5-2018

Cloud Anchor: An Exploration of Service Integrity Attestation with Hardware Roots of Trust

Christopher Alexander Craig
University of Tennessee

To the Graduate Council:

I am submitting herewith a thesis written by Christopher Alexander Craig entitled "Cloud Anchor: An Exploration of Service Integrity Attestation with Hardware Roots of Trust." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Maxfield J. Schuchard, Major Professor

We have read this thesis and recommend its acceptance:

Joseph B. Lyles, Stacy J. Prowell, Jinyuan Sun

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Cloud Anchor: An Exploration of Service Integrity Attestation with Hardware Roots of Trust

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Christopher Alexander Antone Craig

May 2018

To my wife and dying ambition...

Acknowledgments

I would like to thank my wife, my committee, and the musical stylings of Hail Mary Mallon for getting me through this. In addition, I want to thank praise the wisdom of my advisor, Max Schuchard and the entirety of the VolSec student group for giving me the tough love my work so desperately needed.

Abstract

Distributed computing has enabled developers and researchers to solve complex problems at an impressive scale. Users implicitly trust these subtasks to be performed accurately and this trust can be abused by malicious service providers who aim to compromise the integrity of the system. These problems can be solved by using dedicated hardware; however it is expensive or impossible to distribute this solution to all providers in a system. In this paper, we explore InTest, a service integrity attestation framework that uses replay-based consistency checks to detect malicious service providers without the use of dedicated hardware. We investigate if its performance is affected by network topology, its accuracy in the face of incomplete information, and if it can be improved by minimally utilizing dedicated hardware. Our preliminary solution, Cloud Anchor, reduces the number of duplicated tasks by 30% while providing identical detection rates as the prior solution.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed systems can be a wonderful tool for solving widespread and difficult tasks by allowing a user to effectively scale their workload, storage, and communication to the size of their problem. This scalability makes it difficult to manage the integrity of this shared service without proper administrative tools in place. Orchestration and configuration management can be used to produce a resilient infrastructure but for environments that are sensitive to interruptions, downtime can be costly. Remote attestation offers a way for administrative systems to question a service provider and verifiers to submit proof of their integrity. Existing remote attestation implementations [Shi et al., Eldefrawy et al., Seshadri et al.] can effectively provide service integrity for their distributed systems, but either require specialized hardware or kernel support. These requirements may be difficult or impossible to deploy in a large-scale cloud or grid computing environments.

In this paper, we explore a software-only service integrity framework named IntTest which uses an integrated graph analysis scheme to pinpoint malicious service providers. First, we ask if IntTest is an efficient service integrity solution and show that it can accurately pinpoint malicious providers with only a few edges in its graph-based solution. We then show how its performance may vary given different service configurations. We demonstrate how it performs suffers under realistic adversarial models. Finally, we reveal how IntTest can be enhanced with minimal use of dedicated hardware through our solution: Cloud Anchor.

The rest of this paper is laid out as follows. Chapter 2 will provide relevant background on cloud computing, remote attestation, and graph theory concepts used in the IntTest

solution. Chapter 3 will present the fundamental prior work to the IntTest solution, the methodology behind the framework, and the intuition behind our simulation. In Chapter 4 we will explore the solution through our simulation and demonstrate its performance under a variety of scenarios. Lastly, Chapter 5 reveals our improvement to the system Cloud Anchor and preliminary results on its efficiency and effectiveness in the aforementioned environments. Finally, Chapter 6 will present future work and our conclusions.

Chapter 2

Background

2.1 Distributed Systems and Cloud Environments

Distributed systems are modern computing architectures that have changed the way we consume and process information. While simply defined as a system of computers attached via a network, their utility can be expressed in a variety of Software-Oriented Architectures.

Service-Oriented Architectures are a novel design that offers attributes of the distributed environment to the end user. Infrastructure-as-a-Service allows users to use the physical infrastructure of the distributed system as if it were bare metal to control networking, memory, and core cpu usage. This is accomplished via abstractions such as virtual machines and cloud computing software. Platform-as-a-Service offers a base-of-operations for developers to run and execute custom software on their machines. These programs do not interact with the bare metal directly but are commonly used as development environments or deployment platforms. Software-as-a-Service offers the user an application that is deployed across the distributed network for scalability and availability.[Alonso et al.]

The proliferation of Service Oriented Architectures has lead to an increase in distributed security concerns as each platform exposes portions of the network to malicious applications and users who aim to break through those abstractions.

2.2 Roots of Trust

Trust anchors are a security primitive in which the item's security is assumed and not derived. In certificate chains, X509 root certificates provide the starting link in a chain of trust. In distributed systems, they offer a guaranteed starting point for many trusted computing environments or a location for a centralized trusted device. Hardware roots of trust are essential in developing a robust, secure computing environment.

2.2.1 Trusted Platform Module

A Trusted Platform Module is an international standard for a secure coprocessor that performs various tasks related to cryptographic keys to secure the host machine. The main specification was designed by the Trusted Computing Group and provides a suite of hardware components to enable a variety of features.

It is capable of providing a secure pseudorandom number generator for cryptographic primitives and key generation and management. These features enable it to perform attestation for protected services and trusted boot.[Morris]

2.2.2 SGX

Intel Software Guard Extensions are a set of extensions to the Intel architecture that can guarantee confidentiality and integrity for application in spite of the operating system or any higher privileged application on the device.[Costan and Devadas]

2.3 Byzantine Fault Tolerance

For a shared or distributed computing environment, a Byzantine fault is any fault or disagreement when a result is presented to different observers. These can lead to in Byzantine failures which is when a system loses service due to a Byzantine fault in any system that require consensus.[Driscoll et al.]

The name itself comes from a colorful description of Byzantine generals at war who must come to a consensus on when to sack an enemy city. The generals and their lieutenants

are mounted on different sides of the city and must communicate with messengers. The challenge is to come to this consensus and guarantee the messages have not been modified or misreported by a traitors messenger, lieutenant, or general. In spite of the anthropomorphic representation of the problem, it appears frequently in distributed systems that must have consistent information.[Lamport et al.]

If the functioning components of a Byzantine fault tolerant system are operating correctly, they will continue to provide the system's service with improved resiliency.

2.4 Graph Theory

The implementations extended in this paper depend on certain fundamental concepts in the domain of graph theory: cliques and minimum vertex covers.

Cliques are a subset of vertices of an undirected graph such that every two distinct vertices in the clique are connected by and edge or adjacent. Another way to view this concept is that the subgraph formed by the edges and vertices of the clique is complete. A maximal clique is a clique can not be increased by adding a single adjacent vertex in the graph, meaning it is the largest complete subgraph of a graph. Finding the maximal clique of a graph can be accomplished efficiently via the Bron-Kerbosh algorithm.

Bron-Kerbosh is an efficient algorithm discovered by Dutch computer scientists Coenraad Bron and Joep Kerbosch in 1973. It's a recursive backtracking algorithm that, given 3 sets, will discover all maximal cliques of a graph in the set.[Bron and Kerbosch]

Another important concept of graph theory to understand for this paper are vertex covers and specifically the minimum vertex cover of a graph. A vertex cover is defined as a subset of vertices in a graph such that all edges of the subset are incident to at least one vertex in the larger graph. More simply, it's a set S of vertices of G such that every edge of G has at least one of member of S as an endpoint. A minimal vertex cover is the smallest set of vertices whose edges cover the entire graph. While the concept seems simple, finding the minimum vertex cover of a graph is a classically NP-Complete problem. The approaches used in the paper can approximate the minimum vertex cover in a timely manner due to the size of the graphs produced.[Karakostas]

2.5 Remote Attestation

Remote Attestation is a concept that allows a computer's integrity to be evaluated by a remote party. This is commonly performed to check for unknown malware present in the memory (persistent or volatile) of a system. Its utility can be extended to detect insufficient configurations, objectionable system properties, or benign software that may not comply with the party's license agreements.[Lpez and Zhou]

The concept of remote attestation the relationship between two machines: an attestor and a verifier. Similar to its legal definition, the attestor is one who attests to the integrity of the system by providing their signature or checksum. The verifier is the individual or system that is initializing the challenge and verifying the received checksum. Common implementations require computing a checksum or token that will either produce an incorrect result or cause a measurable delay if modified.[Francillon et al.]

Sharing a symmetric key, the verifier submits a challenge to the attestor with a nonce and expects the result in a limited amount of time to constrain any efforts from malicious attacker to forge a legitimate response. The attestor must prove they have a legitimate section of software loaded into memory. They combine the requested value with the nonce, sign the response, and return it to the verifier. The verifier checks the signature and the response from the attestor.[Shi et al., Eldefrawy et al., Seshadri et al.]

Remote attestation frameworks can be used in a variety of environments but are simply demonstrated in a distributed network. Distributed networks can be meticulously managed to ensure each node is performing as intended with an unmodified firmware with untampered input however this does not scale with the growth of these networks. To ease the overhead required to manage distributed nodes, hardware trust anchors can used to manage keys, attest sections of memory. Most remote attestation frameworks that use trust anchors assume they are applied on all critical nodes of a network and concern themselves with efficient management of the attestation metadata. While hardware anchors can successfully check the integrity of nodes in a distributed network, there are many cases in which a hardware anchor is unavailable for a node, or the critical service is provided by a third party. In these

cases, software-only approaches are novel ways to accomplish network integrity without additional hardware.[Lpez and Zhou]

Software-only remote attestation schemes have been developed to attest memory regions, process integrity, and even controlled execution. SCUBA[Eldefrawy et al.], a software-only approach based on it's predecessor Pioneer[Seshadri et al.], is able to attest to memory regions of the remote system by ensuring the checksum is calculated atomically. Another method, BIND[Shi et al.] is able to attest to the memory region of an application nd its output to verify not only the applications integrity but it's output. These functions can be extended to run processes in a controlled manner by passing the controlled execution environment to the critical application. The benefit of these approaches are the exclusion of dedicated hardware but recent improvements to hardware trust anchors can guarantee these properties more efficiently and effectively.

Most implementations required trusted hardware to be ubiquitous throughout the environment Others required a third-party to confirm memory regions or signatures. Initial implementations of this are can be improved with a costly oracle, but not a hybrid solution.

Chapter 3

The IntTest Simulator

3.1 Preliminary Work

As mentioned in the prior section, cloud computing is an effective means of solving complex problems with distributed, parallel solutions. In particular, data-stream processing services appear to be an effective use of the scalability and distribution cloud computing offers; but when many tasks are distributed to disparate service providers, the ability to fully control those services becomes increasingly difficult. To combat this problem, RunTest and its similar frameworks were devised to guarantee the service remained intact as its modular components were distributed to other service providers.

RunTest[10] is a lightweight attestation scheme that determines dataflow integrity by graphing and analyzing consistency relationships between service providers without additional hardware. RunTest expected If two service providers are functionally equivalent and produce different results from the same input, one can be considered malicious. These replay-based consistency checks are used to effect the edges of a consistency graph for each function in processing pipeline.

Consider the following graph in Figure 3.1. For the given data processing pipeline, there are three service functions f, g, h that are processing data in stages by providers $1 - 9$. As data flows through the pipeline, the RunTest framework will record a percentage of the full data-paths at random. It then sends the same input values along a different service path. If two service providers were given the same inputs and the same outputs were produced, the

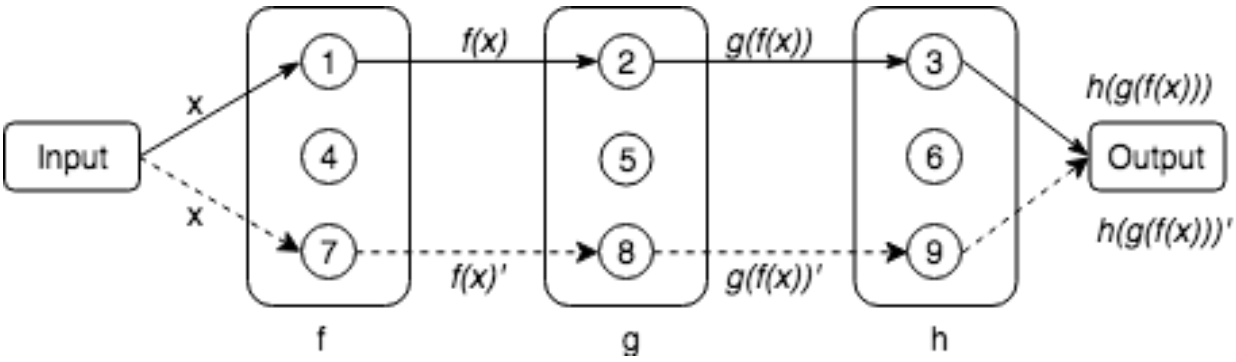


Figure 3.1: Example of a replay-based consistency check

service integrity is preserved and these providers are considered bengin. If the values differ, it can be assumed that one of the given outputs is invalid and its creator is malicious.

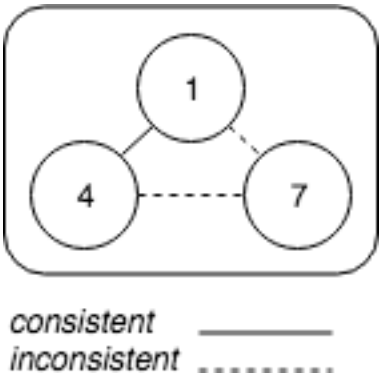


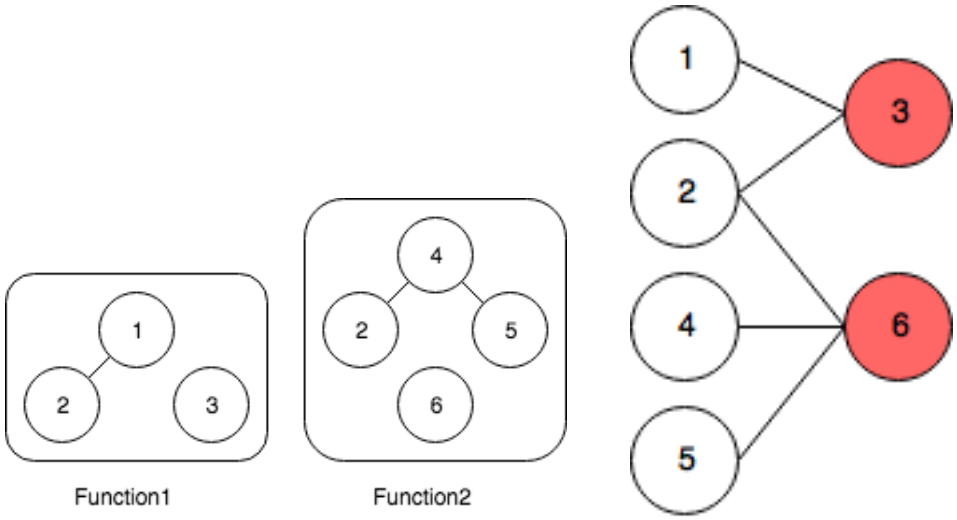
Figure 3.2: Complete consistency graph for function f

RunTest builds a weighted graph by comparing the intermediate and final results of each functionally equivalent service provider with each other. This can pinpoint which provider in the processing pipeline is modifying the result. The inital approach incurrs a high overhead since the duplications require a full run through the processing pipeline to produce redundant results. The authors of their respective papers developed IntTest to alleviate the attestation overhead and reduce false positives.

3.2 IntTest

IntTest[8] is an attestation scheme that builds upon its predecessor RunTest[10] by abandoning weighted graphs in favor of simple per-function consistency graphs and a global

inconsistency graph. Relationships between service providers are still determined using replay-based consistency checks which process verified input values across different service providers and compare their results. Instead of subtracting values from a weighted edge, an edge is placed in the function's consistency graph if they agree. Alternatively, an edge is placed in global inconsistency graph if these values disagree.



(a) Consistency Graphs **(b) Inconsistency Graph**

Figure 3.3: Attestation Graphs

After all input has been processed and our graphs have been formed, IntTest iterates through each graph to pinpoint malicious service providers. Following the pseudocode in Figure 1

The algorithm is divided into two halves that first analyze the global inconsistency graph followed by each consistency graph. The inconsistency graph is connecting edges between two disjoint sets of consistent and inconsistent nodes. This bipartite graph has a minimum vertex cover containing the most inconsistent service providers. The pinpoint algorithm uses this as a lower estimate for the number of malicious service providers we assume to be in the system. Our upper-bound is constrained to $\lfloor n/2 \rfloor$. The algorithm iterates over each provider in the graph and calculates the sum of its neighborhood and remaining edges in the node's residual graph. If the sum is above our estimate K, the provider will be pinpointed as a suspicious node and added to the set Ω.

Algorithm 1 Pinpiont Malicious SPs (G,G_i)

```
function PinpointMaliciousSPs(G,G_i)
    for every K ∈ [|C_G|, ⌊N/2⌋] do
        Ω = ∅, R = ∅
        for every node p in G do
            compute |N_p| + |C_{G'_p}| > K
            if |N_p| + |C_{G'_p}| > K then
                Ω = Ω ∪ {p}
            end if
        end for
        final malicious node set R = R ∪ Ω
        if R = ∅ then
            continue
        else
            for every G_i do
                compute M_i
                set Ω_i to the subset of Ω appearing in G_i
                if Ω_i ∩ M_i ≠ ∅ then
                    R = R ∪ M_i
                end if
            end for
        end if
    end for
end function
```

The second half iterates through each consistency graph searching for the maximal clique, which represents the largest group of consistent nodes for that function. If any values are missing from that group, they are added to the malicious set M_i for further analysis.

In the final stage we search for providers present in both M_i and Ω and add their groups to the final set of malicious nodes R.

3.3 IntTest Simulator

The simulator has been developed in Python to fully replicate an implementation of the IntTest[8] attestation framework. Unlike its predecessor RunTest[10], IntTest investigates service integrity with two separate graphs that represent consistency and inconsistency relationships between service providers. These results are collected and processed in turn to produce a list of potentially malicious nodes.

The original was developed to operate in the NCSU virtual computing laboratory[vcl] which is similar to Amazon's Elastic Cloud service[ama]. The original authors deployed an instance of IBM System S[str], a high-performance stream processor, and manage the input and output values for all participating service providers. The simulator abstracts the task processing dataflow into a simple pipeline to investigate the overload incurred by duplicating results without the additional overhead of the underlying system.

3.4 Simulator Data Structures

To abstract the nuanced operations of the System S the simulator uses a combination of simpler data structures known as Tasks, Functions, and Nodes. Tasks are defined as a series of functions that iterate through the pipeline processing functions $f1, f2, \ldots, fn$ for n functional stages. A task does not have to strictly follow an iterative pipeline in the simulation but we follow the example set in the IntTest implementation.

A Function represents an independent piece of the dataflow pipeline which is simply implemented to pass along boolean values to the next stage in the pipeline. As with the source material, our functions must be input-deterministic.

Nodes are simple data objects that can be labeled *malicious* or *benign* at the start of the simulation. If a nodes input and return values remain *True*, the node is considered benign and will report an unmodified output state. A malicious node will always return *False* and fail our distance measures. A Node may support multiple functions, but we choose to explore the effectiveness of the attestation scheme may benefit from various levels of functional crossover in Chapter 4.

3.4.1 Task Processing

When a task is processed it undergoes the process described in Section 3.2 and each full run of the pipeline is compared to a functionally equivalent duplicate. For IntTest to make a valid comparison the inputs must be equivalent for two nodes to be fairly compared. This is true for intermediate results as well. As a task is processed, the input for each service provider is recorded. Once the task is finished processing, its return values are compared and edges

are formed in their respective attestation graphs. This is sufficient for most data processing; but a benign node may perform the function correctly on tainted input data. Keeping this in mind, the Node's output value only report on the individual node's performance on the given input data, to more accurately reflect a proper evaluation function for each pipeline stage.

Chapter 4

Exploring the IntTest Framework

IntTest was designed for Service-Oriented architectures that process data through known functionally equivalent service providers in a series. The framework can compare intermediate values in an efficient manner to devise suspicious service providers and bypass their influence on the distributed system. In our investigation, we ask if IntTest excessively duplicates tasks to fill its attestation graphs. We inquire how the performance of the framework changes in the face of different service configurations and a realistic attack model. Lastly, we wonder how the existing implementation can be improved if given a costly *oracle* to derive the integrity of a provider.

4.1 Efficiency

The IntTest framework operates under the expectation that all pipeline functions will be thoroughly attested and their function graphs should be as dense as possible. They come to rhis step by choosing a percentage of input values to be used for random replay-based consistency checks. To avoid detection they must randomize the input values and their paths to ensure the attacker is unable to predict when they may be attested and avoid detection. This implies there are many redundant consistency checks in which a pair of providers has already been checked. We inquire whether a complete function graph and inconsistency graph are necessary for a complete evaluation or can the effort of searching for the last few unattested pipeline paths?

We tested this with a pipeline of three functions with ten unique nodes to service each function. 20% of the total pool of providers are classified as malicious and we perform our service checks over 1000 unique data inputs to iterate over the duplication percentage. The malicious nodes were consistent as duplications increased and the results were the average of ten runs. This will reflect how many consistency checks are required to attain an accurate classification.

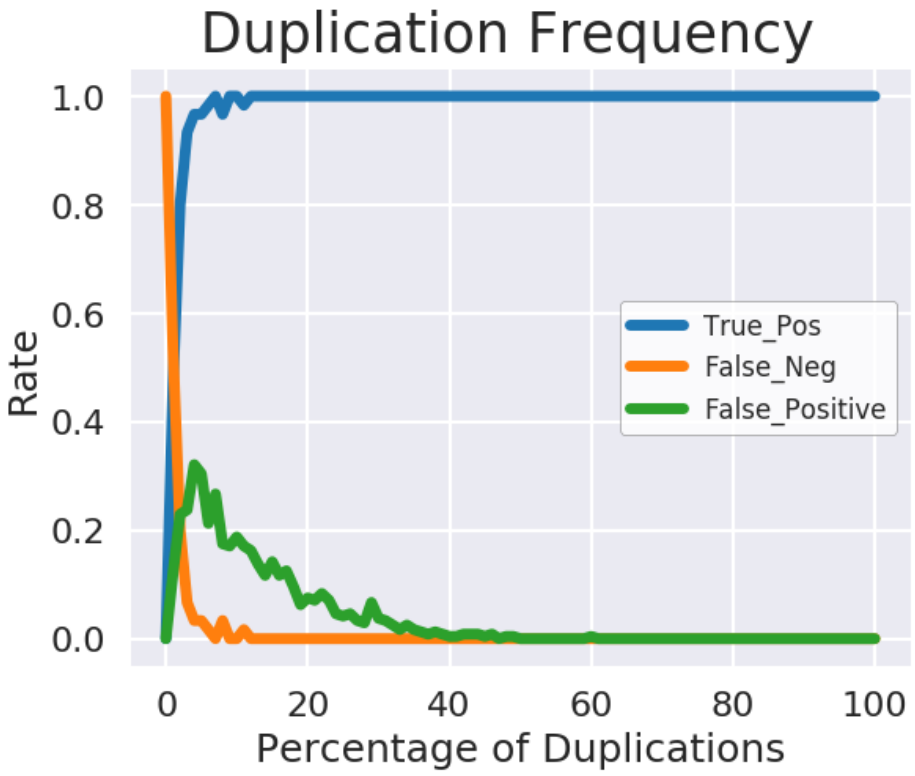


Figure 4.1: 3 service functions, 10 providers each, 20% malicious nodes

Table 4.1: Function Densities for a Tall Pipeline

Duplication Percentage	**Function f_1**	**Function f_2**	**Function f_3**
20%	0.98	0.82	0.89
30%	0.98	0.96	0.93
40%	1.0	1.0	1.0
50%	1.0	1.0	1.0

Figure 4.1 shows that IntTest reaches near 100% true positive rates within the first few checks. Following that, false positives are less than 10% within 200 duplications. Table 4.1 shows the function densities are not fully formed at 20% duplications yet we can easily arrive at an accurate classification of malicious nodes. IntTest dictates that we would have to perform another 200 duplications until the conditions are satisfied for analysis.

4.2 Performance

We can conclude that there is an unnecessary amount of redundancy needed to form dense attestation graphs, but is this true for all service pipeline configurations? To explore this concept we will first evaluate function pipelines with multi-purpose nodes and then pipelines that change the ratio of providers to service functions.

4.2.1 Function Crossover Scenarios

Service providers do not have to be dedicated to a single service and can support a variety of functions. This crossover can be extremely helpful in the aforementioned service-integrity frameworks. Consistency graphs contain attestation information on one function and its service providers; but a malicious provider can be more easily discovered if they tamper multiple functions and appear in multiple graphs. In our evaluation, we divided the multifunctionality of a service provider into three configurations to accommodate different distributed system environments: *None*, *Partial*, and *Full*.

No Crossover

This configuration considers that each service provider only support a single function. This would be describe a dedicated distributed system that may lacks redundancy but each provider is highly specialized to handle the task. Figure 4.1 reflects this configuration state and will be our baseline for performance comparisons. We can see from Table 4.1 that it can form three complete function graphs at 400 duplications and accurately classify benign and malicious nodes.

Partial

Partial crossover considers the jobs are evenly distributed across service providers and every node supports at least two service functions. This scenario represents a highly-available distributed system, in which certain functions are widely distributed across dedicated nodes. These nodes may perform other tasks but are not solely dedicated to serving a single purpose.

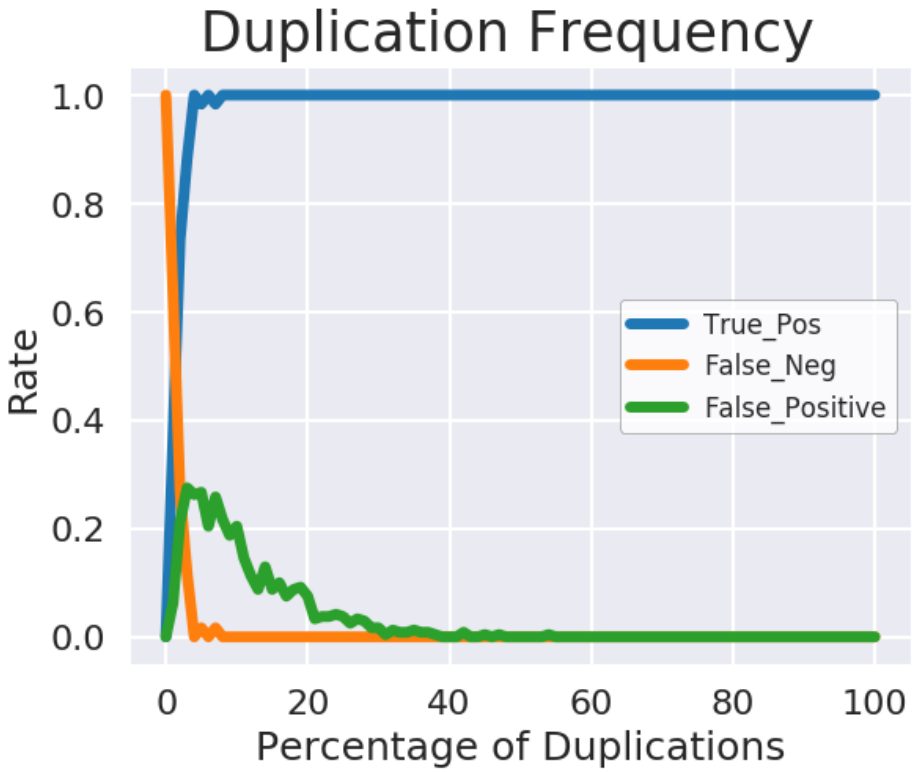


Figure 4.2: 3 service functions, 10 providers each, 20% malicious nodes, partial crossover

As Figure 4.2 shows, there is a slight, but improved performance. The intuition is that these malicious providers should be present in multiple consistency graphs. Since a malicious node and a benign node will always disagree, fewer consistency checks are required to discern the presence of a malicious provider.

Full

The *Full* crossover configuration represents an environment in which all service providers are capable of handling every function. This is similar to a job pool scenario where there are no dedicated providers and all tasks are simply distributed to whichever provider is available.

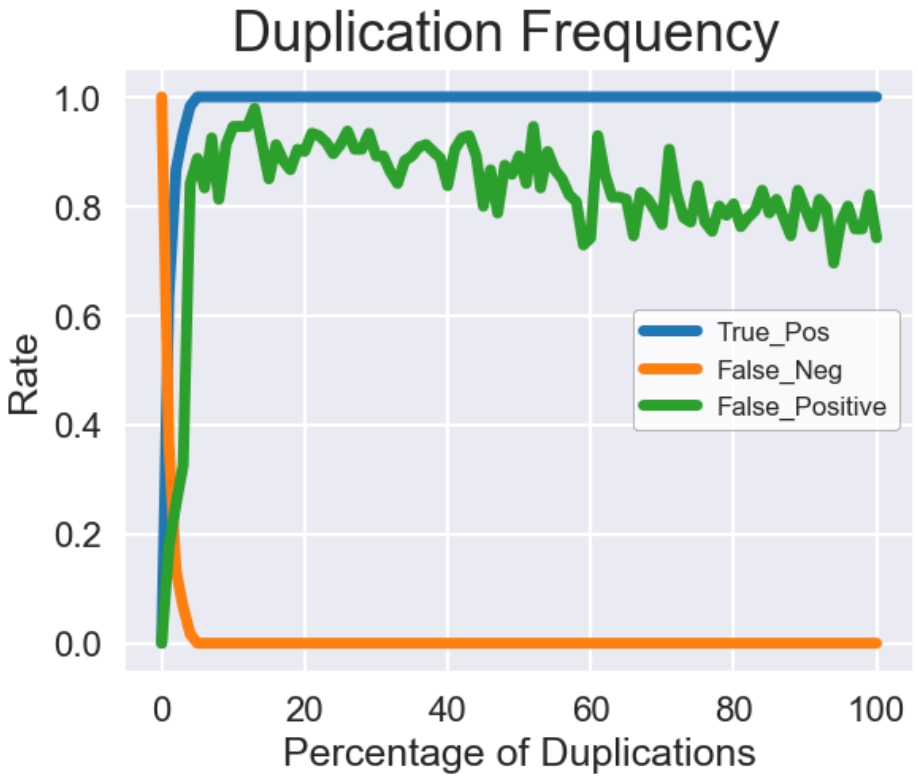
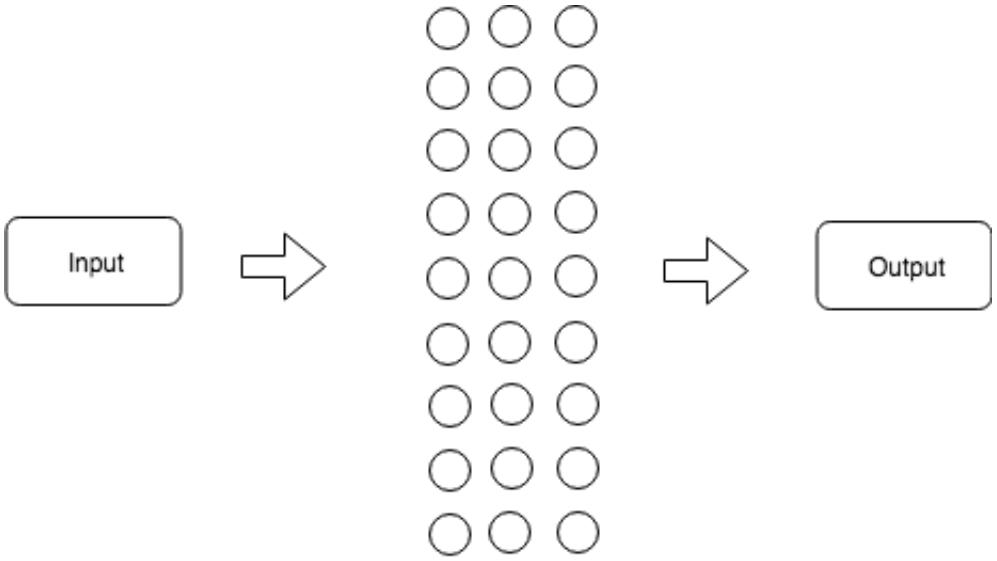


Figure 4.3: 3 service functions, 10 providers each, 20% malicious nodes, partial crossover

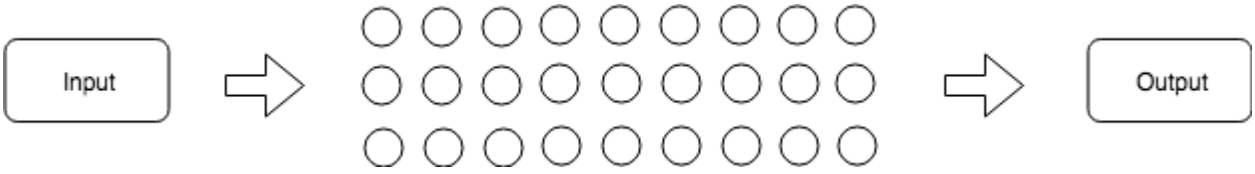
Interestingly, the evidence shows that IntTest has a difficult time handling this scenario. It can easily detect malicious nodes but can not discern malicious from benign and begins to classify the vast majority of nodes as malicious. Function densities show at 1000 duplications show that f_1 has only 88% of the edges in its function graph, f_2 has 79% density, and f_3 has only 68% density. Meaning the maximal clique for each function graph is not fully-formed and performance can ultimately be hindered by multifunctionality.

4.2.2 Pipeline Configuration

In addition to service crossover, the number of service functions supported by a single data-processing pipeline may not be small. For our evaluation we consider two types of pipeline configurations depending on the ratio of service providers to service functions.



(a) Tall Service Pipeline



(b) Wide Service Pipeline

Figure 4.4: Example of a Tall and Wide 27 Node Pipeline

Tall Service Pipelines

Pipeline configurations in which there are fewer service functions than there are service providers are considered to be *Tall*. The testing parameters above displayed a simple pipeline of three service functions with ten providers for each function. Figure 4.1, Figure 4.2, and Figure 4.3 display the accuracy and performance of the IntTest framework under the three aforementioned crossover configurations.

Wide Service Pipelines

Wide service pipeline configurations have many service functions but only a small relative number of service functions to support each function. This can be seen in Figure 4.4a, in

which input must pass through nine service functions and only three service providers can support each effort.

To explore *Wide* pipeline configurations, our pipeline consisted of ten service functions and three providers for each function.

When every provider for each function is distinct, we notice a considerable performance impact. Figure 4.5 reveals true positive rate at 34% while false positives remain low. If there is partial crossover, performance is not as great as its tall counterpart. Figure 4.6 shows that having multi-function providers makes malicious nodes easier to pinpoint, however false positives are difficult to remove without additional duplications. If in a job pool scenario with full crossover, Figure 4.7 shows we are completely unable to discern a malicious node from a benign one. The function graph density for this graph can be found in Table A1.

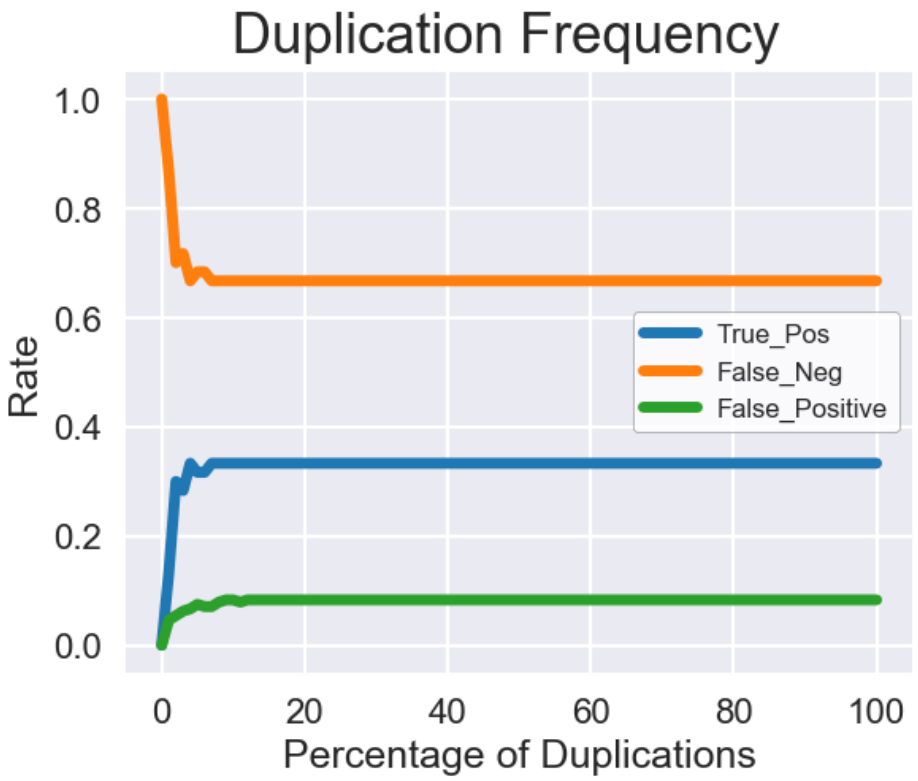


Figure 4.5: 10 service functions, 3 providers each, 20% malicious nodes, no crossover

If we take a moment to consider why the wide configuration caused such poor performance, there is a key difference in the way IntTest analyzes pipelines. The framework can only make a valid comparison between two providers if the input is identical. For the

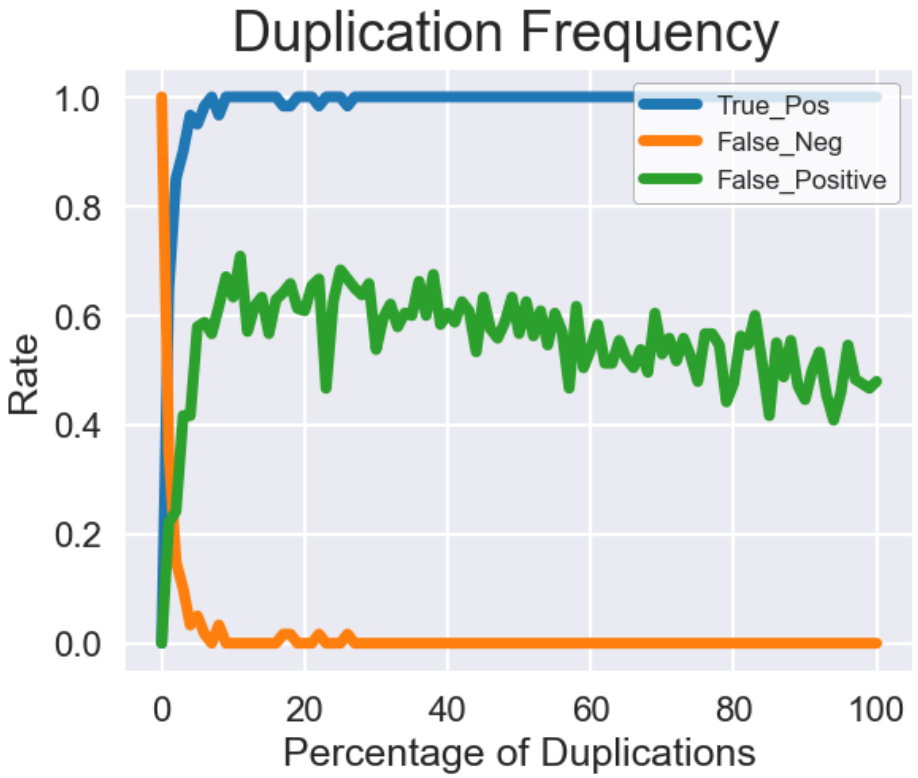


Figure 4.6: 10 service functions, 3 providers each, 20% malicious nodes, crossover

first tall pipeline, one must only select three providers from a specific test pool. For our wide pipeline, we must select ten providers and our odds are higher of selecting a malicious node that would effect our given input path. Figure 4.8 shows the geometric distribution of each service function pipeline with %20 of our providers being malicious. The cumulative distribution function of this distribution shows that we are much more likely to encounter a malicious node in our service path if we are given a wide pipeline as opposed to a tall one.

If IntTest is unable to make valid comparisons, it will have a difficult time classifying malicious providers. Figure 4.9 shows how this effects the per-function consistency graphs for a wide pipeline. Graphs for the first three functions are able to become complete graphs fairly quickly as more duplications are added to its analysis. The next three functions come closer to complete graphs but they require significantly more duplications. Functions f_4,f_5, and f_6 struggle at 600 duplications and function graphs for f_7 and f_8 are appear to not converge after 1000 duplications.

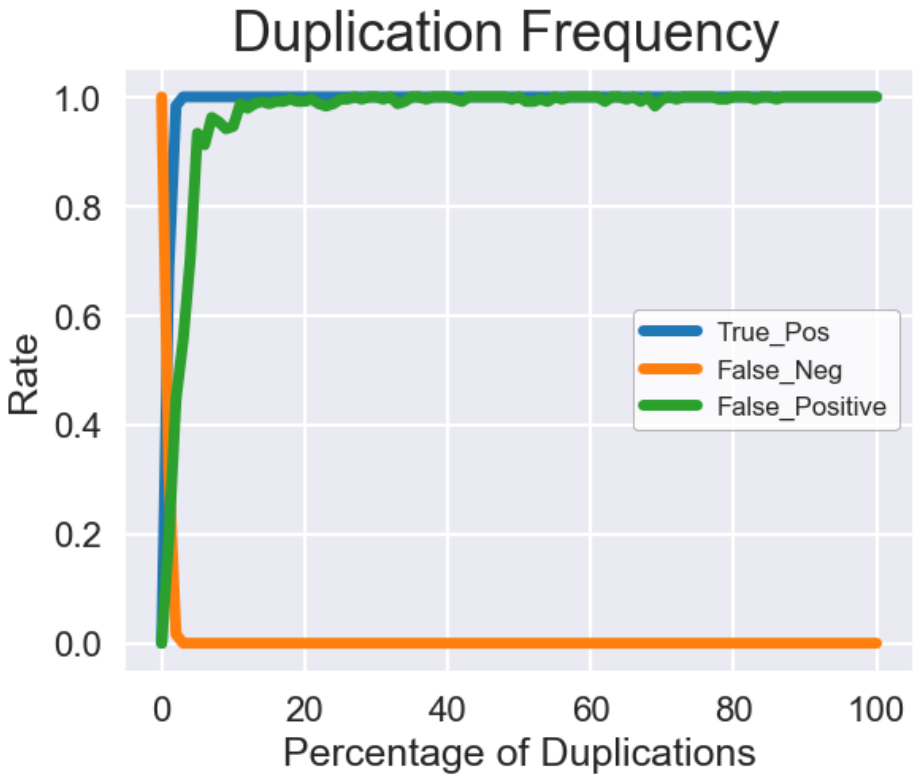


Figure 4.7: 10 service functions, 3 providers each, 20% malicious nodes, full crossover

It appears much of IntTest's performance can be dictated by the pipeline's configuration and we are unable to improve its detection capabilities with additional duplications. In particular, the information available for analysis becomes more sparse as the pipeline adds more service functions. While a complete graph is not required for correct classification, it is easier to have false positives if no comparisons are made between benign nodes. This analysis was performed under the assumption that malicious providers will always report an incorrect value when attested, however what where to happen if the attack model were more realistic?

4.3 Realistic Attack Model

The original IntTest attack model only considered the scenario in which the malicious provider compromise the integrity of the service function every time it is asked. RunTest[10], its predecessor, used weighted graphs to derive the inconsistency of a provider. This is not

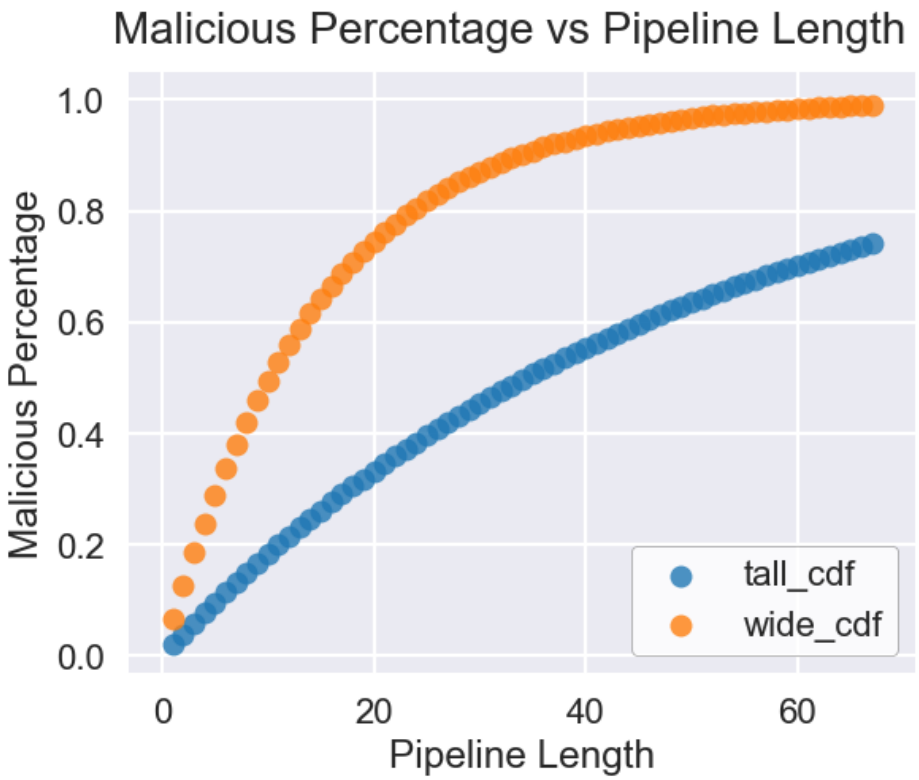
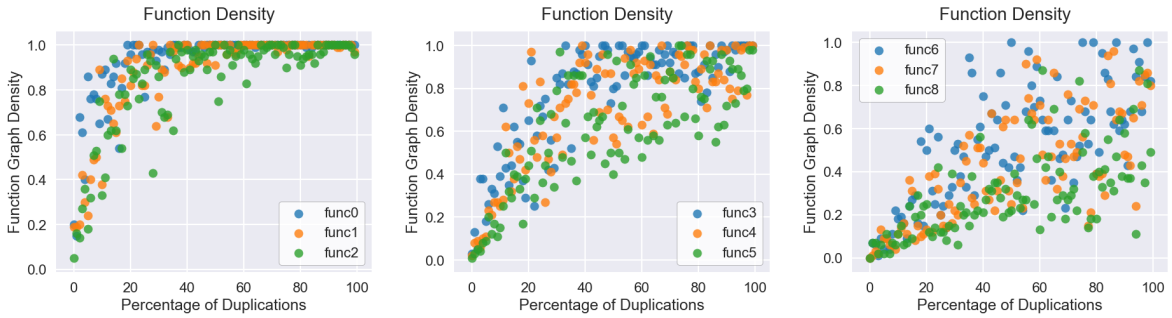


Figure 4.8: Geometric Distribution of the wide and tall pipeline configurations

the case for this service framework, which will permanently classify a node malicious on its first offense. While a reasonable scenario, it is not realistic as a malicious provider may lay in wait and risk detection compromising a particular input value. To analyze this scenario, we enable the malicious provider to *lie* and report as a benign node with a probability of 50%.

As our previous section demonstrated, its performance is largely dependent on the pipeline configuration. We suspect that tall pipelines will be fairly resistant to this level of tampering. As stated in Section 4.2.2 there many service paths chosen for service integrity checks and each have a relatively small path of three service functions. There is less likelihood that a malicious node will be in the path than a wide pipeline. The Figure 4.10 confirms our suspicions and is able to quickly arrive at the correct number of malicious nodes and benign nodes at 200 duplications; however the averages are noticeably more volatile with malicious nodes changing their answers.



(a) Functions f_1,f_2,f_3 **(b) Functions f_4,f_5,f_6** **(c) Functions f_7,f_8,f_9**

Figure 4.9: Graphing attestation graph densities for a wide pipeline

Performance already suffered under wider pipelines and the same is true when the attack model has changed. Figure 4.11 shows the performance similar performance but with much more volatility. There are certain configurations that appear extremely difficult if near impossible, such as Figure 4.12. It appears that in the face of lying adversaries, IntTest's performance can only worsen. In the following section, we explore how a physical trust-anchor can improve the performance and efficiency of the IntTest framework.

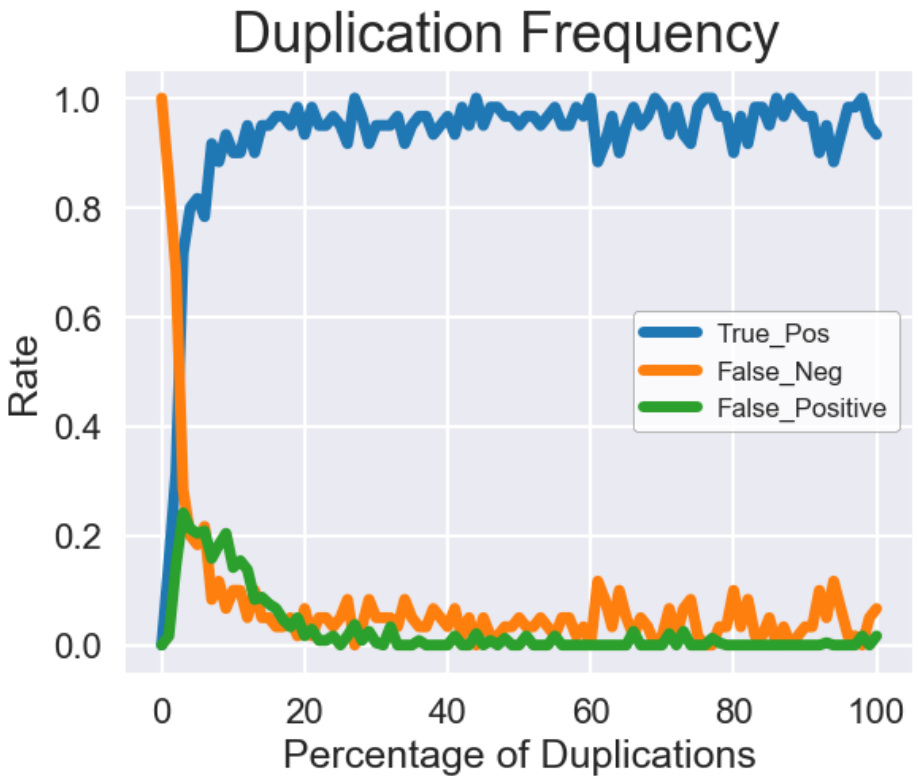


Figure 4.10: Three functions, 1000 data tuples, 20% malicious providers, no crossover

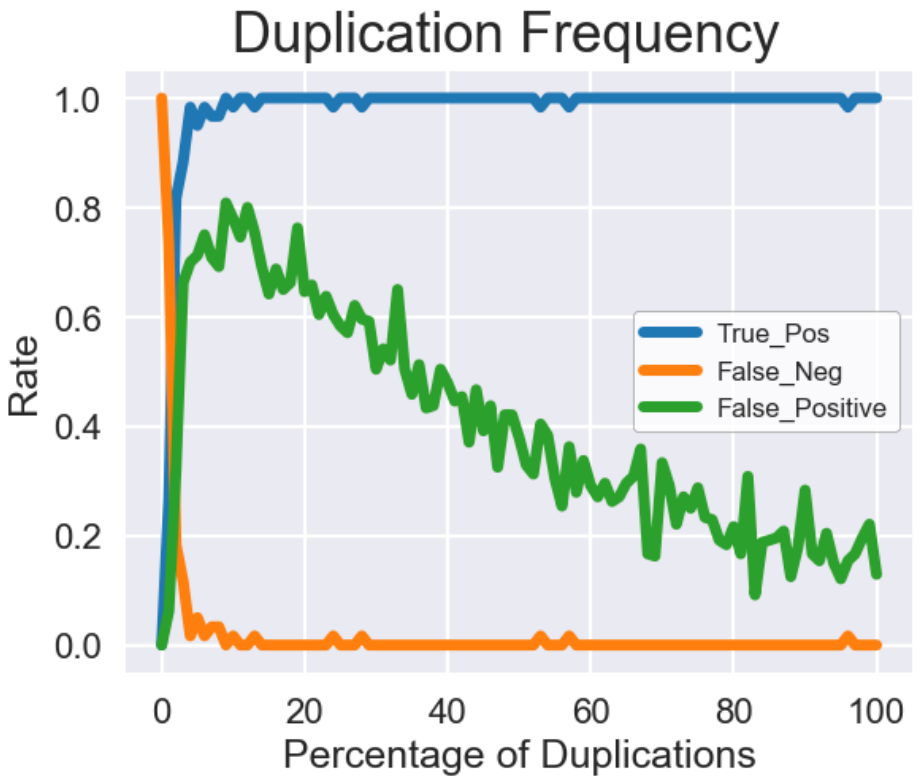


Figure 4.11: Ten functions, 1000 data tuples, 20% malicious providers, partial crossover

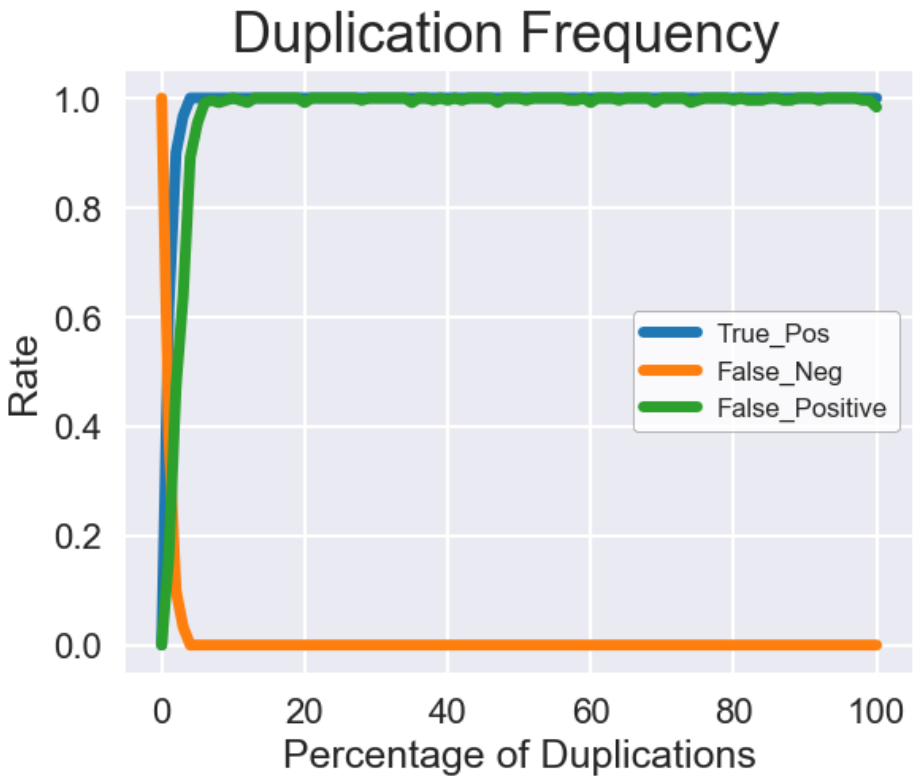


Figure 4.12: Ten functions, 1000 data tuples, 20% malicious providers, full crossover

Chapter 5

Cloud Anchor

Cloud Anchor builds upon the framework laid out by Juan Du, Wei Wei, Xiaohui Gu, and others in the RunTest[10], AdaptTest[9], IntTest[8] attestation schemes.

The Chapter 3 and Chapter 4 enumerated many of the advantages of a software-only service integrity framework as well as its limitations. We investigate the merit of adding trusted hardware to facilitate the role of an *oracle* for this solution. Instead of implementing hardware ubiquitously throughout the distributed network, we can attest to the integrity of a provider. We show that it fundamentally changes the efficiency of the framework and enables it to tackle difficult pipeline configurations.

5.1 Intuition

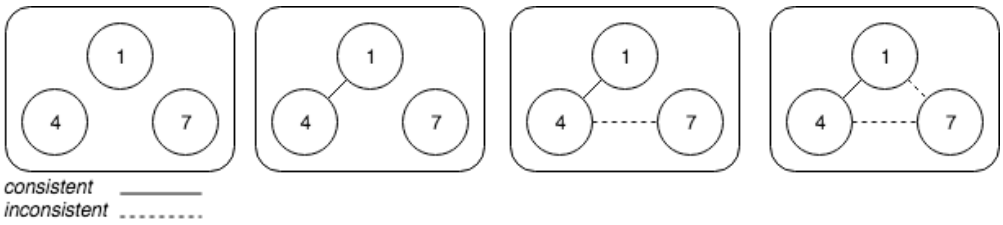
IntTest is a software-only integrity framework, whose primary advantage to remotely attest nodes one can not modify with physical hardware. In IoT distributed networks the service providers are small, embedded devices and power consumption is a priority. For many cloud environments, the service providers may be a third-party who would not permit a new device into their infrastructure. Many solutions such as Intel SGX2.2.2 and TPMs2.2.1 are being used by service providers to offer their services trusted enclaves. Alternatively, we may control one service as a means for comparison and use the aforementioned remote attestation features to produce our own oracle.

Armed with a way to confirm the integrity of a provider we can trust their output, we can use this trust in addition to our consistency and inconsistency relationships to maintain application integrity and improve efficiency.

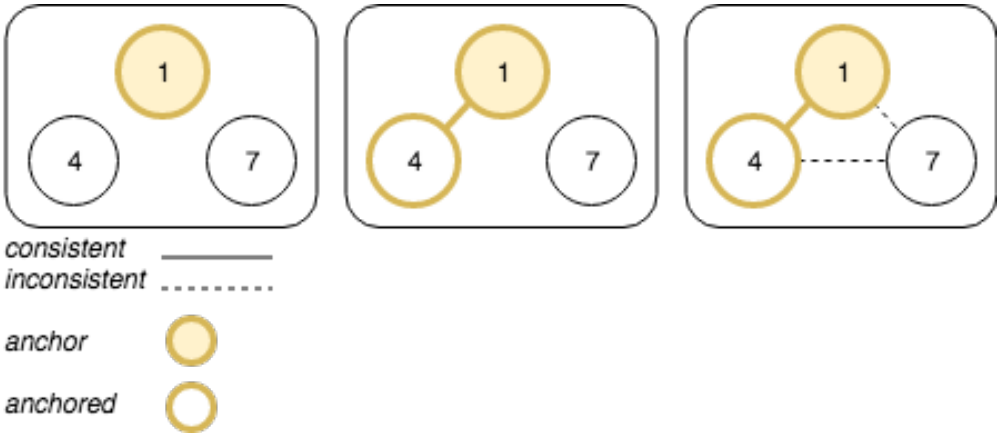
5.2 Cloud Anchor Implementation

Cloud Anchor aims to use the trust of an oracle and distribute that trust throughout the consistency and inconsistency relationships. This is accomplished by selecting an arbitrary node as our *anchor*. We will use the oracle to derive the integrity of this node. Once the node is marked as an anchor, we continue to use the service pipeline identical to the IntTest framework. We'll use replay-based consistency checks to duplicate output for comparison to another path. We enter a special case if these duplications are with an anchor. If a node is being compared to an anchor and the input and output values are identical, the node is considered to be *anchored* and is added to a set of anchored nodes. Since the nodes are input-deterministic, comparisons to an anchored node can be considered a direct comparison to the anchor. Duplications with any member of this anchored set to any member outside the set will resolve connections made to all anchored members.

Consider Figure 5.1. This graph demonstrates the number of duplications required to achieve a complete consistency graph for function f in Figure 3.1. IntTest would have to perform at least three duplications to compare the inputs and outputs between providers 1, 4, 7. In the example above, we discover that 1 and 4 are consistent. In another, 4 and 7 are inconsistent. We know that provider 1 was already demonstrated to be equivalent to provider 4, yet we must run a novel duplication between providers 1 and 7 to fill the consistency graph. For Cloud Anchor, consistency checks are transitive and results are propagated to all anchored nodes. It should be noted that replay-based consistency checks are evaluated per-function, and in our evaluation we select the node with this most multifunctionality to distribute that trust to as many service functions as possible.



(a) Before Cloud Anchor



(b) After Cloud Anchor

Figure 5.1: Example of Cloud Anchor

5.3 Preliminary Results

We evaluated Cloud Anchor under the same guidelines as IntTest. In addition, we we limited our evaluation to have a single anchor which is the provider who supports the most service functions. Finally, we iterated through all the previous pipeline configurations.

We noticed an immediate performance change for tall pipelines. The number of service functions they support are only a few but many duplications are required to increase the density of each service function. Figure 5.2 shows our performance was slightly better but significant improvement came when analyzing full crossover for a tall pipeline in Figure 5.4. IntTest was completely unable to resolve false positives in 1000 duplications yet, Cloud Anchor has correctly classified all malicious within 400 duplications.

The wide piplines appear to have similar performance to IntTest because the benefits of using trust transitivity are minimal when the number of service providers in the anchored

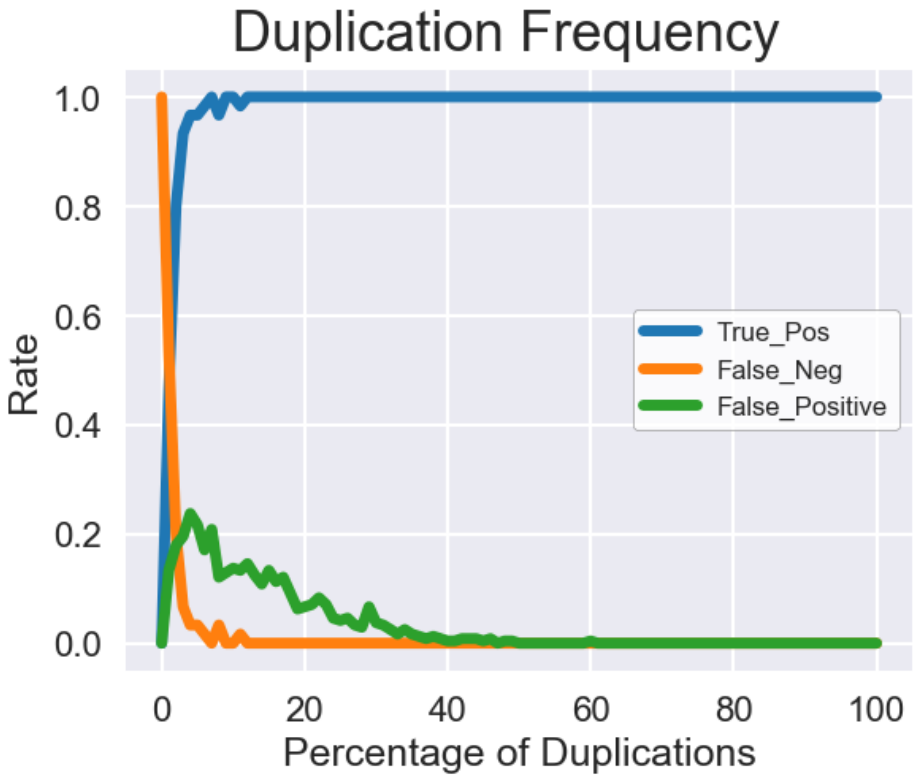
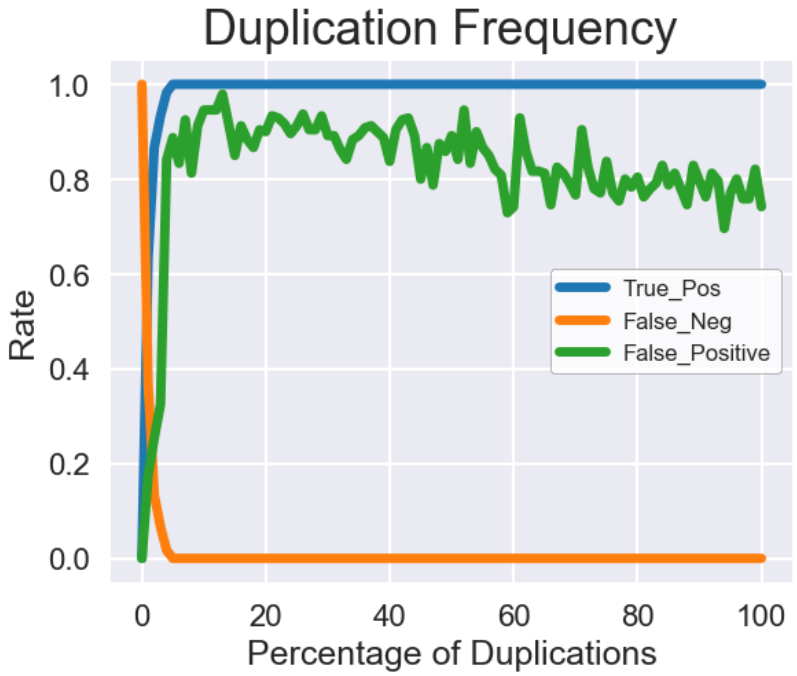


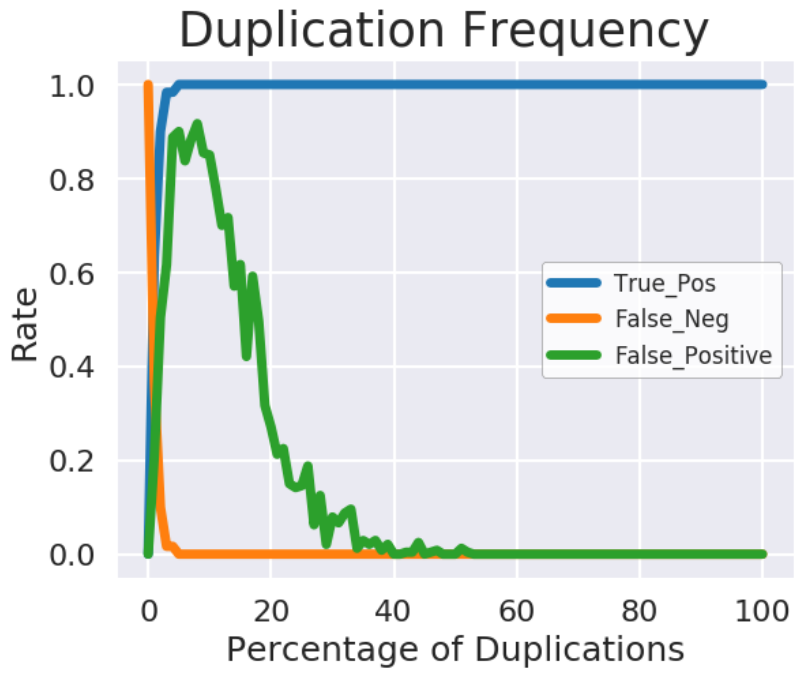
Figure 5.2: Three functions, 1000 data tuples, no crossover

function are small, as Figure. This can be improved by with more crossover. In this scenario, a single anchored provider can effect multiple service functions. Figure 5.4 shows that we can improve the false positive rate over 1000 duplications by 10%.

There are some limitations with this solution. Like its predecessor, if the majority of providers are malicious for a pipeline function. If the service function has no root of trust, the minority will be incorrectly classified as malicious. This problem can be remedied by guaranteeing an anchor in every service function. In our test environment, we aim to use the oracle as little as possible and used a single anchor. In addition, Cloud Anchor performs poorly when malicious nodes lie. Figure 5.5 shows an extremely volatile graph with a non-zero false negative rate. In these cases, we detected that a malicious node lied when being attested to the anchor and was added to the anchored set. As in the previous case, this can be corrected with the continued use of an oracle and is a topic for future work.

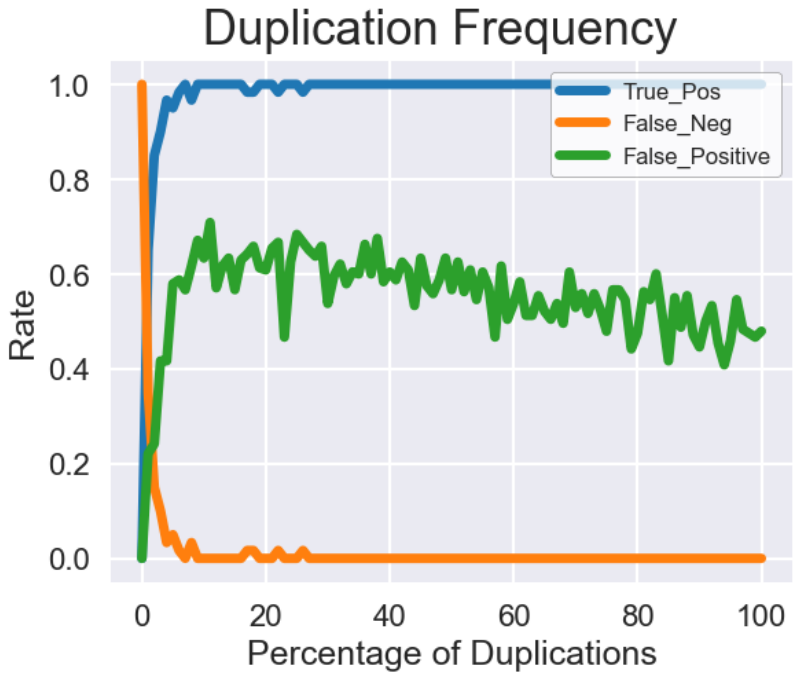


(a) IntTest

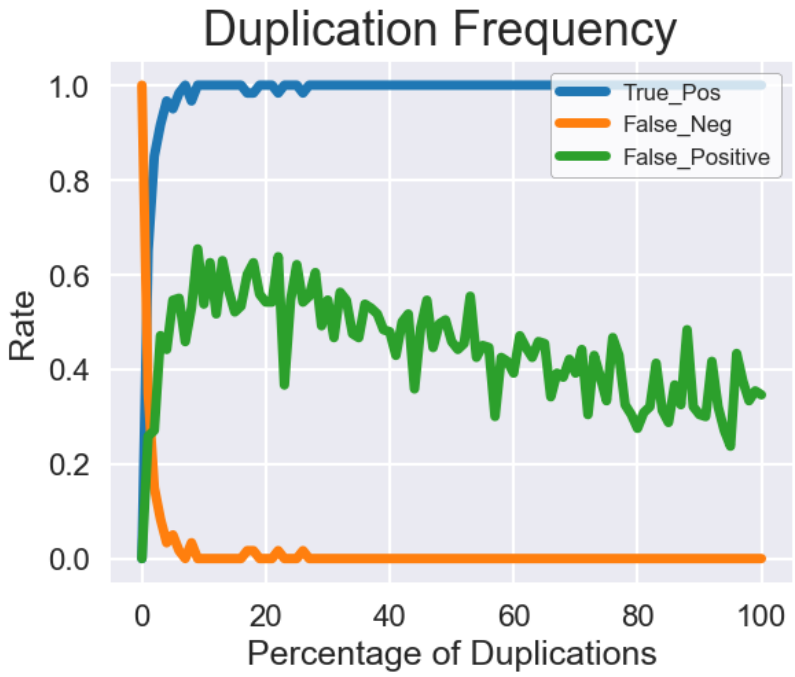


(b) Cloud Anchor

Figure 5.3: Duplication frequencies of a wide pipeline with partial crossover



(a) IntTest



(b) Cloud Anchor

Figure 5.4: Duplication frequencies of a wide pipeline with partial crossover

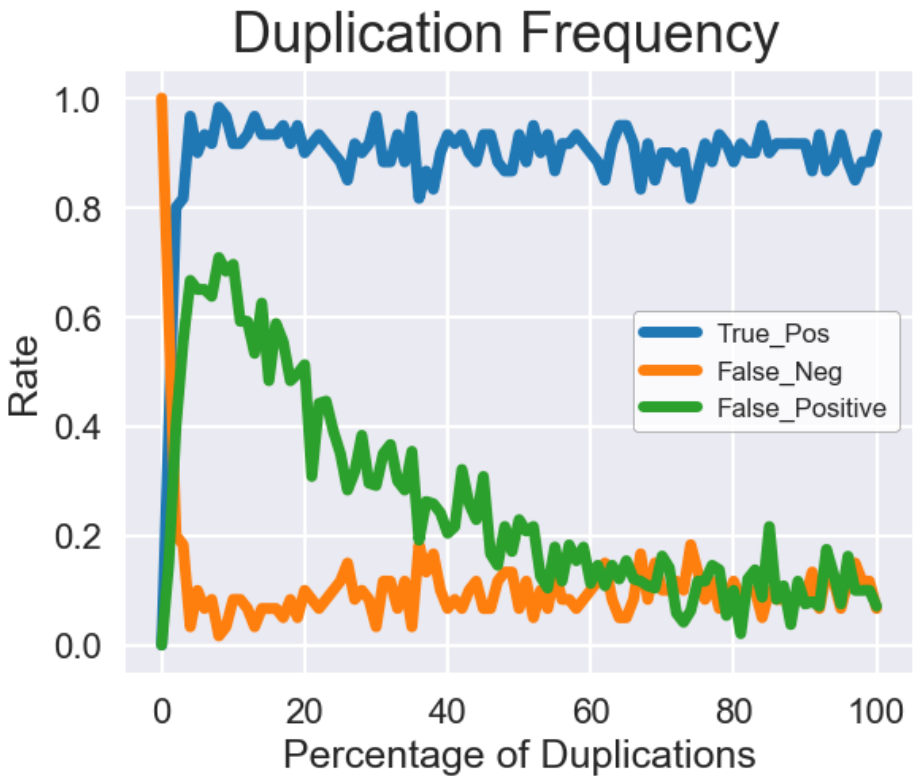


Figure 5.5: Wide service pipeline, 1000 data tuples, partial crossover, with liars

Chapter 6

Conclusions

6.1 Future Work

The current implementation of Cloud Anchor has many of the same limitations as its predecessor. If the majority of providers for a certain function are malicious, the implementation will incorrectly classify the minority. Also, if they are infrequently malicious, they may escape replay-based checks altogether.

Pipeline density is a constant problem for wide service pipelines. Wenever the data itself is being used as a means for validation, the compromise of that token can lead to a loss of information further down the pipeline. We suspect evaluating the pipeline in stages and removing nodes from the pipeline before input is fully processed can be a means to remedy this particular problem in future iterations.

Lastly, we intend to explore an implementation of the Cloud Anchor framework in a live, distributed environment. Cloud Anchor uses trust transitivity to quickly add edges to consistency attestation graphs, but we believe it can be used solve other problems for services that do not exist entirely within a secure enclave.

6.2 Conclusion

We explored IntTest the software-only service integrity framework, its benefits and its limitations. We demonstrated its inefficiency and which configurations lead to improved

or diminished performance. We showed how it responds to a realistic attack model and described a potential solution in Cloud Anchor. This solution uses an oracle to establish a root of trust and propagates that through the same graph analysis scheme used in IntTest. We showed how it greatly improves the efficiency of the previous framework and how trust transitivity may be an effective means for handling service-integrity for hybrid trust environments.

Bibliography

[ama] Amazon web services (AWS) - cloud computing services. 12

[str] Stream computing platforms, applications, and analytics - IBM. 12

[vcl] VCL. 12

[Alonso et al.] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer-Verlag. 3

[Bron and Kerbosch] Bron, C. and Kerbosch, J. Algorithm 457: Finding all cliques of an undirected graph. 16(9):575–577. 5

[Costan and Devadas] Costan, V. and Devadas, S. Intel SGX explained. 4

[Driscoll et al.] Driscoll, K., Hall, B., Paulitsch, M., Zumsteg, P., and Sivencrona, H. The real byzantine generals. In *The 23rd Digital Avionics Systems Conference (IEEE Cat. No.04CH37576)*, volume 2, pages 6.D.4–61–11 Vol.2. 4

[8] Du, J., Dean, D. J., Tan, Y., Gu, X., and Yu, T. Scalable distributed service integrity attestation for software-as-a-service clouds. 25(3):730–739. 9, 11, 28

[9] Du, J., Shah, N., and Gu, X. Adaptive data-driven service integrity attestation for multi-tenant cloud systems. In *Proceedings of the Nineteenth International Workshop on Quality of Service*, page 29. IEEE Press. 28

[10] Du, J., Wei, W., Gu, X., and Yu, T. RunTest: assuring integrity of dataflow processing in cloud computing infrastructures. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 293–304. ACM. 8, 9, 11, 22, 28

[Eldefrawy et al.] Eldefrawy, K., Tsudik, G., Francillon, A., and Perito, D. SMART: Secure and minimal architecture for (establishing dynamic) root of trust. In *NDSS*, volume 12, pages 1–15. 1, 6, 7

[Francillon et al.] Francillon, A., Nguyen, Q., Rasmussen, K. B., and Tsudik, G. A minimalist approach to remote attestation. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6. 6

[Karakostas] Karakostas, G. A better approximation ratio for the vertex cover problem. 5(4):41:1–41:8. 5

[Lamport et al.] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. 4(3):382–401. 5

[Lpez and Zhou] Lpez, J. and Zhou, J. *Wireless sensor network security*. Ios Press. 6, 7

[Morris] Morris, T. Trusted platform module. In *Encyclopedia of Cryptography and Security*, pages 1332–1335. Springer, Boston, MA. DOI: 10.1007/978-1-4419-5906-5_796. 4

[Seshadri et al.] Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., and Khosla, P. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 1–16. ACM. 1, 6, 7

[Shi et al.] Shi, E., Perrig, A., and Van Doorn, L. Bind: A fine-grained attestation service for secure distributed systems. In *Security and Privacy, 2005 IEEE Symposium on*, pages 154–168. IEEE. 1, 6, 7

Appendices

A Additional Graphs and Figures

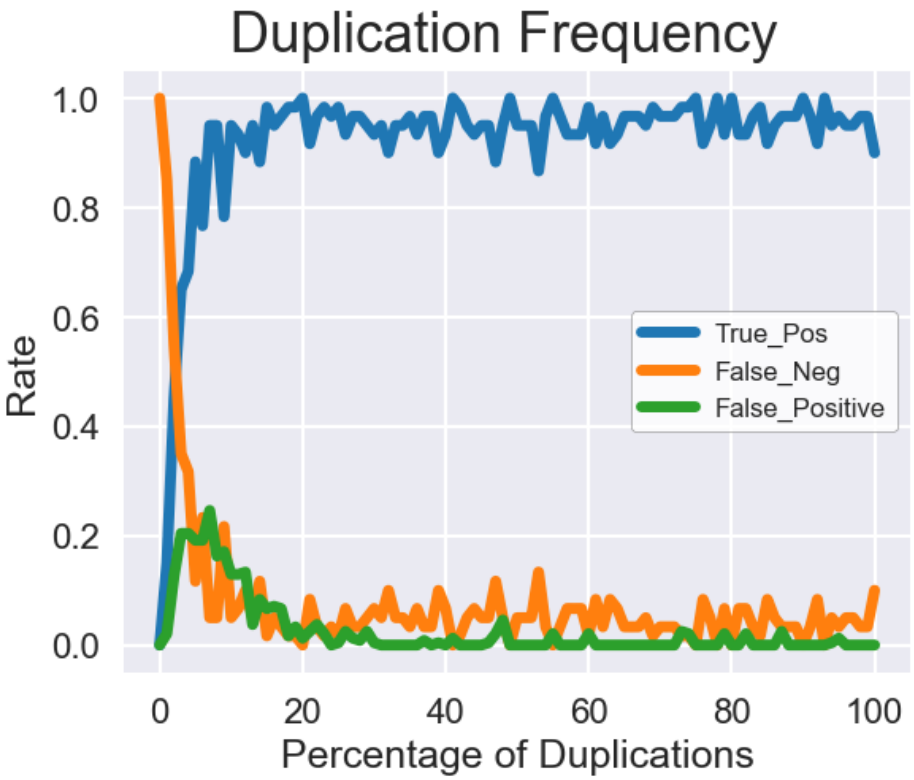


Figure A1: Three functions, 1000 data tuples, 20% malicious providers, partial crossover, with liars

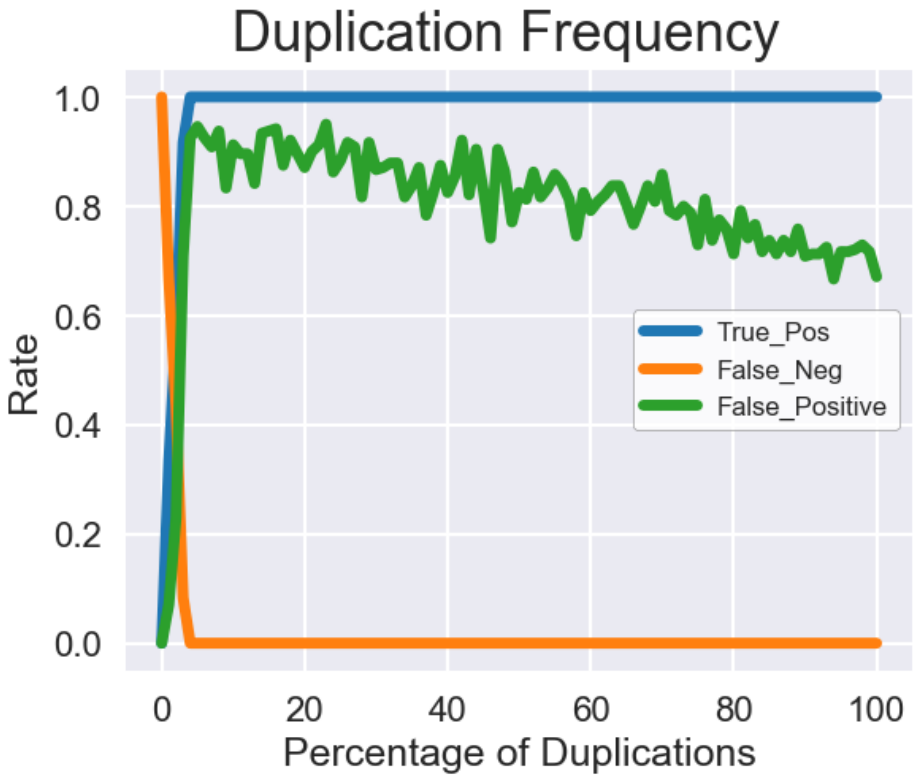


Figure A2: Three functions, 1000 data tuples, 20% malicious providers, full crossover, with liars

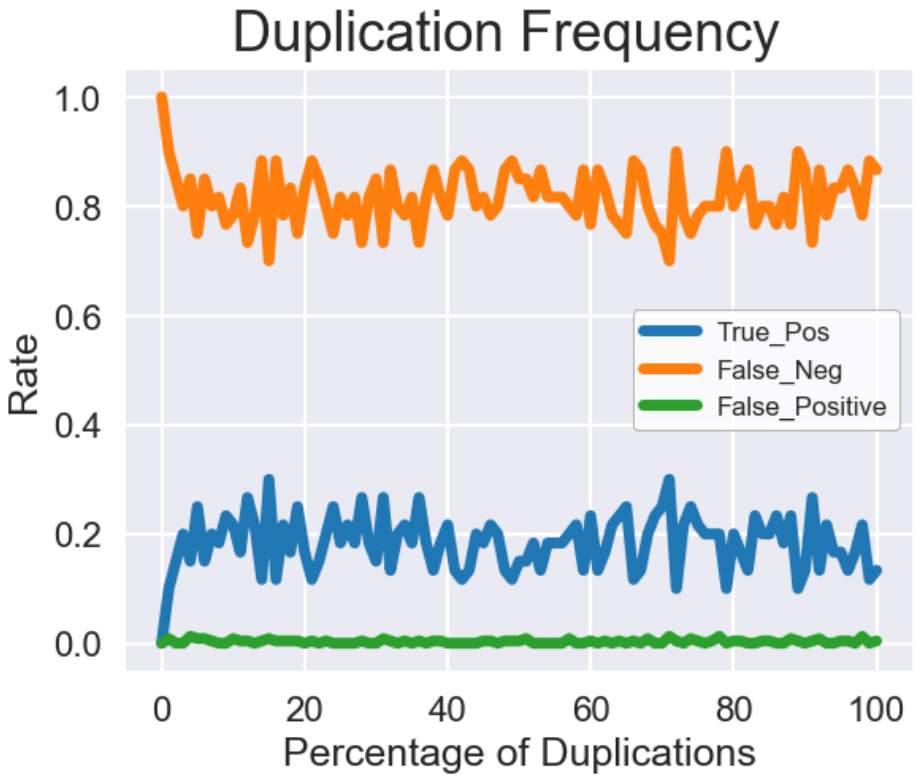


Figure A3: Ten functions, 1000 data tuples, 20% malicious providers, no crossover, with liars

Table A1: Function Densities for a Wide Pipeline

Trial Number	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
1	0.87	0.78	0.6	0.5	0.37	0.26	0.17	0.12	0.08	0.06
2	0.9	0.76	0.63	0.48	0.37	0.29	0.17	0.13	0.11	0.07
3	0.91	0.81	0.65	0.47	0.39	0.25	0.22	0.15	0.11	0.08
4	0.89	0.77	0.67	0.49	0.39	0.3	0.19	0.14	0.09	0.06
5	0.89	0.77	0.64	0.49	0.38	0.31	0.21	0.15	0.11	0.07
6	0.89	0.78	0.67	0.49	0.39	0.29	0.21	0.14	0.1	0.05
7	0.9	0.82	0.65	0.5	0.4	0.27	0.17	0.13	0.08	0.06
8	0.91	0.77	0.65	0.52	0.38	0.31	0.22	0.13	0.09	0.06
9	0.88	0.8	0.63	0.49	0.36	0.25	0.19	0.12	0.09	0.07
10	0.89	0.79	0.64	0.5	0.39	0.26	0.16	0.1	0.06	0.04

Vita

Christopher Craig is a Software Engineer, Penetration Tester, and Technology enthusiast who is working on completing his Master's Degree in Computer Science at the University of Tennessee. Christopher is a full-time employee at Oak Ridge National Labs as a Cyber Security Software Engineer developing novel technologies in 10Gb/s load balancing, network security, and distributed system security.