evaluation time [ms] for code inside and outside of the enclave, as well as the geometric mean

		Enclave	Non-enclave	Overhead
SumArray	Mean	3.52	2.27	1.55
	Median	3.33	2.16	1.54
Find	Mean	3.04	1.99	1.53
	Median	2.88	1.92	1.50
BinarySearch	Mean	2.82	1.83	1.54
	Median	2.62	1.72	1.52
MergeSort	Mean	50.84	33.41	1.52
	Median	49.94	32.70	1.53
Bubblesort	Mean	833.92	565.10	1.48
	Median	833.24	564.69	1.48
	Geometric mean:	16.34	10.82	1.51

Listing 4: Ensuring correctness of sorting algorithm

```
1 predicate Sorted(q: seq<int>) {
2    forall i,j :: 0 <= i <= j < |q| ==> q[i] <= q[j]
       }
3 method Sort(a: array?<int>) returns (b: array<int>)
4  requires a!=null
5  decreases a.Length
6  ensures Sorted(b[..])
7  ensures multiset(b[..])==multiset(a[..])
8  ensures b.Length == a.Length
```

C. Upgrading functions

To verify that the functionality of Dafny is as expected, even when running in the TEE, we test this by sending two versions of known working implementations for different sorting algorithms. In Listing 4, the specification for the sorting function is defined. To verify that this specification works for different sorting implementations, the system was initially deployed with an implementation of the Bubble Sort Algorithm. Upon deployment, Dafny verified that the implementation was correct according to the specification and generated the equivalent Python code. The generated code was triggered by sending an array of randomly generated integers, and the resulting array was verified to be correctly sorted. After this stage, the client deployed another sorting implementation, now using the Merge Sort algorithm. Dafny correctly verified that this implementation was also correct according to the same specification, and the generated Python code was again tested in the same manner.

D. Rejecting an implementation

To verify that Dafny indeed rejects implementations that do not adhere to the specification, even when running in the TEE, we modified a piece of code to be slightly incorrect with respect to the specification. In Listing 5, the specification for the Voting function is specified. The original implementation of this function can be seen in Listing 6.

To create a incorrect version of the implementation, we change line 11 from `if b[y]>a.Length/2` to `if b[y]>=a.Length/2` (notice the change of > to >=). This is a very small change and might come from a simple development error, but now the specification does not hold. We saw in our experiments that Dafny correctly rejects the new implementation and does not generate any new Python code. Hence, if the function were to be triggered again, the old (and correct) code would be executed.

Listing 5: Ensuring correctnes of majority voting

```
1 method Count_votes(a:array<int>) returns
      (count:int, value:int)
2  ensures count==-1 || count>a.Length/2;
3  ensures count==-1 || count==multiset(a[..])[value];
```

Listing 6: Implementation of Voting function

```
1  {
2   var b :=multiset(a[..]);
3     var keys := ( set keys | keys in a[..] );
4     count:=-1;
5     value:=-1;
6     var c := keys;
7     while ( c != {} )
8           decreases c;
9           invariant count==-1 || count>a.Length/2
10          invariant count==-1 ||
                count==multiset(a[..])[value]
11        {
12           var y :| y in c;
13           if b[y]>a.Length/2 {
14                 count:=b[y];
15                 value:=y;
16              }
17        c := c - { y };
18    }
19 }
```

E. Threats to validity

Dafny can transpile to several languages other than those we have evaluated. It is possible that transpiling to another language, especially one that needs compilation, would affect the overhead when deploying and upgrading functions. In addition, it is also possible that another language runtime overhead might differ for other runtime environments, such as CLR or Java.

The experiments have been performed in a controlled environment on a few different Dafny implementations. More complex setups that might require more memory use might affect the outcome negatively. Such issues could be solved by using an enclave with more memory, but this has not been tested. A larger enclave could have a positive effect on the overhead since less data would need to be cached on the host environment, but this has not been evaluated.

VII. Use Cases

To demonstrate the applicability and scalability of our approach, we show a couple of use cases with more complex requirements and implementations.

A. Use cases

To set the system in a realistic setting, we present several potential use cases for this setup. These use cases perform computations with data that can be considered confidential. Hence, the system can be trusted if it is possible to verify that it does not leak sensitive information beyond the result of the computation and that crucial functionality is verified to be

correct. In addition, these are examples that it is reasonable to believe will need to be upgraded during the lifetime of the application.

1) Confidential Taxi Monitoring: In this setting, different taxi companies monitor their individual taxi cars in the same city. To be able to get a view of where there is a need for sending taxi cars, and where there might be too many, each company would like to compute the total number of taxi cars in certain zones in the city, but neither taxi company wants to reveal the position of their taxi cars to any other company.

In our implementation, the count mechanism is defined by the specification in Listing 7, and mandates that enough taxis have to be part of the computation, as well as that at least a threshold of taxis must be in each position to get the value. Each taxi reports its real-time position at some time interval, and the taxi companies can then query how many taxi cars there are in certain zones of the city. If too few cars are in a specific zone, the zone is reported to have 0 cars.

Listing 7: Ensuring number of taxis is at least 100 entries per count

```
1 method CountTaxis(i:int, a: array<int>) returns
       (c:int)
2      requires a != null
3      requires a.Length>0
4      ensures c>100 || c==0
```

2) Majority Voting: For safety-critical operations, such as healthcare and aviation, several redundant sensors are used for each critical property to detect sensor failures. In avionics, the requirement for safety sets very high standards for the possibility of several sensors failing simultaneously. A voting system is used to determine the actual value based on the measurement of each sensor. It is crucial that this system is correct, and by leveraging the power of formal verification, one can be certain that the implementation fulfills the requirements. Using a TEE in this setting reduces the risk of tampering with the software, and one can verify that the conditions for the software will hold, even if the implementation changes.

In our implementation, the voting system allows sensors to asynchronously report data using a secure channel to the TEE. After a certain time, this data is sent to a function that is specified by a function specification that ensures how the output will be constructed. In our example, the input consists of an array containing the value reported from each sensor. The voting function will report the winning value as long as more than half of the sensors report the same value; otherwise, -1 will be reported, indicating an error. The specification for this implementation can be seen in Listing 5.

3) Smart Health Aggregation: In this setting, we assume that users store potentially confidential data encrypted in the cloud for availability and backup. The users still want to compute statistics on the population to be able to compare themselves to the rest of the users.

For example, users can upload the time that they run $5\,\mathrm{km}$ and then request the median value for all users who have done the same. Only users who have provided their decryption key to the enclave can be included in the computation, so each user must explicitly agree on this requirement. For the aggregation, a sorting function has been developed in Dafny, and the main application uses this verified code generated by Dafny to sort an array containing all included users' data. The median is then computed by the main application using the sorted array.

The specification of the sorting function can be seen in Listing 4. There are many different sorting algorithms available, and they differ depending on how one expects the values in the array to be distributed, or if one wants to optimize for speed or memory. At the same time, it can be fairly hard to inspect a sorting algorithm manually and verify its correctness. By contrast, it is arguably simpler to inspect the post-conditions of a sorting algorithm, which can be done with little technical competence. Our system makes it easy to verify the correctness of the sorting algorithm, without even looking at the implementation, while at the same time, the implementation can easily be changed if the developer decides that another implementation is more suitable.

B. Ethereum Virtual Machine in Dafny

The GitHub project `Dafny-EVM`[9] aims to create a formal specification of the Ethereum Virtual Machine[10]. The project is actively maintained, is ranked #2 on GitHub based on the number of stars on Dafny projects, and consists of 6900 lines of Dafny source code.

Ethereum is one of the most popular platforms to run smart contracts and one of the most popular cryptocurrencies. Hence, it is of great importance that the virtual machines that act as nodes in the network are correct.

We successfully used Dafny to verify the source code of *Dafny-EVM* inside the enclave. In addition, we transpile the code to Java and compile and bundle the project into a jar-file, also inside the enclave.

By moving Dafny-EVM inside the TEE, without any modification of the source code, we can provide secure attestation of the verification, transpilation, and build process. We can hence transfer the integrity guarantees given by the TEE, to any user that wants a formally verified version of the EVM, with no need to verify and build the project themselves. The final binary would be retrieved directly from the TEE and enable a sort of attested *formal verification and build as a service*.

Smart contracts can be a way to perform a trusted computation in the sense that the implementation can compute over many users' data, but cannot be changed once deployed. Much work has gone into verifying the semantics of smart contracts [3], [19], but less attention has been paid to the correctness of the EVM, which executes the smart contracts themselves.

Dafny-EVM consists of source code written in Dafny. After a successful verification, these files are transpiled to Java code, which is then compiled and packed as a jar file. The final binary is subsequently formally proven to adhere to

[9]https://github.com/Consensys/evm-dafny/tree/master
[10]https://ethereum.org/en/developers/docs/evm/

the contracts' specification in the Dafny code. The resulting bytecode can then be compared to other EVM implementations on byte level, and hence be reasoned about formally. If the bytecode is identical, they should adhere to the same proof.

1) Challenges building in the TEE: Gramine aspires to run unmodified applications inside a trusted enclave, but some aspects are more complex to get to work than others, and some aspects are not implemented by Gramine[11] for different reasons. Because of this, the build instructions for this project could not be executed without modification.

a) Gradle: The project uses Gradle as a build system, which is not made to run in an isolated system. Some issues in getting Gradle to run inside the enclave were due to limited (internal) network communication. Relevant *ioctl* calls must be explicitly allowed in the Gramine manifest file. In addition, the Gradle process used all of the available memory until it crashed the enclave, something that we did not observe when running it on the host system. Our best guess is that Gradle spawns separate processes, which are allocated much more memory than necessary because of how Gramine allocates memory.

The functions of the Gradle configuration flags are not always obvious. For example, Gradle accepts an `--offline` flag, which could be assumed to mean that it will not try to pull dependencies from online sources. This is though only a best effort promise. If it is missing some dependency, it will still try to fetch this from online sources without asking. We observed a similar behavior with the `--no-daemon` flag. Despite the name, this does not mean that it is running without daemons, it just means that the daemons will be stopped when the build is completed.

b) Debugging TEE applications in Gramine: Debugging inside the enclave is problematic since the applications, by their nature, are isolated from the host. The best strategy is to enable more fine-grained logging in Gramine.

To determine the root cause of the error when running Gradle inside the enclave, we increased the log level to `trace`. This level prints a very large amount of information, and can be difficult to analyze in the prompt.

To reduce the noise in the logs, we instead determined the commands that Gradle ran by increasing its log level. By manual inspection of these logs, we extracted the commands that Gradle executed during the build. We then ran each command individually in the enclave without Gradle. This made it possible to pinpoint what fails and when. The verification process was completed successfully, and so was both the transpilation and the compilation process. The build failed when the `jar` command was used to bundle the compiled class files, since `jar` uses the system call `copy_file_range`, which is not implemented in Gramine. This is only possible to notice if using log level `all` or `trace`.

c) Building a jar in Gramine: The jar binary is used to package compiled Java class files into a bundle that can be run independently (on a system equipped with the Java Virtual Machine). The binary takes as argument the path where the class files are located, and a (non-mandatory) path where to store the jar bundle. It then creates the file in a temporary location, after which it moves the bundle to the specified path. To optimize this, the system call `copy_file_range` is used. This command moves the file in the file system without moving any data on disk. Unfortunately, this system command is not implemented in Gramine. However, if the output path argument to the jar command is omitted, the binary instead writes the output to stdout. In this case, it is possible to pipe this data to the desired file location and hence build a jar bundle inside the enclave.

This workaround is unfortunately not possible to use when using Dafny to build the jar file, since Dafny mandates the use of an output file when calling the `jar` command. Because of this, we were not able to execute the supplied tests in the repository. To do this, a change needs to be made in the Dafny API, to allow the user to supply a flag or similar to make the jar binary pipe the output to a file rather than using the non-supported system call. We believe that this should be a minor change, but it is out of scope for this paper.

2) Our contribution: By using an existing project on GitHub that is highly ranked compared to other Dafny projects, we have shown that our framework is feasible for real applications. We were able to both verify and build the project inside the secure enclave using Dafny and Java. We noted limitations in Gramine that prevent us from using parts of Dafny as one would on the host system, but presented a workaround that still allows us to build the jar package inside the enclave. We have shown how to debug the application inside the enclave, which can be beneficial for future work.

VIII. Related work

To the best of our knowledge, our work is the first to use unmodified off-the-shelf verification tools inside the enclave to enforce properties of both correctness and security. In this section, we will discuss related works in the areas of *Proof carrying code*, *Smart Contracts*, *TEE-backed analysis*, and *Software verification*.

A. Proof carrying code

Verification of code before execution is not new. The idea of *Proof carrying code* [31], [32] introduces the idea of including a proof with the code that can be checked by the user before execution. This makes it possible to verify certain properties of the program. The proof can verify security-related properties, but not necessarily show correctness.

The challenge at the time was that the creation of the proof was not automated and hence very time consuming [31]. We believe that advances in automated provers now make it possible to state relevant properties without the burden of a full proof, although writing a strong specification remains a challenge.

[11] https://gramine.readthedocs.io/en/stable/devel/features.html?highlight=system%20calls

B. Smart contracts

The issue of not being able to change trusted code has mainly arisen in the context of smart contracts, where the proxy pattern has been proposed as a workaround [11]. In this setting, the proxy contract is static, but uses other contracts for computation, and these contracts can be changed. Obviously, if allowing the contracts to be upgraded arbitrarily, the security and confidentiality model would fall. Antonino et al. [5], [4] propose a system with a trusted deployer that verifies that certain properties of the contract hold after upgrade. By using pre- and post-conditions that are verified, if a contract violates these properties, transactions will fail and revert. Smart contracts still have the limitation of large computational overhead when aggregating on large amounts of data, and are generally not the best choice for use with confidential data, because all transactions are stored publicly on the blockchain.

C. TEE-backed analytics

TEEs have been adopted in many different settings to enhance security and transparency for the users. This paper builds on previous work by Birgersson et al. [8], [9], which aims at computing arbitrary aggregations on confidential data in a multi-user setting. Islam et al. [15] present a secure and confidential solution for a Trigger-Action platform. The rules are encrypted, as well as data from triggers, and can only be evaluated inside a trusted enclave. In the work of Ayoade et al. [6], blockchain technology is used as a tamper-proof system for managing access control in an IoT environment, while a TEE is used for secure storage. Walnut [37] takes the security of a Trigger-Action platform even further, and combines homomorphic encryption with multiple TEEs to enhance security. Gremaud et al. [14] and Moazen et al. [27] present architectures for deploying Trigger-Action platforms on untrusted hosts by using TEEs. Neither of these works focuses on a multi-user setting, and neither addresses the issue of upgrading already deployed code.

Other works [26], [2], [10], [42] propose sandboxing techniques to isolate and mediate communication between applications and with the host environment. The focus on TRUVALT is instead to give guarantees to data owners that a specific computation will be carried out, without the need to trust the host environment.

TEEs have also been used to protect the device itself [35], [17]. For a more comprehensive list of methods to use trusted execution environments to protect devices in a cloud/fog environment, we refer to the literature review by Valadares et al. [40].

Machine learning applications are another setting where TEEs are used. Both FLATEE [28] and Chen et al. [12] use TEEs in federated learning schemes to replace homomorphic encryption while preserving integrity and protecting against data poisoning attacks.

In these settings, the users are not aware of the details of the application and are not responsible for attestation; hence it is not an issue to upgrade the code as it is when the same instance should be transparent for all users. Our work differs in this regard as we transfer the integrity guarantees given by the TEE to the user. Because we separate the properties of the code from the implementation, we are still able to perform upgrades as long as the properties are fulfilled. At the same time, the users do not need any extra capabilities or do any extra work, since the verifier of these properties is run inside the enclave. Users hence need to inspect less code; only properties and not the implementation.

D. Software verification

There are still many challenges in the area of software verification. The authors of [38], [29] reason about both computational contracts as well as effectful computational contracts. Fördős [13] proposes a method for identifying 'trust zones' in Erlang source code, as well as detecting potential vulnerabilities by static analysis. These works are considered orthogonal to ours.

The use of formal verification tools in the context of TEEs is itself nothing new. Kumar Jangid et al. [18] use Tamarin to reason about state continuity for enclave applications, something that is orthogonal to our work. Zhang at al. [45] present a framework for detecting data leakage in ML applications. The framework is used on the code before it is deployed and is hence a tool for the developers rather than a guarantee for the users.

Liu at al. [21] decompile deployed functions and verifies policies similar to proof carrying code, but focus on memory corruption. Hence, previous work on verification and TEEs is about reasoning about code in the enclave, but does not run verification tools inside the enclave. More similar to our work is DuetSGX [30], [33], which is a framework to automatically verify that an application is differentially private, but it does not verify correctness. This is, hence, rather an interesting functionality to add to our work for increased privacy.

IX. Conclusion

We have presented TRUVALT, a framework for automatically and formally verifying code running inside a TEE. The framework consists of formal specifications of functions, a formal verification tool, and a Main part that provides a communication API and binds all parts together. The TEE is leveraged for integrity and transparency, while a formal verification tool, Dafny, is automatically enforcing function specifications without the need to reveal the implementation. We have successfully used the tool to verify 747 Dafny files found online. We have evaluated the tool by running it both inside and outside a secure enclave. The verification time in the enclave shows an overhead of $90\times$, but since code is only verified during initial deployment and upgrades, something that should happen fairly infrequently, it is acceptable. In addition, we have evaluated the framework on a micro-benchmark, which shows an overhead of only about 50%, which is considered acceptable because of the increased security and correctness guarantees that come with the framework. Finally, we verified that the framework is suitable for a real-world application by running the verification procedure as well as

the transpilation and compilation of a popular GitHub project, Dafny-EVM.

TRUVALT makes it possible to seamlessly upgrade code running in a TEE, without changing the attestation hash, while still providing guarantees that the initial contract with the users is still intact. Our work fulfills a strong need for verification combined with upgradability. We expect recent improvements in automated proof technology to give rise to stronger specifications and thus trustworthy automated upgrades.

X. Future work

The current system does not protect against rollback attacks. This is in general, no issue since all previous versions of the functions have been verified, but for integrity purposes of the application, such protection would still be valuable. Dafny is a good verifier in terms of proving correctness for an application, but is fairly limited. It has, for example, no capability for randomness. To add focus on privacy on top of correctness when upgrading code, something similar to DuetSGX [30], [33] would be an interesting continuation. Extensions of Dafny to provide information flow analysis have been suggested [34], but unfortunately, that implementation depended on an old version of C#, but such additions to Dafny would make our system even more versatile.

Dafny is just one verification technique, and others would be interesting to incorporate and evaluate in a similar setting. Notable examples would be Liquid Haskell [41], [22], VeriFast [16] and Coq/Spoq [20].

Acknowledgements

This work was partially supported by the Centre for Cyber Defence and Information Security (CDIS), Digital Futures, the Swedish Research Council (VR), and Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Comput. Surv.*, 51(4), July 2018.

[2] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven trigger-action platforms. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2899–2916. USENIX Association, August 2021.

[3] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:101227, 2020.

[4] P. Antonino, J. Ferreira, A. Sampaio, et al. A refinement-based approach to safe smart contract deployment and evolution. In *Softw Syst Model 23*, pages 657–693, August 2024.

[5] Pedro Antonino, Juliandson Ferreira, Augusto Sampaio, and A. W. Roscoe. Specification is law: Safe creation and upgrade of Ethereum smart contracts. In Bernd-Holger Schlingloff and Ming Chai, editors, *Software Engineering and Formal Methods*, pages 227–243, Cham, 2022. Springer International Publishing.

[6] Gbadebo Ayoade, Vishal Karande, Latifur Khan, and Kevin Hamlen. Decentralized IoT data management using blockchain and trusted execution environment. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 15–22, 2018.

[7] Musard Balliu, Iulia Bastys, and Andrei Sabelfeld. Securing IoT apps. *IEEE Security & Privacy Magazine*, 2019.

[8] Marcus Birgersson, Cyrille Artho, and Musard Balliu. Security-aware multi-user architecture for IoT. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 102–113, 2021.

[9] Marcus Birgersson, Cyrille Artho, and Musard Balliu. Sharing without showing: Secure cloud analytics with trusted execution environments. In *2024 IEEE Secure Development Conference (SecDev)*, pages 105–116, 2024.

[10] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *2010 5th International Conference on Malicious and Unwanted Software*, pages 55–62, 2010.

[11] Van Cuong Bui, Sheng Wen, Jiangshan Yu, Xin Xia, Mohammad Sayad Haghighi, and Yang Xiang. Evaluating upgradable smart contract. In *2021 IEEE International Conference on Blockchain (Blockchain)*, pages 252–256, 2021.

[12] Yu Chen, Fang Luo, Tong Li, Tao Xiang, Zheli Liu, and Jin Li. A training-integrity privacy-preserving federated learning scheme with trusted execution environment. *Information Sciences*, 522:69–79, 2020.

[13] Viktória Fördős. Secure design and verification of erlang systems. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang*, Erlang 2020, page 31–40, New York, NY, USA, August 2020. Association for Computing Machinery.

[14] Pascal Gremaud, Arnaud Durand, and Jacques Pasquier. Privacy-Preserving IoT cloud data processing using SGX. In *Proceedings of the 9th International Conference on the Internet of Things*, IoT 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[15] Md Shihabul Islam, Mustafa Safa Özdayi, Latifur Khan, and Murat Kantarcioglu. Secure IoT data analytics in cloud via Intel SGX. In *13th IEEE International Conference on Cloud Computing, CLOUD 2020, Virtual Event, 18-24 October 2020*, pages 43–52, 2020.

[16] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA Formal Methods*, page 41–55. Springer, Berlin, Heidelberg, 2011.

[17] Jinsoo Jang and Brent Byunghoon Kang. 3rdParTEE: Securing third-party IoT services using the trusted execution environment. *IEEE Internet of Things Journal*, 9(17):15814–15826, 2022.

[18] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. Towards formal verification of state continuity for enclave programs. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 573–590. USENIX Association, August 2021.

[19] Moez Krichen, Mariam Lahami, and Qasem Abu Al–Haija. Formal methods for the verification of smart contracts: A review. In *2022 15th International Conference on Security of Information and Networks (SIN)*, page 01–08, November 2022.

[20] Xupeng Li, Xuheng Li, Wei Qiang, Ronghui Gu, and Jason Nieh. Spoq: Scaling Machine-Checkable systems verification in coq. page 851–869, 2023.

[21] W. Liu, W. Wang, H. Chen, X. Wang, Y. Lu, K. Chen, X. Wang, Q. Shen, Y. Chen, and H. Tang. Privacyscope: Automatic analysis of private data leakage in tee-protected applications. In *Proceedings. International Conference on Dependable Systems and Networks*, pages 413–425, 2021.

[22] Yiyun Liu, James Parker, Patrick Redmond, Lindsey Kuper, Michael Hicks, and Niki Vazou. Verifying replicated data types with type-class refinements in liquid haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):216:1–216:30, November 2020.

[23] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification, 2024.

[24] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. page 1289, 2017.

[25] Bertrand Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.

[26] Mark S Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja safe active content in sanitized javascript.

[27] Mojtaba Moazen, Nicolae Paladi, Adnan Jamil Ahsan, and Musard Balliu. Tapshield: Securing trigger-action platforms against strong attackers. In *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025*. IEEE, 2025.

[28] Arup Mondal, Yash More, Ruthu Hulikal Rooparaghunath, and Debayan Gupta. Flatee: Federated learning across trusted execution environments. arXiv, 2021.

[29] Cameron Moy, Christos Dimoulas, and Matthias Felleisen. Effectful software contracts. *Artifact: Effectful Software Contracts*, 8(POPL):88:2639–88:2666, January 2024.

[30] Joseph P. Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, and Dawn Song. Duet: An expressive higher-order language and linear type system for statically enforcing differential privacy, 2019.

[31] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[32] George C. Necula and Robert R. Schneck. Proof-carrying code with untrusted proof rules. In Mitsuhiro Okada, Benjamin C. Pierce, Andre Scedrov, Hideyuki Tokuda, and Akinori Yonezawa, editors, *Software Security — Theories and Systems*, pages 283–298, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[33] Phillip Nguyen, Alex Silence, David Darais, and Joseph P. Near. DuetSGX: Differential privacy with secure hardware, 2020.

[34] Daniel Parkinson. Dafnyinfoflow. https://github.com/D-Parkinson1/DafnyInfoFlow, 2021.

[35] Sandro Pinto, Tiago Gomes, Jorge Pereira, Jorge Cabral, and Adriano Tavares. IIoTEED: An enhanced, trusted execution environment for industrial IoT edge devices. *IEEE Internet Computing*, 21(1):40–47, 2017.

[36] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64, 2015.

[37] Sandy Schoettler, Andrew Thompson, Rakshith Gopalakrishna, and Trinabh Gupta. Walnut: A low-trust trigger-action platform. arXiv, 2020.

[38] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Computational contracts. *Science of Computer Programming*, 98:360–375, February 2015.

[39] Nick Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18(2):28, 1996.

[40] Dalton Cezane Gomes Valadares, Newton Carlos Will, Jean Caminha, Mirko Barbosa Perkusich, Angelo Perkusich, and Kyller Costa Gorgonio. Systematic literature review on the use of trusted execution environments to protect cloud/fog-based internet of things applications. *IEEE Access*, 9:80953–80969, 2021.

[41] Niki Vazou. *Liquid Haskell: Haskell as a theorem prover*. University of California, San Diego, 2016.

[42] Zhiyuan Wan, David Lo, Xin Xia, and Liang Cai. Practical and effective sandboxing for linux containers. *Empirical Software Engineering*, 24:4034–4070, 2019.

[43] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

[44] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642. IEEE, 2023.

[45] Ruide Zhang, Ning Zhang, Assad Moini, Wenjing Lou, and Y. Thomas Hou. PrivacyScope: Automatic analysis of private data leakage in TEE-protected applications. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 34–44, 2020.