Enabling Non-Experts to Develop Distributed Trust Applications

Nicholas Ngai

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-256
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-256.html

December 1, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Enabling Non-Experts to Develop Distributed Trust Applications

by Nicholas Ngai

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

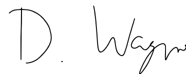
Committee:

Professor Raluca Ada Popa
Research Advisor

May 12, 2023

(Date)

* * * * * * *



Professor David Wagner
Second Reader

May 12, 2023

(Date)

Enabling Non-Experts to Develop Distributed Trust Applications

Nicholas Ngai
nicholas.ngai@berkeley.edu
University of California, Berkeley
Berkeley, California, USA

ABSTRACT

Distributed trust applications provide strong guarantees of security, in that they require a set of multiple "trust domains" to be compromised in order to compromise an entire application, rather than relying on a central point of attack, which is often the target of attacks and data breaches. Yet, distributed trust has yet to see wide-scale adoption despite its plethora of benefits.

The DoTS platform intially proposed and presented by Tan and Kaviani promises to solve many of these issues by addressing the *practical* needs of distributed trust applications as a software development kit for developers to build applications with distributed trust. In this paper, we present mid- and low-level architectural modifications to the DoTS platform in order to make the platform and API more accessible to developers, while fulfilling its original vision.

We demonstrate both that our modifications maintain the simplicity of developing new applications on top of DoTS, with our secret key recovery and (n, t)-threshold ECDSA signing applciations requiring fewer than 500 lines of integration code each, and we demonstrate that it is equally easy to port existing applications to utilize the DoTS platform, replacing over 2,600 lines of code in MP-SPDZ with simple function invocations.

We show that, by leveraging the unique conditions of such applications typically executing in distinct trust domains in geographically separated regions, applications utilizing the updated DoTS design suffer as little as 6.4–9.5% overhead compared to ad hoc, complex, application-specific network solutions.

1 INTRODUCTION

Distributed trust is a rapidly growing and evolving research space in computer security literature. At its core, the notion of distributed trust is simple: Rather than relying on a single, central point of attack, distribute trust amongst different trust domains such that a failure condition occurs if and only most or all of these trust domains are breached or become malicious.

These protocols and systems take many shapes and forms, from simple, easy-to-understand secret-sharing protocols [41] to large, complex, heterogenous systems with many moving parts like Bitcoin [34], certificate transparency [29], and IPFS [44]. Yet, with the notable exception of cryptocurrencies, many of these distributed trust systems developed in the academic space have yet to see adoption by wider communities of individuals, and many of the distributed trust applications *have* seen adoption are largely those developed in-house by large corporations, such as Apple and Google's joint venture on privacy-preserving COVID-19 exposure notifications [1]. Yet, in the aforementioned cryptocurrency space, the introduction of distributed trust ledgers like Bitcoin and its flexible cousin Ethereum [5] promising user-programmable smart contracts have brought an incredible explosion in the the localized ecosystem of distributed trust on distributed ledgers and blockchains. Ethereum, in particular, promises the ability simply write a smart contract in the straightforward form of code, and such code is immediately deployable and imbued with all the security guarantees of executing on the decentralized ledger, and the cryptocurrency community has quickly utilized this primitive to construct the affectionately named "Web 3.0."

We can infer from this history that the desire to implement and deploy distributed trust systems is present, just that the barriers to develop and deploy distributed trust render distributed trust to be not yet *accessible* to the average developer. While companies like Apple and Google have plenty of resources to develop in-house, polished, production-ready solutions to be deployed at a global scale, such resources and expertise aren't available to the average individual or even small corporations.

A prime example of this is in Signal's key recovery. While Signal offers strong guarantees for the security of locally-stored chat messages [31–33, 36], its strong levels of protection also mean that the client device, as the only trusted piece of hardware, is a single point of failure, and if the client device is damaged or destroyed, there is no way to recover the secret information stored on the device. This is a non-ideal property for the end-user of Signal, and the problem ultimately stems from the fact that Signal, as a single domain of trust, cannot be individually trusted to store the user's chat messages securely. While Signal has explored and deployed a weaker form of distributed trust in the form of secure enclaves, which create a trusted execution environment (TEE) that can only be tampered with or impersonated by the hardware manufacturer (Intel in the case of Intel SGX [22]), such solutions present scalability issues and require proprietary hardware in order to implement. Distributed trust applications, by contrast, present intuitive solutions which can be implemented with incredibly simple protocols on just about any hardware, but the ease of access for distributed trust applications is simply not yet there.

While much work on distributed trust protocols and applications has focused on the lowering the *theoretical* barriers to deploying distributed trust, such as reducing the computational cost or required network bandwidth to levels that are reasonably implementable, little work has been done to explore the *practical* barriers of developing distributed trust applications. The main efforts in this area have been Dotme [24], focusing on orthogonal issue of generating independent trust domains in the first place using on-demand cloud functions, and the DoTS platform, a proposal to build out a developer toolkit for developing distributed trust applications. We focus our efforts on the latter in order to bring performance and ease-of-use improvements to the developer and performance improvements to the end-user.

	Intel SGX	Distributed Trust
Application	SGX applications	Distributed trust applications
Developer SDK	Intel SGX SDK	**DoTS**
Security Primitive	SGX hardware instructions	Multiple trust domains

Figure 1: The role of DoTS as a bridge between the security "primitive" of trust domains and the high level logic of an application, with comparisons to Intel SGX.

Due to the human-focused and implementation-focused nature of this work, this paper will draw both from expertise on the theoretical side of distributed trust applications, identifying their needs and solutions to address those needs, and from practical experience and observations regarding how developers approach the development of other kinds of applications, such as simple web applications or Intel SGX applications. These such applications all share in common the existence of libraries that "just work" out of the box for developers to use, such as Node.js's Express library [42], Go's Gin web framework [43], the Intel SGX SDK [23], and the Open Enclave SDK [8]. With a these existing, successful frameworks in mind, the goal of this work is to flesh out and describe design details of an implementation of the DoTS platform that bring this out-of-the-box functionality to DoTS.

2 PRELIMINARIES

As the original DoTS prototype our updated design and implementation both depend on the same ideas and initial vision, we start by discussing here the core principles and modeling shared by both designs.

2.1 Guiding Principles

Our work in developing the updated DoTS platform has been guided largely by two principles:

(1) **Low barrier to entry.** For developers to use a new platform, it must be easy to begin rapidly developing distributed trust applications using that platform. In particular, we highlight *language agnosticism*, allowing developers to write their application in any language rather than forcing a particular language.

(2) **High performance.** The performance cost of building applications must be low compared to "rolling your own infrastructure." We assume that the network is the biggest bottleneck in distributed trust communications and ensure that the overhead introduced by relying on the platform to perform communications is minimal. Additionally, the platform should provide the ability to support multiple concurrent requests at a high throughput.

Lowering the barrier to entry is, of course, our primary goal with the platform, and we accept that enabling nicer abstractions and offloading infrastructure to a third-party platform comes with performance overheads. However, this trade-off should be minimal at best in order to respect our performance principle.

2.2 Threat Model

In the world of distributed trust, parties are segregated into separate trust domains (TDs), and the relationships of the TDs must obey the property of *isolation*: No TD is able to influence the decisions or behavior of other TDs. This is often implemented by choosing multiple cloud vendors to run distributed trust applications upon or hosting a single computing node on trusted hardware to establish a trusted TD, regardless of the behavior of other, potentially malicious TDs.

Establishing true isolation between TDs is a complex principle to analyze due to the human nature of business relationships between entities, we but consider these human factors out of scope and focus only on *technical isolation* for our purposes. Without loss of generality, we model each trust domain as consisting of a single host server running the DoTS platform and any number of DoTS applications. For this host server to be an isolated trust domain, it must be independent from the host servers of all other trust domains, and we assume no information is communicated between host servers except through the standard communication channels. To obey technical isolation, instances of the DoTS platform running on different TDs must not trust each other, and the compromise of one server instance must not lead to the compromise of others.

Additionally, we also provide for the option for multiple, untrusting applications to share the same DoTS network. Thus, we assume that all applications running on the platform may be malicious and may collude, including across trust domains. As a result, applications must also be isolated from one another, such that the malicious compromise of a set of applications does not affect the correct behavior and security of other, honest distributed trust applications running on the same network. Due to the potential multi-tenant nature of applications on the DoTS platform, there is a possibility of side-channel leakage of information from one app to another. While there is a long line of research in both the execution of and defense against such side channels [21, 27, 28, 30, 45, 46, 49], we do not consider these in scope for our design and consider only the direct attacks an application could perform against the platform itself.

Finally, single points of trust may extend beneath the DoTS platform to software and hardware such as the underlying operating system or networking hardware. A simple example of this is the Linux kernel: If two TDs' host servers both execute the Linux kernel as part of their operating systems, this means that a vulnerability in or a compromise of the Linux kernel source tree would lead to both TDs being compromised, simultaneously. Attacks on the software supply chain pose a real threat to underlying software points of attack, such as in the notorious SolarWinds hack [40].

While it is ultimately up to the TDs to ensure their hardware and software resources are as isolated as possible from other TDs (perhaps by minimizing the shared base of hardware and software resources), the DoTS platform must account for this by ensuring that its functionality isn't bound to specific third-party resources,

so that these resources may be swapped out integerchangably. We discuss our computing model more extensively in §2.4.

2.3 Networking Model

We assume that all servers within a DoTS network may directly communicate with one another over some form of network link. Since servers in different trust domains will almost certainly communicate over the wider Internet, this is equivalent to the assumption that all servers within a DoTS network are directly reachable over the Internet. Similarly, a DoTS client must be able to communicate directly with all servers within the network in order to execute a request. Several techniques are available which allow network traffic to be tunneled to an otherwise unreachable destination, from direct proxying techniques to onion services hosted over the Tor network [12]. We do not consider or evaluate these techniques as part of this paper, though proxying work may be the subject of future work.

2.4 Computing Model

The DoTS platform is designed to be run on a variety of hardware, rather than being compiled and targeted for a single instruction set architecture such as x86-64 or ARM to reduce the centralization of trust on these specific hardware vendors. All protocols and software are designed and written to be compatible between hosts running on different platforms.

Analogously, our server application relies extensively on a POSIX-compliant environment in order to provide basic functionality such as inter-process communication (IPC) and process isolation. It is tempting to use more Linux-specific features of the operating system to provide improved performance, and we recognize that Linux is the most widely used POSIX-compatible operating system in the server environment, but this leads the Linux kernel source tree to be a potential single point of attack. As a result, we constrain ourselves to the POSIX programming model rather than relying on specific features provided by the Linux kernel.

With hardware- and OS-independence, the ideal situation would be for independent TDs to run the DoTS platform on completely separate hardware architectures and operating systems, to reduce the base of trusted code that must be shared between all TDs.

Finally, the DoTS platform itself is compiled with reproducible build techniques in order to ensure that the path from source code (which can be verified by independent review) to compiled binary is reproducible. Techniques to mitigate attacks on source code distribution by means of distributed trust are discussed extensively in [10].

2.5 Application Model

We model a distributed trust application in two halves: the application client and the application servers. The client is trusted by the user, and each TD has exactly one instance of the application server. Application servers expose a set of possible RPC functions calls, typically with one function corresponding with an action the application client can take, analogous to the client and server model in HTTP.

A request in a distributed trust application as a *set* of unary function calls, one to each the host server in each TD, in the DoTS network. Each server handles a single function call, does some distributed processing, potentially involving communication with other servers in other TDs, and returns a single response.

For example, the application server in a simple semi-honest distributed trust file storage application would provide the following functions:

- $\mathsf{Store}(\mathsf{id}, \mathsf{shard})$: Store shard under the ID id. Perform MPC with the other servers to ensure that the set of all shards received by all servers are shards of a valid file.
- $\mathsf{Load}(\mathsf{id})$: Return the shard for ID id to the client.

This is a general model of distributed trust applications that is able to account for a large number of applications, though it requires a set of several, sequential unary application calls in order to model distributed trust applications where the client must be an online participant in the protocol with several rounds of communication between the client and the servers. Bidirectional streaming RPCs supported by gRPC [20] may be a future method of implementing these protocols with online clients, though we reserve this for future work.

3 DOTS'S INITIAL PROTOTYPE

In order to gain an understanding of the base system we aim to improve, we now describe briefly the original prototype designed by Tan and Kaviani.

3.1 Clients and Servers

The DoTS prototype utilizes a gRPC [20] client and server to communicate. To execute an application, the client invokes an `Exec` gRPC call to express which application hosted by the DoTS server it wishes to execute, along with the name of the function it wishes to invoke. In order to provide input and output, the client may choose to upload and retrieve "blobs" to and from the server's filesystem, and file descriptors to these input and output files are passed to the application when the client specifies it.

3.2 Application Invocation

Applications in the DoTS prototype are modeled as stateless, deterministic functions in the form of one-shot application binaries which are executed, using the input FDs as input and outputting its results to the output FDs. Since an application may comprise several functions, the function name is additionally passed to the application, which would is read by the application binary to determine what function it should perform.

In order to provide server-to-server communication, each invocation of an application begins with a process of establishing TCP connections to all other servers in the DoTS network before executing the application binary. Once the connections have been formed, the application binary is executed and passed, in addition to the input and output FDs, FDs corresponding to the TCP sockets themselves. The application may use low-level `send` and `recv` system calls in order to communicate with other servers, which provides efficiency equivalent to if the application had established the TCP connections itself.

3.3 Design Issues and Roadmap for Improvement

With this initial prototype design, we identify three primary areas for improving both the usability and the performance of a the DoTS platform:

The UploadBlob-Exec-RetrieveBlob paradigm results in high latency in function calls and race conditions. Because the original prototype did not allow for arguments in the Exec gRPC call, the only way to provide transient inputs for an application function (i.e. function arguments) was to specify those arguments as blobs uploaded in a previous UploadBlob gRPC call. While persistent storage is a key issue that the DoTS platform should address, its use as function arguments and outputs means that a minimum of 3 RTTs are required in order to execute any function with inputs and outputs. Additionally, no atomicity guarantees are provided for these inputs and outputs, resulting in potential race conditions for two clients attempting to invoke the same function at the same time, using the same filenames for their input/output blobs. A simple way to address these issues is to provide a mechanism to specify function inputs and retrieve function outputs within the Exec gRPC call itself.

The socket-passing communication mechanism provides a non-intuitive server-to-server communication primitive, lack of flexibility in transport protocols, and lack of scalability. While TCP sockets and stream sockets in general are a powerful primitive for ensuring that a stream of data is sent and received correctly, it is less useful in developing distributed trust applications. The most popular distributed communications interface today is a message-passing interface titled as such [17], which allows callers to send and receive individual messages across different servers in the network, with the message received depending on the "tag" specified in the receive call, which must match an accompanying send call's tag. Because messages within an given tag are ordered, MPI effectively multiplexes multiple ordered streams of messages onto a single TCP connection stream, which is work that the application developer would have to implement themselves in order with the basic TCP-socket-passing interface.

The socket-passing interface presents other issues. Because sockets are merely a networking abstraction of TCP provided by the operating system, there no way to pass an equivalent "TLS socket" to the application without considerable and unnecessary effort on behalf of the platform. Additionally, because the original design required invocation of application to perform its own TCP handshake and maintain its own TCP connection, the system would fail to maintain more applications invocations than the number of ports available on the system, typically around 16,384.

To solve these issues, we elect to provide a message-passing interface directly to the application communicated through a single control socket to the DoTS platform, exposed with a thin wrapper around low-level operations on this control socket. This enables the platform to maintain a single set of connections with other platform instances on other servers while handling any number of application invocations at once due to a flexible multiplexing protocol.

4 SYSTEM OVERVIEW

A distributed trust application running atop the DoTS platform is composed of three primary components:

- The **platform server**. This is a daemon that runs on several nodes, which directly communicate with each other to provide core functionality to applications such as node communication and synchronization.
- The **application server**. This is a daemon running on the same node as the platform server, written by the application developer and integrated with the platform server using a low-level protocol or dedicated library.
- The **application client**. This is a client that communicates with the application server by issuing gRPC [20] calls through the platform server.

We note that we do not currently include an analogous platform client library. We believe that providing the protocol buffer definitions and their associated gRPC function definitions is sufficient to develop a client application, since gRPC stubs can be created automatically using code generation available with protocol buffer compilers, and the work needed to implement a gRPC client that makes several concurrent calls to several gRPC servers is minimal enough and dynamic enough depending on the application that we consider it sufficient to offload that work to the developer.

The key aspect of this DoTS platform design is the separation between the platform server and the application server, which serves two purposes. First, it offloads key tasks such as message passing, request handling, and output buffering off to the platform server, rather than relying on the developer to correctly design and implement these functionalities on their own. Second, it allows for the modularity of application and platform server code, allowing multiple different applications to share the same networking stack and the same hardware across multiple trust domains, which makes it easier for distributed trust app developers to deploy their apps to multiple, pre-existing trust domains rather than requiring them to set it up from scratch.

For the platform server and application server to communicate, they set up a Unix-domain socket known as the control socket in order to pass control messages back and forth. The protocol used to communicate through the control socket is described in detail in §6.

A typical client-server DoTS platform request follows the following steps:

1. The application client makes a unary gRPC call to each server part of a DoTS network, including the application it wants to call, the function it wishes to invoke, and any arguments to be passed to the application server function.
2. The platform server receives this gRPC call and forwards the function name, arguments, and client details to the application server through a UNIX-domain socket.
3. The application server handles the client's request. As needed, it calls into the platform server through the control socket socket in order to perform communication between other nodes' application servers and other miscellaneous tasks.
4. The application server provides its output to the platform server through the control socket.

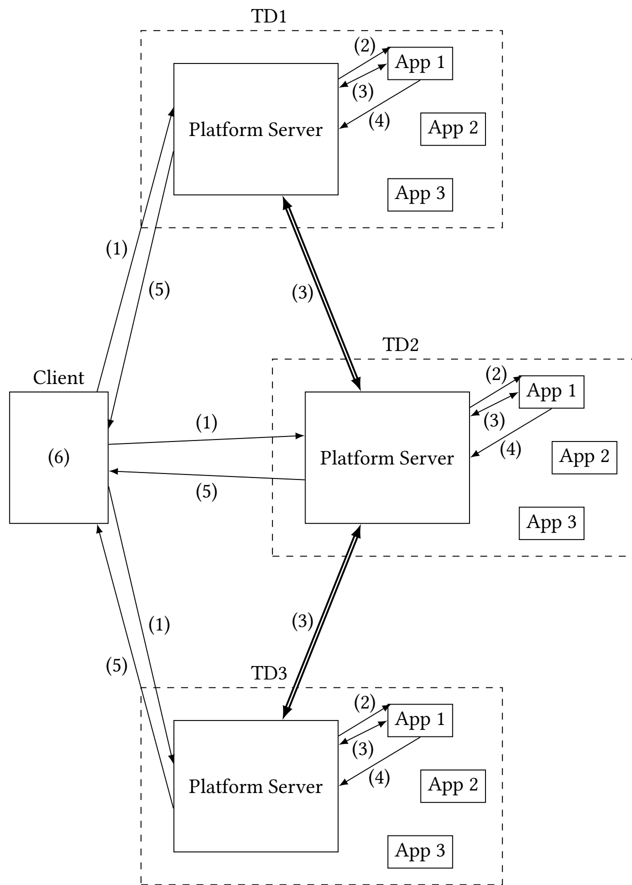


Figure 2: A walkthrough of the steps involved in a DoTS request and response. (1) The application client makes a request to all servers in the DoTS network. (2) The platform server forwards the request to the appropriate application server. (3) The application server may make various calls the platform server in order to perform communication or other platform functions. (4) The application server provides its output to the platform server. (5) The platform server forwards the results back to the client. (6) The client performs local computation from the servers' responses to produce the final result.

(5) Once the application server signals that it is finished processing its task, the platform server returns the gRPC response back to the client containing the application server's result code and output.
(6) Once the client receives all gRPC responses from the platform servers, it performs any final local computation in order to produce the final result.

An illustration of these steps is provided in Figure 2.

5 PLATFORM SERVER ARCHITECTURE

We now describe in detail the design and construction of our DoTS platform server.

```
---
peer_bind_addr:
peer_port: 51000
peer_security: tls

grpc_bind_addr:
grpc_port: 50050
grpc_security: tls
grpc_tls_cert_file: snakeoil-cert.pem
grpc_tls_cert_key_file: snakeoil-key.pem

file_storage_dir: ./files

limits:
  app_msg_buffer_megabytes: 1024
  app_restart_timeout_seconds: 1
  app_max_restarts: 10

nodes:
  sky:
    addr: sky.cs.berkeley.edu:51000
  rise:
    addr: rise.cs.berkeley.edU:51000
    peer_tls_cert_file: rise-cert.pem

apps:
  skrecovery:
    path: ./applications/skrecovery/a.out
  signing:
    path: ./applications/signing/a.out
```

Figure 3: A sample configuration file for the DoTS platform server, written in YAML.

5.1 Configuration and Initialization

The behavior of the platform server is principally governed using a configuration file, an example of which is given in Figure 3. The configuration defines the peer nodes in the network that it should establish connections with, configuration parameters for the connections themselves, and definitions for the application servers it should spawn and manage.

Upon initialization, the application server establishes TCP connections with other nodes present in its configuration file. All nodes within a network are expected to be configured with the same peers, so other nodes will be establishing such connections with all other nodes in the network to form a complete graph of pairwise connections. The use of these connections is detailed in §5.2, and establishing security over these connections via TLS is described in §5.3. Since a node acting as a TCP listener may not be able to identify several TCP clients connecting at the same time, the first message the client sends to a TCP listener is its gob [19]-encoded node ID, which the server uses to identify the client at the other end of an incoming connection.

This initialization routine additionally includes spawning the application servers and establishing connections with them using

control sockets. The platform server additionally automatically waits on the spawned application processes as its children waiting on them to reap them if they crash due to an application bug or malicious behavior and restarting them as needed. A configurable restart count and restart delay is available to administrators in order to prevent a malicious application from immediately crashing and and being restarted in order to consume resources.

5.2 Message-Passing Over TCP

The stream abstraction provided by the established connections is used to execute a simple message-passing protocol used by applications and various aspects of the platform itself. The design we borrow from is heavily inspired by the Message Passing Interface [17], a standardized interface used to implement distributed systems using message-passing techniques. For now, we implement only basic point-to-point communication with Send and Recv, analogous to MPI's MPI_Send and MPI_Recv, and messages may be sent and received in any order by attaching a *tag* to each sent message, with each corresponding receive specifying the tag for which it wants to receive a corresponding message.

We implement this basic point-to-point communication by serializing the stream of messages along the established connection, which are subsequently deserialized and handled by the receiving platform server. As of now, all servers are implemented using the same programming language, Go, so we utilize the gob serialization and deserialization protocol [19] in order to implement this stream of messages. We leave to future work to construct a more general protocol that works across languages to enable the prospect of multiple independent implementations.

Because servers located in different TDs likely communicate with each other over the wider Internet, where network latency is likely to dominate over additional latency imposed by excess memory copies or communicating the received data through IPC to application servers, so we choose to implement full buffering of messages on the receiving end rather than imposing the "ready send" mechanism of traditional MPI, where the send may block until the receive has posted. This makes the development of platform server features easier and provides a more programmer-friendly API for application developers, as both application may simply execute a Send followed by a Recv.

In order to prevent a malicious application or malicious platform server peer from consuming an unbounded amount of memory by posting many Send operations without a corresponding Recv, messages are separated into one or more *communicators*, where each communicator provides the Send and Recv interface to its caller. The platform server itself uses a single communicator (the *platform communicator*) for exchanging messages between the platform servers themselves, and each application server is allocated a single communicator (the *application communicator*) for its application-specific communication.

Communicators are responsible for the accounting of how much data has been buffered at a given time, implemented using atomic add and subtract operators when buffering a received message and removing a received message from the buffer. If the buffered message size increases past a configurable value, the communicator is destroyed, and any users of the communicator are terminated.[1]

5.3 Server-to-Server TLS

In order to secure platform and application communications against a malicious network fabric, we provide a configuration option to use TLS with mutual authentication [37] over the existing TCP connections established during the initialization phase. In a mutually authenticated TLS connection both parties present their TLS certificates as part of the TLS handshake, rather than simply authenticating the server in standard TLS.

By default, the TLS authentication mechanism uses standard DNS-based authentication described in [39] and authenticates by establishing a chain of trust to the root certificate authority (CA) pool present by default on most systems. As mentioned in §5.1, since the TLS listener may not be able to identify the client until after the handshake completes, so the TLS client will verify the listener's TLS certificate during the TLS handshake, and the TLS listener will verify the client's TLS certificate only after the client has sent its node ID, with which the server can look up the configured hostname for that node ID. If verification fails, the listener aborts the connection.

We recognize, however, that this scheme establishes CAs as a central point of attack, since they may accidentally or maliciously issue false certificates used to conduct a MITM attack on the these TLS connection. Techniques to provide certificate transparency and auditing and reduce the burden of CAs as a central point of attack are a subject of extensive research [29], and we leave this as future work for now. As a stop-gap measure, we provide a mechanism to pin the certificates used to establish these TLS connections rather than relying on certificates signed by central authorities, and operators of the DoTS platform server instance may choose to communicate these pinned certificates out of band if they choose.

5.4 Client-to-Server TLS

Analogously to the server-server connection security in the previous section, we also provide a configurable option to secure gRPC requests using TLS. gRPC supports server authentication using TLS natively [20], and we simply expose that as a configurable option to server operators. It is up to the application developer to decide how to authenticate these TLS certificates on the client, since these authentication mechanisms can vary greatly, and we provide no client library beyond those automatically generated from the gRPC definitions, which expose all necessary functionality automatically.

6 APPLICATION ARCHITECTURE

Applications within the DoTS platform are written by application developers building on top of resources provided the DoTS platform and project itself. These applications are generally composed of 2 parts: the application client and the application server. Analogously, developers using the DoTS platform are provided two main repositories of material to build upon: the application client gRPC library,

[1]In the future, in the case of a malicious platform server instance sending many messages in order to consume a large amount of memory, we would like to simply disconnect this node from the network and inform applications of the updated network topology. This is not supported as of now, and the platform server itself will simply be terminated. This is fine since we rely on other platform servers for availability.

```
service DecExec {
    rpc Exec(App) returns (Result) {}
}

message UUID {
    fixed64 hi = 1;
    fixed64 lo = 2;
}

message App {
    UUID request_id = 7;
    string app_name = 1;
    string func_name = 3;
    repeated bytes args = 8;
}

message Result {
    int32 code = 3;
    bytes output = 2;
}
```

Figure 4: The Protocol Buffers gRPC definition for the Exec RPC.

called dotspb [14], to execute requests to the platform servers and the application server library, called libdots [15], for the application server to interact with the platform server. We go into the design of the client dotspb library in §6.1, the design of the server libdots library communication which exposes platform functions and communication functions in §6.3, and the low-level protocol used by libdots to communicate directly with the platform server in §6.3.

6.1 Client gRPC Library: dotspb

dotspb (short for *DoTS Protocol Buffer*) is the repository that exposes Protocol Buffer and gRPC definitions to application developers. Developers are expected to use these definitions in order to construct application clients using these gRPC definitions, and we currently provide vendored versions of these generated gRPC clients and servers for Go, using Google's own `protoc-gen-go` [4], and for Rust, using `tonic` [7].

The main RPC call that we expose is the Exec call, taking as input an App containing the request ID, app name, function name, and function arguments and returning the bytes in the output. The schema definition in the `proto3` format is given in Figure 4. The design of these input arguments and outputs was chosen to mirror the development of a standard command-line application, which typically takes as input a number of command-line parameters in the form of strings (the `argv` arguments to the `main` function in a standard C program) and outputs a stream of bytes to the standard output of the application, so that application developers are familiar with this input/output interface. Application clients that want the ability to control the structure of their inputs more precisely may use any validation and structuring of their inputs as they desire (for example, using their own protobuf definition and deserializing the string of bytes immediately upon receipt of the request).

6.2 High-Level Server Application Library: libdots

libdots is a library which the application server uses to communicate with the platform server, in order to provide core functionalities needed by many distributed trust applications. It is difficult to altogether avoid code that must be written, compiled, and loaded into the application server binary itself, so in order to come as close as possible to maintaining our goal of language-agnosticism, we set the following objectives:

1. Constrain libdots to be as small as possible, preferring to offload any functionality to the platform server itself where possible.
2. Write the libdots library in C in order to be as portable as possible. Compiled C code can often simply be imported into many languages using foreign function interfaces (FFI).
3. Establish a well-documented protocol which libdoes uses to communicate with the platform server through the control socket so that a native version of libdots may be easily implemented in other languages. This is the reference given in §6.3.

The job of the application-platform interface is to provide three key functionalities: environment initialization and deinitialization, request handling, and communication of application servers across different servers/TDs. Our interface, is, in turn, broken down into three main parts: the environment API, the request handling API, and the communications API. We now describe in detail the design of these APIs in the C language.

6.2.1 *Initialize Runtime.*

```c
int dots_init(void);
```

This function must be the first function called by the application in order to initialize the libdots runtime retrieve the startup environment for the application server. Currently, the environment consists of two values: The number of servers in the DoTS network (the "world size") and the index of the current server (the "world rank") between 0 (inclusive) and the world size (exclusive).

After this is called, all other libdots functions may be called.

It returns 0 upon success and a negative error value upon failure.

6.2.2 *Get Environment Parameters.*

```c
size_t dots_get_world_rank(void);
size_t dots_get_world_size(void);
```

These functions return the rank of the current DoTS node (between 0 and the world size minus 1) and the total number of nodes in the DoTS network, respectively.

6.2.3 *Deinitialize Environment.*

```c
void dots_finalize(void);
```

This function should be called immediately before the program exits. It frees memory allocated by `dots_init` and indicates to the platform server that the application is about to shut down cleanly.

6.2.4 Accept Request.

```
int dots_request_accept(dots_request_t *req);
```

This function blocks until a request is received from a client. It populates the request information in `req`.

The fields of the `dots_request_t` struct are directly accessible and provide to callers the values of the request ID, the called function name, and the function arguments. These function arguments are provided in terms of pointer-length pairs to represent the raw bytes passed by the clients.

The bytestring arguments are additionally guaranteed to be followed by a NULL byte, one byte past the end of the length of the buffer. This means that an application that wishes to use its arguments as NULL-terminated string arguments can directly use these arguments without having to copy them to a new buffer and manually NULL-terminate them. Because it length of the arguments themselves aren't affected, arguments in the form of raw bytes are not affected by this optimization.

It returns 0 upon success and a negative error value upon failure.

6.2.5 Output Bytes.

```
int dots_output(dots_request_t *req,
        const unsigned char *data,
        size_t data_len);
```

This function outputs `data_len` bytes starting from `data` to the output stream of the request `req`.

It returns 0 upon success and a negative error value upon failure.

6.2.6 Output Formatted String.

```
int dots_outputf(dots_request_t *req,
        const char *fmt, ...);
```

This function evaluates the format string given by `fmt` with the additional arguments passed to the function and writes the resulting string to the output stream of the request `req`, excluding the terminating NULL byte.

While this function isn't strictly necessary and thus violates our principle of minimizing the size of libdots as much as possible, it provides an extremely convenient abstraction to programmers looking to program an application server in C.

It returns 0 upon success and a negative error value upon failure.

6.2.7 Finish Request.

```
int dots_request_finish(dots_request_t *req,
        int code);
```

This function indicates to the platform server that the request has finished processing the request `req` and may return any output, along with the provided result `code`. This function does *not* deallocate any resources, and `req` may still be accessed after this function is called. The reason for this separation is that we do not want deallocation to happen in a function that can potential return an error.

It returns 0 upon success and a negative error value upon failure.

6.2.8 Deallocate Request.

```
int dots_request_finalize(dots_request_t *req);
```

This function deallocates resources allocated by `dots_request_accept` in the request `req`. Note that this function does not automatically call `dots_request_finish`, and both functions must be called for the correct handling of a request.

6.2.9 Send Message.

```
int dots_msg_send(dots_request_t *req,
        const void *buf, size_t len,
        size_t recipient, int tag);
```

This function sends `len` bytes starting from `buf` from the current application instance to the recipient with the given rank `recipient` and message tag `tag`. If `req` is non-NULL, the message is associated with the given request and can be received with `dots_msg_recv` calls associated with the same request. Else, if `req` is NULL, the message is sent independent of any request and can be received with `dots_msg_recv` calls also passed a NULL request.

If multiple messages are sent with the same `req` and `tag`, these messages will be strictly ordered, such that the first sent message will always be received before the second sent message.

Unlike the familiar `MPI_Send` that programmers might be familiar with, this is a fully buffered send that returns immediately without blocking. Details of this buffering are described in detail in §5.2, since this is the underlying message passing mechanism on the platform server.

This function returns 0 upon success and a negative error value upon failure.

6.2.10 Receive Message.

```
int dots_msg_send(dots_request_t *req,
        const void *buf, size_t len,
        size_t recipient, int tag,
        size_t *recv_len);
```

This function receives a maximum of `len` bytes into a buffer starting at `buf` from the sender with rank `sender` and message tag `tag`. The actual number of bytes received is written to `recv_len`.

See §6.2.9 for more details about the corresponding send API, which also describes how messages are received.

This function returns 0 upon success and a negative error value upon failure.

6.3 Low-Level Application-Platform Server Protocol Design

We now describe the low-level protocol used to communicate control messages between the application server and the platform server, through the Unix-domain control socket.

The general structure of these control messages is given in Figure 5. All integers are in big-endian to ensure portabilty across platforms and to ensure independence of the protocol from the platform itself. In general, the message is broken into 3 main parts:

- A 64-byte, structured *header* containing values that must be present in every message type.

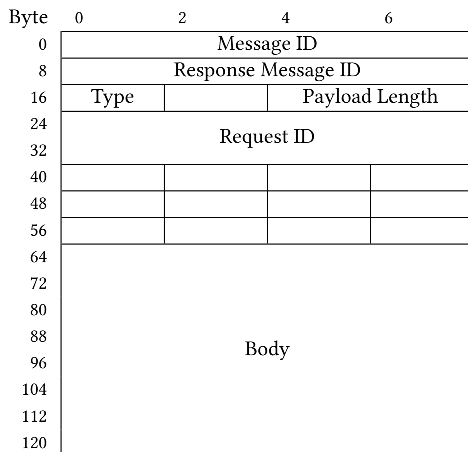


Figure 5: Layout of a control message read from and written to the control socket by the platform and application servers.

- A 64-byte *body* whose structure depends on the type of the message.
- A variable-length *payload* whose length is defined by a value in the header and whose interpretation also depends on the type of the message.

The reason for the distinction between a message body and a message payload is two-fold. First, ensuring that a relatively large, pre-allocated space for bits that can be interpreted differently depending on the message type means that, for most message types, only a single `recv` system call needs to be made, rather than forcing the usage of two distinct `recv` calls for every single message in order to receive a payload containing type-specific information. Second, including the message payload as a "first-class" member of request keeps the critical path between reading the fixed-sized portion of the message (the header and body) as short as possible, rather than requiring relatively expensive logic to parse out the payload length before the bytes of the payload can be received. Because reading the message header and body and reading the payload must happen in a single, atomic action requiring a lock, keeping this critical path as short as possible is ideal in a multi-threaded environment.

Table 1 lists the valid control message types as well as interpretations of the message body and payload in those specific message types. The name of the message type is given, integer value of the type encoded in the type field of the message header and interpretation of the bits in the message body and payload.

For messages marked as triggering a response, we refer to the initial message as the *trigger message*, and we refer to the message sent in response (the message type suffixed with `_RESP`) as the *response message*. This response message will have the trigger message's message ID in the `Response message ID` field of the message in order to identify which message it is in response to.

For each trigger message, at most one response message will ever be sent. In single-threaded contexts where at most one trigger message will be sent at a time, it is sufficient to expect that the next received message will be the response message when receiving from the control socket. However, in multi-threaded contexts, where multiple trigger messages may be sent at once before responses are received, response messages may not necessarily be received in the same order as the corresponding trigger messages are sent. Because at most one thread should receive from the control socket at a time, the control message receive handler must implement a simple buffering mechanism to buffer response messages intended for other threads. This buffer need not be any larger than the larger possible number of outstanding trigger messages awaiting a response, typically equal to the number of program threads in the application.

7 IMPLEMENTATION

Our implementation of the DoTS platform is separated into 3 repositories: `dots-server` [16] containing the platform server implmentation, `libdots` [15] containing the application server library (§6.2), and `dotspb` [14] containing the client application protos and vendored Go and Rust generated clients and servers.

Our server consists of ~1.6k source lines of code (SLOC) in Go, and our libdots implementation consists of ~670 SLOC in C, as counted by CLOC [9]. We describe the implementation of several distributed trust apps in detail in §8.1, as we consider ease of implementation as a core evaluation metric of this project.

8 EVALUATION

In this section, we answer the following questions to evaluate whether our redesign of the DoTS platform satisfies the two objectives posed in §2.1: 1) Does the DoTS platform improve the relative ease of programming distributed trust applications?, and 2) Does the DoTS platform provide a level of performance comparable to that of designing an ad-hoc networking stack and encryption layer?

To answer both of these questions, we implemented two distributed trust applications, semi-honest secret key recovery using Shamir's secret sharing [41] and (t, n)-threshold ECDSA signatures based on the Gennaro & Goldfeder (CCS '20) [6], both in Rust. For the former, we implemented the protocol from scratch using elliptic curve primitives provided by the `elliptic-curve` crate [2] to evaluate the effort needed to implement a small-scale distributed application from scratch, and for the latter, we pulled an off-the-shelf GG20 implementation [47] to evaluate the effort needed to connect an existing MPC library as a DoTS server application.

Finally, to evaluate the complexity of porting an existing, large-scale MPC library with a large, pre-existing networking codebase to the DoTS platform, we additionally ported the MP-SPDZ framework [25] to be implemented as a DoTS application server and ran the default `tutorial` MP-SPDZ application as well as the more complex `aes` and `hmac` applications.

8.1 Ease of Use

We aim to answer the first question by counting source lines of code (SLOC) needed to produce each of the three resulting DoTS applications. To do this, we use CLOC [9] to count the lines of code needed to implement the secret key recovery and (n, t)-threshold ECDSA applications, which counts lines of code without including blank lines or comments. For MP-SPDZ, we use a combination of CLOC and the functionality provided by `git diff` in order to count the number of lines of code changed needed to port MP-SPDZ to

Name	Numeric Type	Sender	Body Values	Payload Contents	Triggers Response?
MSG_SEND	2	Application	recipient (32), tag (32)	Message data	
MSG_RECV	3	Application	sender (32), tag (32)		*
MSG_RECV_RESP	4	Platform		Message data	
OUTPUT	5	Application		Output data	
REQ_ACCEPT	6	Application	code (32)		*
REQ_ACCEPT_RESP	7	Platform		Request data	
REQ_FINISH	8	Application			

Table 1: Table of control message types used by libdots to implement communication with the platform server. The Sender column indicates whether the control message is sent from the application to the platform or from the platform to the application, the Body Values column indicates any values present in the fixed-size message body, the Payload Contents indicates the interpretation of the data stored in the message payload, and the Triggers Response? column indicates whether this kind of message triggers the other party to send a corresponding message of same type suffixed with the `_RESP` suffix.

use DoTS as a transport mechanism.transport mechanism. The MP-SPDZ source tree includes a `Networking` folder which we estimate could be eliminated entirely in favor of a simple networking abstraction that underlyingly uses DoTS messages and calls to libdots, so we estimate that all lines of code could be eliminated from this folder, which we count using CLOC. Due to the way the MP-SPDZ framework was written, it be too difficult for us to fully go through and replace all instances of the ad-hoc networking functionality with libdots wrappers, which is why we rely on these estimation techniques for MP-SPDZ.

These results are given in Table 2. Overall, these data show that the it is relatively easy to both implement new distrubted trust applications and port existing distributed trust libraries and frameworks to use the DoTS platform. Of the two applications that we wrote from scratch, both were easily written in under 500 lines of code with only two source files (one for the client and one for the server). For our porting of MP-SPDZ, the task was more difficult with almost 500 lines of code across 44 files, but we estimate that, had the MP-SPDZ source code implemented better abstraction boundaries between networking and cryptographic code, this number could have been far fewer, as few as a single file modification and deleted extraneous files.

8.2 Performance Evaluation

We expect that using the DoTS platform will introduce some amount of overhead compared to ad-hoc networking code, since messages must travel through two additional layers of IPC and an additional layer of buffering on the receiving end in order for a message to be passed from client to server. In order to measure the impact of this additional overhead, we run MP-SPDZ's implementation of MASCOT [26] in both unmodified MP-SPDZ with its ad-hoc networking and our port of MP-SPDZ as a DoTS application server. These MP-SPDZ applications are run in two settings:

- A "low-latency" setting where 4 servers are located on the same machine. The aim of this setting is to is to get a quantitative measurement of the absolute amount of overhead added by the DoTS over ad-hoc networking. Here, we utilize standard TCP connections the localhost sockets to remove the cryptographic overhead imposed by using TLS to better isolate the DoTS networking overhead itself.
- A "realistic" setting where 2 servers are located in geographically distinct locations. The aim this setting is to show how much of a performance impact the DoTS platform imposes in a close-to real-world application deployment scenario with the introduction of latency introduced by the wider Internet. Here, we enable TLS on the DoTS platform server connections to provide anc confideniality for the traffic flowing across the Internet.

The software, hardware, and network link information used in our evaluation is given in Table 3. For both settings, we compiled and ran three of the MPC programs provided in the MP-SPDZ repository, `aes` [35], `dijkstra_example` [11], and `htmac-1-1-10` [38].

Figure 6 shows the total running time of our applications on in the low-latency setting. Even in this extreme case of removing the inherent networking latency of the Internet entirely (a fairly unrealistic setting given that different trust domains must almost always be in geographically separate locations), the DoTS platform adds only ∼47% overhead over the ad-hoc networking setup of MP-SPDZ. Cryptographic protocols that are less sensitive to latency due to fewer rounds of commuincation, like Yao's garbled circuits [48] would likely fare even better in this setting, since the throughput of local communication is high for any form of local computation.

Figure 7 shows the running time of applications in the realistic setting. These data show that while the DoTS platform does indeed add additional overhead to communication between servers, this overhead is largely imperceptible when running distributed trust protocols in a realistic setting, with geographically separated servers. The difference between MP-SPDZ's ad hoc networking and the DoTS platform in these data is as little as 6.4% in the AES benchmark and a 9.5% difference on average, with the positive trade-off being that the application developer enjoys a much easier

[2]This line count was generated using `git diff --stat` in order to count the number of lines changed to the source code of the repository to integrate the DoTS platform communication functionality.

[3]This line count was generated as the SLOC of the `Networking` folder, which we estimate could be eliminated entirely when integrating MP-SPDZ as a DoTS application server.

[4]This is an AWS t3a.xlarge EC2 instance hosted in region us-west-1b.

Application	Language	Lines Added	Lines Removed	Files Changed
Secret Key Recovery	Rust	485	—	2
(n, t)-Threshold ECDSA	Rust	435	—	2
MP-SPDZ	C++	489[2]	2,611[3]	44

Table 2: Lines of code needed to implement and/or port distributed trust applications to an application server on the DoTS platform. The Files Changed column counts only source files, not including metadata files such as dependency management or version control files.

ID	Processor	Memory	OS	Downlink (Mbps)	Uplink (Mbps)	RTT - Server 0 (ms)	RTT - Server 1 (ms)
0	AMD Ryzen 5 5600X	64 GB	Debian 11	513.40	362.98	—	8.478
1	AMD EPYC 7571[4]	16 GB	Ubuntu 22.04	605.70	373.63	8.478	—

Table 3: Evaluation hardware, software, and network setup.

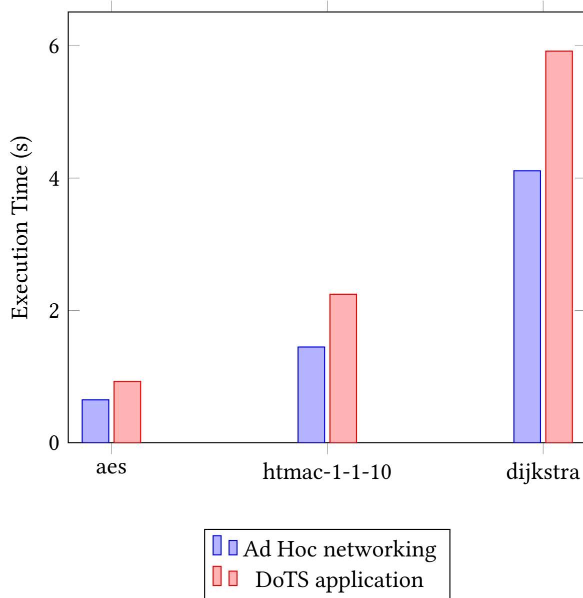


Figure 6: Running time, in the low-latency, local setting, of MP-SPDZ applications as a standalone binary using its own ad hoc networking library as a DoTS server application.

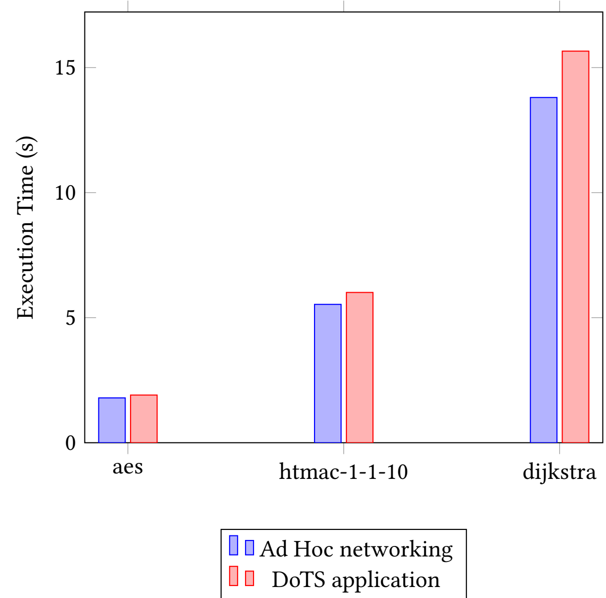


Figure 7: Running time, in the geographically separated, realistic setting, of MP-SPDZ applications as a standalone binary using its own ad hoc networking library as a DoTS server application.

development and deployment experience as shown in the previous section.

We note that our realistic testing methodology still provides a quite liberal estimate of the overall incurred overhead of the DoTS platform, since the two servers used in our testing are geographically close (one in Berkeley and one in Northern California) with a relatively low RTT. A higher RTT between the two servers would likely decrease the perceivable effect of the overhead incurred by the DoTS platform even further.

9 RELATED WORK

While the line of distributed trust research and spans many different fields of study from distributed systems design to theoretical research on protocols, the research space investigating the practicality of implementing distributed trust protocols is a lot smaller, though several parallel lines of work exist with the same aim of making distributed trust easier to deploy in the real world.

9.1 Dotme

Perhaps the most immediately adjacent work in this line of research is Dotme, which aims to provide an easier method of deploying multiple independent trust domains for service providers to leverage the properties of distributed trust. A traditional distributed trust deployment relies on an application developer developing business relationships with other third parties to execute their distributed

trust protocols without giving the application developer access to the third party's servers (effectively deploying their app while maintaining their independence as a trust domain). DotMe proposes the use of on-demand VM allocation on cloud providers by the user in order to establish these trust domains, service provider automatically begins federating with these completely separate trust domains (cloud VMs which the service provider exerts no control over) in order to deploy distributed trust.

While DoTS does not attempt to solve the difficult issue of deploying multiple, separate trust domains, it shares with it the common goal of lowering the barrier to entry to the development of distributed trust applications.[5] In addition, Dotme provides a "serverless" model, where distributed trust apps run on cloud providers effectively as cloud functions, automatically allocating and deallocating on-demand without any persistent state on the server-side. This makes the implementation of some distributed apps requiring server-side secret state more difficult in the Dotme framework, while server-side state comes for free in DoTS due to the persistent nature of the platform and application server daemons.

9.2 MPCAuth

As authentication is a mechanism that is required by many distributed trust applications, a protocol to implement scalable authentication to a large number of distributed trust nodes simultaneously greatly improves the barrier to entry. MPCAuth (formerly known as N-for-1 Auth) is a protocol that allows for the authentication of a client to an arbitrary number of servers with only a constant amount of *work* required by the user, both using traditional, password-based authentication and more authentication protocols such as multi-factor authentication via email or U2F.

Analogously to DoTS, it is a protocol that is able to replace the ad-hoc authentication mechanisms developed for distributed applications, allowing application developers to focus more time on writing application logic and the core of the distributed trust application itself, rather than on infrastructural issues such as authentication.

10 FUTURE WORK

In many previous sections of the paper, we have identified many potential areas of future work with regards to the DoTS platform. We use those remarks and several other potential areas of development in order to guide future work and research into the DoTS platform and other mechanisms to ease the development of distributed trust applications.

10.1 Global-Scale and BFT Applications

Work on Tor [12] and other protocols requiring global-scale communication such as Bitcoin [34] and IPFS [44] has led to the establishment of protocols where large numbers of parties across the Internet must communicate with one another to implement distributed trust applications. Byzantine fault tolerance, is generally a property shared by these applications, such that the application still makes forward progress despite the failure or malicious behavior of one or more nodes. In many cases, there is no current method to easily translate those applications into a DoTS application, and much of the work in optimizing these communications protocols for the global scale may be useful to integrate into DoTS.

10.2 Practical DoTS Platform and Application Deployment

While much of the core DoTS platform and some key applications has been written and can be run on commodity hardware, deployment is not yet a streamlined process. Further, the coordinated and manual editing of configuration files in order to deploy applications means that application developers must still develop business relationships with providers (runners of DoTS platform instances) in order to deploy their applications, which has been identified as a source of difficulty in the deployment of distributed trust applications [24]. Technologies like Docker [13] and Kubernetes [3] have found much of their success due to the streamlined packaging, distribution, and deployment of applications onto physical compute hardware, and the notion of a "publicly available DoTS network" onto which developers could deploy arbitrary distributed trust applications (perhaps for a small fee) would mean that a independent developers could easily deploy distributed trust applications with strong guarantees of security with minimal barrier to entry.

As a stepping stone to this goal, removing the infrastructural guarantee that all DoTS nodes must be publicly accessible would additionally aid the deployment of DoTS platform instances in the first place. As mentioned in §2.3, a simple way to achieve this would be a publicly available relay to tunnel platform traffic behind a firewall or NAT, or DoTS nodes could be set up to act in a secondary capacity as a relay.

10.3 Interactive Client Applications

§2.5 discussed the application model, in which the client provides a single unary input and receives a single set of outputs from the application servers. This model, while general, does not encompass protocols where the client must be an active participant in the protocol and can currently only implement such protocols as a sequence of chained unary calls, which is cumbersome and non-intuitive. gRPC allows for bidirectional streaming RPC in addition to unary ones [20], which seems to be a natural solution to this issue.

10.4 Securing Server-to-Server Communication using Distributed Trust

Simply using TLS to secure sever-to-server and client-to-server communications ultimately leaves much to be desired, leaving CAs as a central point of attack for a network attacker to compormise the system. Technologies like certificate transparency [29] provide some mitigation against these forms of attack in the form of auditing, but there is currently no integration of these TLS-related distributed trust mechanisms into DoTS.

10.5 Secure Enclaves

Secure enclaves provide a useful mechanism of compiling and executing code in a manner such that a remote client can ensure that the software is executing unmodified on a remote host via

[5] Indeed, the two projects share a common heritage, and the authors of both projects collaborated closely on these projects.

attestation and that a malicious host operating system can't observe or tamper with the memory contents of the enclave. Using a combination of secure enclaves like Intel SGX [22] and oblivious computation techniques like ORAM [18], it is possible to construct a local trust domain whose security is connected to that of the hardware enclave manufacturer, rather than that of the host, such as is proposed in [10]. The ability to embed key functions of the DoTS platform server (or application servers) into a secure enclave would bolster the security guarantees of the DoTS network and provide applications developers with an easy way to leverage such technologies.

11 CONCLUSION

Many of the barriers to writing distributed trust applications are infrastructural, rather than theoretical. Many of these infrastructural needs can be addressed with DoTS, though its original prototype implementation left many potential ares of improvement. With the modifications and updated design presented in this paper, DoTS is able to provide a single, out-of-the-box solution for developing distributed trust applications. We discussed our updated design design and implementation details of the DoTS platform, which ultimately exposes a simple, general, and flexible programmer interface which supports a wide range of distributed trust applications. We showed that the core focus of DoTS is maintained, showing that applications are simple to write, requiring less than 500 lines of integration code to implement common protocols such as secret sharing or (n, t)-threshold ECDSA. We also showed that our interface is general enough to support the porting of existing distributed trust applications like MP-SPDZ into the DoTS platform, simply replacing its ad hoc networking with calls out to the DoTS platform. Finally, we showed that while the improved design of the DoTS platform introduces a nominal amount of overhead, this overhead is quickly dominated by the latency of network traffic going across the Internet, upper bounded by 9.5% in our most generous realistic testing model.

Ultimately, the usability of the DoTS platform will depend largely on the future work of the platform. While networking is a core issue that we addressed in the course of this paper, other features like application storage and rapid deployment to shared trust domains will be likely be needed before our platform will be deployable in the real world.

Yet, we expect that the world of widespread distributed trust adoption is not too far off. The computer security research community has already solved most of the theoretical barriers to deploying distributed trust. What remains is the solutions to the practical barriers to distributed trust deployment, and the DoTS platform presents a viable, extensible, and rapidly developing solution to address these practical needs of developers.

ACKNOWLEDGMENTS

This project would not be possible without the collaboration of many of my peers in the Sky Lab at UC Berkeley:

- My peers Sijun Tan and Darya Kaviani collaborated closely with me in discussing and executing many of the updated design decisions to the DoTS paltform as we worked to find the best API to suit the needs of distributed trust apps.
- Emma Dauterman, Mayank Rathee, Katerina Sotiraki, Sam Kumar, and Sijun Tan were the originators of the DoTS platform idea. They provided the high-level vision of making distributed trust accessible to non-expert application developers.
- Mayank Rathee, Katerina Sotiraki, and Sijun Tan were the initial designers and developers of the DoTS platform prototype.
- Allison Li, Yuwen Zhang, and Michael Ren are primarily responsible for the implementation of the secret key recovery DoTS application, under the guidance of Emma Dauterman.
- Darya Kaviani is primarily responsible for the implementation of the (n, t)-threshold ECDSA signing DoTS application.
- Many of the ideas we had came from our collaboration with the students of CS 294-163 in the Fall 2022 semester, taught by Raluca Ada Popa and Emma Dauterman and assisted by Mayank Rathee. While much of their work was ultimately deprecated as we continued to iterate upon the core platform, all of the ideas generated from those projects were incredibly helpful.

Finally, I'd also like to thank my advisor, Raluca Ada Popa, for her guidance both in academic understanding of notions of distributed trust and in practical mentorship in the life of a graduate student. My friends in Cal Christian Fellowship and my family, too, have also been a huge part in keeping me sane during the writing of this thesis. Thank you to all of you!

REFERENCES

[1] Apple and Google. 2021. *Exposure Notification Privacy-preserving Analytics (ENPA) White Paper.* Technical Report. Apple. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf
[2] Tony Arcieri. 2023. https://docs.rs/elliptic-curve
[3] The Kubernetes Authors. 2023. https://kubernetes.io/
[4] Protocol Buffers. 2023. https://github.com/protocolbuffers/protobuf-go
[5] Vitalik Buterin. 2014. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.* Technical Report. Ethereum Foundation.
[6] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. 2020. UC Non-Interactive, Proactive, Threshold ECDSA with Identifiable Aborts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20).* Association for Computing Machinery, New York, NY, USA, 1769–1787. https://doi.org/10.1145/3372297.3423367
[7] carllerche and LucioFranco. 2023. https://docs.rs/tonic
[8] Open Enclave contributors. 2023. Open Enclave SDK. https://github.com/openenclave/openenclave
[9] Al Danial. 2023. https://github.com/AlDanial/cloc
[10] Emma Dauterman, Vivian Fang, Natacha Crooks, and Raluca Ada Popa. 2022. Reflections on Trusting Distributed Trust. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (Austin, Texas) *(HotNets '22).* Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3563766.3564089
[11] E. W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numer. Math.* 1, 1 (01 Dec 1959), 269–271. https://doi.org/10.1007/BF01386390
[12] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *13th USENIX Security Symposium (USENIX Security 04).* USENIX Association, San Diego, CA, 18 pages. https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router
[13] Docker, Inc. 2023. https://www.docker.com/
[14] Sky DoTS. 2023. https://github.com/dtrust-project/dotspb
[15] Sky DoTS. 2023. https://github.com/dtrust-project/libdots
[16] Sky DoTS. 2023. https://github.com/dtrust-project/dots-server
[17] Message Passing Interface Forum. 2021. *MPI: A Message-Passing Interface Standard, Version 4.0.* Technical Report. Message Passing Interface Forum. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
[18] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (may 1996), 431–473. https://doi.org/10.1145/

233551.233553
[19] Google. 2023. gob. https://pkg.go.dev/encoding/gob
[20] gRPC Authors. 2023. https://grpc.io/
[21] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems*, Eric Bodden, Mathias Payer, and Elias Athanasopoulos (Eds.). Springer International Publishing, Cham, 161–176.
[22] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy* (Tel-Aviv, Israel) *(HASP '13)*. Association for Computing Machinery, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/2487726.2488370
[23] Intel. 2023. Intel Software Guard Extensions for Linux* OS. https://github.com/intel/linux-sgx
[24] Darya Kaviani, Sijun Tan, and Raluca Ada Popa. 2023. *Don't Trust Me: A Platform for Deploying On-Demand Distributed Trust*. Technical Report. University of California, Berkeley.
[25] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872
[26] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 830–842. https://doi.org/10.1145/2976749.2978357
[27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2020. Spectre Attacks: Exploiting Speculative Execution. *Commun. ACM* 63, 7 (jun 2020), 93–101. https://doi.org/10.1145/3399742
[28] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the Isolated Kernel Address Space using Tagged TLBs. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Virtual Event, Europe, 309–321. https://doi.org/10.1109/EuroSP48549.2020.00027
[29] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. https://doi.org/10.17487/RFC6962
[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp
[31] Moxie Marlinspike and Trevor Perrin. 2016. *The Double Ratchet Algorithm*. Technical Report. Signal.
[32] Moxie Marlinspike and Trevor Perrin. 2016. *The X3DH Key Agreement Protocol*. Technical Report. Signal.
[33] Moxie Marlinspike and Trevor Perrin. 2017. *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. Technical Report. Signal.
[34] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system*. Technical Report. Bitcoin Project. https://bitcoin.org/bitcoin.pdf
[35] National Institute of Standards and Technology. 2001. *Advanced Encryption Standard (AES)*. Technical Report. National Institute of Standards and Technology. https://doi.org/10.6028/NIST.FIPS.197-upd1
[36] Trevor Perrin. 2016. *The XEdDSA and VXEdDSA Signature Schemes*. Technical Report. Signal.
[37] Eric Rescorla. 2018. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446. https://doi.org/10.17487/RFC8446
[38] Dragos Rotaru, Nigel P. Smart, and Martijn Stam. 2017. Modes of Operation Suitable for Computing on Encrypted Data. *IACR Transactions on Symmetric Cryptology* 2017, 3 (Sep. 2017), 294–324. https://doi.org/10.13154/tosc.v2017.i3.294-324
[39] Peter Saint-Andre and Jeff Hodges. 2011. Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS). RFC 6125. https://doi.org/10.17487/RFC6125
[40] Congressional Research Service. 2021. *SolarWinds Attack—No Easy Fix*. Technical Report. Congressional Research Service.
[41] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (nov 1979), 612–613. https://doi.org/10.1145/359168.359176
[42] StrongLoop, IBM, et al. 2017. Express. https://expressjs.com/
[43] Gin Team. 2022. Gin Web Framework. https://gin-gonic.com/
[44] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. 2022. Design and Evaluation of IPFS: A Storage Layer for the Decentralized Web. In *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 739–752. https://doi.org/10.1145/3544216.3544232
[45] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2015. A Placement Vulnerability Study in Multi-Tenant Public Clouds. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 913–928.
[46] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 679–697. https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen
[47] ZenGo X. 2023. https://github.com/ZenGo-X/multi-party-ecdsa
[48] Andrew Chi-Chih Yao. 1986. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE, Toronto, ON, Canada, 162–167. https://doi.org/10.1109/SFCS.1986.25
[49] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-Tenant Side-Channel Attacks in PaaS Clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (Scottsdale, Arizona, USA) *(CCS '14)*. Association for Computing Machinery, New York, NY, USA, 990–1003. https://doi.org/10.1145/2660267.2660356