A Space-Time Separated and Jointly Evolving Relationship-Based Network Access and Data Protection System with NP-complete Defenses

by

David Charles Last

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
May 5, 2013

Keywords: computer, network, security

Approved by

Chwan-Hwa "John" Wu, Chair, Professor of Electrical Engineering
John A. Hamilton, Jr., Professor of Computer Science and Software Engineering
Shiwen Mao, Professor of Electrical Engineering
Darrel Hankerson, Professor of Mathematics
Xiao Qin, Professor of Computer Science and Software Engineering

Abstract

Attacks on information networks have been increasing in frequency and success in recent years. Attack methods are becoming increasingly sophisticated, and network defense systems have not kept pace. IDS and IPS systems utilizing signature- and statistics-based methods are not agile enough for today's environment. This paper presents an alternative solution; the Intrusion-resilient, Denial-of-Service resistant, Agent-assisted Cybersecurity system (IDACS). IDACS utilizes the concept of a space-time separated and jointly-evolving relationship to provide network defenses that can defend against zero-day and metamorphic attacks. IDACS provides network security in three key areas: attack detection and prevention, digital forensics to identify the origin of the attack, and deep protection of at-rest encrypted data in case of a successful network breach. IDACS combines these three aspects into a complex space-time relationship that provides mutual reinforcement between these aspects. A mathemtical analysis of IDACS reveals that several facets of its network defense are NP-complete, presenting a potential attacker with an incredibly complex problem to solve. Multiple simulations of a fielded IDACS system demonstrate the high attack detection rate, network traitor identification rate, and data protection capabilities provided by this system.

Acknowledgements

I would like to thank my professor, Dr. Chwan-Hwa "John" Wu, for his inspiration, dedication, and encouragement over the course of my PhD studies. Without his help and guidance, I would not be where I am today. I would also like to thank my committee members for their efforts and their advice concerning my research. I would like to thank my wife for her love, support, and patience through this entire process; she has made great sacrifices to support me as I pursue my dreams. I would also lik to thank the Department of Defense SMART scholarship program, which has funded me in this research over the course of my PhD studies. Finally, I would like to thank my Creator God for His blessings of life, health, intelligence, and opportunity that have brought me to this point.

Table of Contents

List Of Tables

List Of Figures

Table 1. Summary of Notation

Symbol	Name	Type	Description
SA_{χ}	Security Agent	Location	Network-side authentication machine
SSA_{κ}	Super Security Agent	Location	Network-side authorization machine
DB_{γ}	Database	Location	Network-side data storage machine
$User_{\omega}$	User	Human	Human User of the IDACS network
$Client_{\rho}$	Client computer	Location	Client-side computer (laptop, smartphone, etc.)
$Badge_{\zeta}$	User Badge	Location	Client-side smartcard security badge
Pwd_{θ}	User Password	Location	Client-side password
PIN_{λ}	Badge PIN	State	Client-side PIN entered into User Badge
UA_{β}	User Agent	virtual location	Small software application downloaded from IDACS Network to Client computer to perform security operations
$Cust_{\psi}$	Customer	State	Combination of $User_{\omega}$, $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, PIN_{λ}, and UA_{β}, authorized to access the IDACS Network
$Seed_{\sigma}$, $\overline{\boldsymbol{Seed}_{\chi}}$, $\overline{\boldsymbol{Seed}_{\varepsilon}}$, $\overline{\boldsymbol{Seed}_{\chi}} \Diamond PIN_{\lambda}$, $\overline{\boldsymbol{Seed}_{\varepsilon}} \Diamond Badge_{\zeta}$	Seed	State	Cryptographic seed stored on $Client_{\rho}$, $Badge_{\zeta}$, SA_{χ}, or SSA_{κ} or derived from Pwd_{θ} or PIN_{λ}
$\boldsymbol{\mathcal{S}}$, $\boldsymbol{\mathcal{S}}Client_{\rho}$, $\boldsymbol{\mathcal{S}}\overline{\boldsymbol{Client}}$	Super-state	State	Represents a combination of states
$Ticket_{\psi}$	Client Security Ticket	State	Data structure used to send, authenticate, and authorize a data or service request from $Cust_{\psi}$ to IDACS Network
Req_{ψ}	Merchandise Request	State	Data request that specifies target data and desired operation
OTP_{χ}, $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$	One-Time Password (OTP)	State	Used to authenticate $Cust_{\psi}$ with all SA_{χ} in $\overline{\boldsymbol{SA}}$
PID_{ε}, $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$	Pseudo-ID (PID)	State	Used to authorize $Cust_{\psi}$ and Req_{ψ} with $\overline{\boldsymbol{SA}}$ and $\overline{\boldsymbol{SSA}}$
N	Authentication Chain Length		Lenght of Authentication Chain, i.e. how many SAs and SSAs are in the approach and return authentication chains
Key(A, B)	Shared cryptographic key	State	Cryptographic key shared between locations A and B
XV_1	Xchain value	State	Cryptographic hash value calculated for authentication between machines in $\overline{\boldsymbol{SA}}$ and $\overline{\boldsymbol{SSA}}$
TK_2	Network Security Ticket	State	IDACS network message containing $Ticket_{\psi}$ and XV values
$\backslash TK_2 \backslash$	Packet record	State	Log record of the critical attributes of an IDACS network message
F-box(**L**ookup)	Lookup transform	Transform	Based on certain inputs (super-states), returns a particular ordered set of seeds from a location or state
F-box(**C**oncat)	Concatenate transform	Transform	Concatenates a set of objects
F-box(**H**ash)	Hash transform	Transform	Performs a cryptographic hash on the inputs
F-box(**N**ext)	Next-SA-SSA transform	Transform	Calculates the next SA or SSA in the authentication chain
F-box(**I**nsert)	Insert Log Record transform	Transform	Inserts a packet record *TK*\ for received network message *TK* into a location's security logs
F-box(**R**trv)	Retrieve Log Record transform	Transform	Retrieves a packet record from a location's security logs based on specified search criteria
F-box(**R**rand)	Random transform	Transform	Returns a random byte string
F-box(**O**ffset)	Data Block Offset transform	Transform	Returns the length of the next Data Block
F-box(**X**Lth)	Xslice Length transform	Transform	Returns the length of the next Xslice
F-box(**SS**tring)	Substring transform	Transform	Returns a substring of the input string
F-box(**E**ncrypt)	Encrypt transform	Transform	Encrypts the input data with the input key
A◊B		Notation	Data block A is stored at location/state B
A → B: C		Notation	Location A sends message C to location B
$\overline{\boldsymbol{E}}$={$E_1$, E_2, …}		Notation	Notation indicating a set of objects

1 Introduction

As the world proceeds farther into the Information Age, the need to protect sensitive information is increasing quickly. Many organizations are storing their information in centralized secure datacenters [1]. However, news stories over the past few years from Google, RSA, PlayStation, and others continue to remind us that information/cyber security technology has not kept pace with the capabilities of attackers. Much of the research to date in improving network defenses has focused on detecting and preventing "incorrect" network and database access [2] [3] [4] [5] [6], and allowing all other access to proceed through rigorous access control [7] [8] [9] [10] [11]. However, this security technology is unlikely to be able to defend against zero-day attacks and mutating malware [12], since these attacks present new footprints that have not yet been classified as malicious.

In order to deal with these emerging threats, the research presented in this paper approaches the problem from a different angle: mathematically define "correct" network access behavior for protected information and services, and block all other behavior. The mathematically-governed access behaviors provide sufficient complexity to be unpredictable to attackers, but are easily verified by the security system. This design should provide three mathematically-related capabilities; rigorous but fast network access control, efficient real-time forensics capabilities, and further protection of at-rest data in case of a network breach. The mathematical design that provides this level of protection is based on the theory of the Space-Time Separated and Jointly Evolving relationship. This theory calls for space-time evolving relationships between authentication credentials, file/database systems, and protected data in the realms of space and time in order to render the breaking of the access control system mathematically infeasible. Furthermore, this space-time separated and evolving relationship is encoded into network application layer packets, and becomes a means for rapidly tracing attacks back to the source attacker, thus providing real-time forensics capability. The relationship also determines the storage locations of protected data (e.g. in a cloud) and authentication credentials (e.g. on security tokens) in a time-evolving manner so that it becomes infeasible for attackers to decode the dynamic relationships. Hence, three distinct capabilities (or modules) of a security system are described by a single principle of the space-time separated and evolving relationship.

A simple example helps explain the space-time separated and jointly evolving relationship concept. Consider a military-style restricted area that uses a challenge-response system to unlock the doors to restricted areas. Any user *U* must carry an electronic codebook *CB* which contains a list of challenges and responses. This list is generated from a few cryptographic seeds unique to *U* as well as the *U*'s PIN, the state of *CB*, and the state of the current door (Fig. 1). Each door

between restricted areas presents *U* with a challenge code; *U* must use *CB* to locate the corresponding response code to open the door. As soon as *U* opens the door, the state of the door and the state of *CB* are pseudo-randomly changed with forward secrecy, resulting in a new challenge-response list if *U* attempts to re-open this door. Additionally, the cryptographic seeds associated with *U* residing on *CB* are changed at regular time intervals (e.g. Δt = 1 minute) with forward secrecy; these changes are propagated to all doors in the system. Each door presents *U* with multiple challenges, and multiple doors must be opened to access different restricted areas. Additionally, the system maintains logs of the histories of the *CB* state, the door states, and the challenge-response pairs; if an attacker *A* attempts to open a door using an older, stolen challenge-response pair, the door system can compare this pair to all previous challenge-response pairs to trace back where and when *A* stole this pair, thus identifying security breaches in the system.

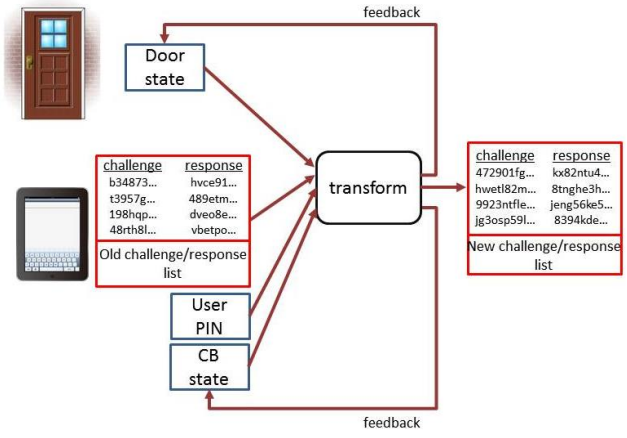


Fig. 1. *CB* and door states are used to calculate the matching between challenges and responses

This paper presents the Intrusion-resilient, Denial-of-Service resistant, Agent-assisted Cybersecurity system (IDACS). IDACS relies on the concept of the space-time separated and jointly evolving relationship to achieve a high level of security in computer and information networks. This is accomplished in three ways. First, the space-time separated and evolving relationship is used as the basis for the IDACS Network Access Control protocol. By using multiple (space-separated) time-evolving items for identifying an information or service access, e.g. file name and user ID, IDACS can efficiently allow legal access and block illegal access to the IDACS network. Second, the mathematical properties of the space-time separated and evolving relationship of the IDACS Network Access Control protocol provide a number of built-in forensics capabilities. Attacks by unauthorized users can be detected, blocked, traced back to the origin of the attack, and analyzed to determine what authentication items have been compromised, all in a very quick and efficient manner using the properties of this relationship. Third, IDACS uses the space-time separated and evolving relationship to protect at-rest encrypted data stored on network-connected devices (in the cloud or on PCs or mobile devices such as tablets or smartphones). IDACS uses jointly space-

separated and time-evolving storage to store critical pieces of at-rest ciphertext in the IDACS network so that reassembling and decrypting the ciphertext without access to the distributed pieces spread in the cloud is mathematically infeasible. The space-time separated and evolving relationship aspect of authentication seeds is transparent to legitimate users, but it presents an insurmountable barrier to attackers due to the NP-completeness of generating authentication credentials as well as the encoded file/database systems using space/time-varying IDs, locations, and protections. Additionally, this relationship aspect of authentication seeds and states contributes to the speed of the IDACS forensics capabilities. These three built-in modules of IDACS are all interconnected using the space/time relationship and all provide mutual support for each other, as indicated in Fig. 2. The "Access Control" module will be discussed in "3 IDACS Network Description and Security", the "Forensics" module will be discussed in "4 IDACS Network Attack Detection, Prevention, and Traceback", and the "Distribution" module will be discussed in "5 IDACS Protection of Encrypted Data". Additionally, the current implementation of IDACS will be discussed in "6 Implementation".

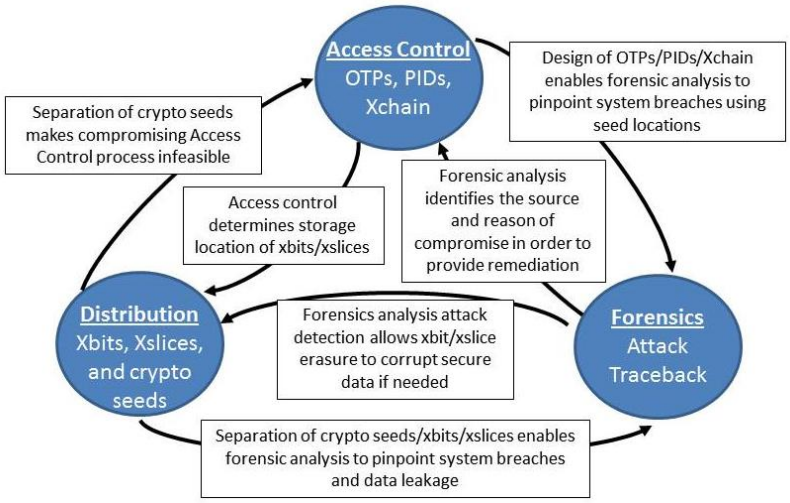


Fig. 2. Relationships between built-in modules of IDACS

2 Related Work

The research presented in the paper focuses on the detection and prevention of unauthorized network and data access. Due to the current security environment, this area has been the focus of much research. Recent research has especially focused on Denial of Service (DoS) attack detection [13] [14] [15] [16] [17] and botnet attack detection [18] [19] [20]. Until now, Intrusion Detection and Prevention systems have been using two major methods for attack detection: signatures and statistics, which both focus on the characteristics of illegal activity. In today's threat environment, however, new threats are cropping up at a prodigious rate, and many attacks go undetected by signature-based methods [21]. Researchers have begun talking of a need for a paradigm shift in the field of intrusion detection.

The research presented in this paper presents such a paradigm shift; it applies the idea of space-time separated and jointly evolving relationships to mathematically define and permit only "correct" network access. Network authentication and authorization are spread across space (multiple network authentication points) and time (time-evolving authentication credentials) with joint evolution between the space and time parameters in order to make "correct" network behavior and its history mathematically infeasible for an attacker to reconstruct. While some have previously addressed the idea of space/time relationships [22] [23], this research will use the realms of space and time jointly in previously unconsidered ways. This paper also demonstrates the real-time forensics for attack traceback capabilities and the attack report correlation and aggregation capabilities of the proposed system. While digital forensics [24] [25] [26] [14] and attack report correlation [27] [28] have been discussed in other research, space/time relationships have not been previously leveraged to provide speed and accuracy and avoid ambiguity in the manner presented in this paper. Additionally, the concept of distributed data storage has been addressed at length in prior research. However, it is typically focused on scalability [29] and redundancy for integrity and availability [30] [31]. Little work has been done in the area of distributed storage for security purposes.

3 IDACS Network Description and Security

3.1 Introduction

With the rise of the Internet and computer networks, network security has become increasingly important. Recent stories in the news have reminded us how vulnerable network-connected computer systems are, and the frightening regularity with which they are breached. Many of these breaches are the result of the exploitation of zero-day and metamorphic attacks, using previously unseen attack vectors, or metamorphic variants of known attacks, to strike at the vulnerable underbellies of networks. There is a rising need for a type of network defense system that can detect and block new and emerging threats. The proposed solution here is to use a space-time separated and jointly evolving relationship to define "correct" network behavior in a way that is easily verified by the network but mathematically infeasible for an attacker to replicate. This section introduces the Intrusion-resilient, Denial of Service resistant, Agent assisted Cybersecurity System (IDACS), which utilizes this space-time relationship to achieve a high level of network security. The elements and operations of IDACS will be introduced, and the mathematical properties of its defenses will be analyzed and proven. Additionally, simulations will be provide a demonstration of the real-world strength of IDACS against external attacks.

3.2 IDACS Description

3.2.1 Definitions for IDACS Elements and Operations

The following definitions and notation are used as the basis for the description of the IDACS network:

Definition 1 : A *location* is a physical device with an associated physical location. The physical device includes memory storage and data processing capabilities. A *virtual location* is a software object with memory storage and data processing capabilities. A virtual location is capable of residing in different physical locations.

Definition 2 : A *state* represents the PID (Definition 3) and memory contents associated with a piece of data that can change over time. It can also represent the memory contents of a physical location. The relationship between states and locations is detailed in Definition 17.

Definition 3 : A location or state may be represented by a permanent, well-protected ID, or by a time-changing *Pseudo-ID (PID)* which is derived by

PID(A) = hash(ID(A), crypto seeds, time-changing sequence number)

PID(A) may also be represented implicitly as A. Specific applications of PIDs are discussed in Definition 21.

Definition 4 : A *transform* is a mathematical operation which accepts a set of states and/or locations as inputs and produces a set of states and/or locations as outputs. In this paper, all transforms are represented by the notation *F-box()*. In this

notation, the parentheses contain a number of parameters which are inputs to the transform. The first parameter defines the actual internal operation of the transform. For example, a transform that computes a cryptographic hash of the inputs would be called F-box(hash), with "hash" being represented as $\mathfrak{H}$ash; the remaining parameters would detail the inputs to the hash function.

$$output = \text{F-box}(\mathfrak{H}\text{ash}, input\ data)$$

Transforms may be combined in a particular order in order to form new transforms. For example, a given transform may involve a lookup ($\mathfrak{L}$ookup) followed by a concatenate ($\mathfrak{C}$oncat) of the outputs of the lookup. Transforms may be combined according to the following notation:

$$output = \text{F-box}(\mathfrak{L}\text{ookup}\cdot\mathfrak{C}\text{oncat}, input\_1, input\_2, input\_3)$$

Many transforms make changes to their input superstates (e.g. $\mathcal{S}Cust_{\psi}$ as discussed in Definition 17), although these changes are abstracted in this notation.

Definition 5 : Some variables are a *function* of other variables; that is, if the value of variable A is a function of the values of variable B and time *t*, then the value of A depends on the value of B at time *t*. This relationship is represented by the notation A:f(B, *t*). This relationship implies that B is the input to an F-box() that is used to calculate the value of output A.

Definition 6 : A set of elements $\bar{\mathbf{E}} = \{E_1, E_2, \ldots E_x\}$ is a collection of elements. An *ordered* set shall be defined as a set where the ordinality (order) of the elements in the set is one of the attributes of the set. Changing the ordinality of the members of $\bar{\mathbf{E}}$ creates a different set $\bar{\mathbf{E}}'$. Therefore, if $\bar{\mathbf{E}} = \{E_1, E_2, E_3\}$ and $\bar{\mathbf{E}}' = \{E_3, E_1, E_2\}$, then $\bar{\mathbf{E}} \neq \bar{\mathbf{E}}'$. Unless specified, all sets are unordered.

3.2.2 IDACS System Components

The IDACS Network previously discussed is composed of a number of network entities. These items are defined in Definition 7 to Definition 16, and they are shown graphically in Fig. 3. Given the IDACS Network, it contains the elements shown in Table 2.

Table 2. IDACS System Elements						
	..a set of	**... termed**	**...which are**	**There are**	**The set of**	**...is represented as**
Definition 7	servers	Security Agents (SAs)	locations.	q SAs SA_χ, $\chi \in [1, q]$.	SA_χ	$\overline{\boldsymbol{SA}} = \{SA_1, SA_2, \ldots, SA_q\}$
Definition 8	servers	Super Security Agents (SSAs)	locations.	n SSAs SSA_κ, $\kappa \in [1, n]$.	SSA_κ	$\overline{\boldsymbol{SSA}} = \{SSA_1, SSA_2, \ldots, SSA_n\}$
Definition 9	servers	Databases	locations.	h Databases DB_γ, $\gamma \in [1, h]$	DB_γ	$\overline{\boldsymbol{DB}} = \{DB_1, DB_2, \ldots, DB_h\}$
Definition 10	humans	Users	humans.	u Users, $User_\omega$, $\omega \in [1, u]$.	$User_\omega$	$\overline{\boldsymbol{User}} = \{User_1, \ldots, User_u\}$
Definition 11	Computers/Devices	Clients	locations.	z Clients, $Client_\rho$, $\rho \in [1, z]$.	$Client_\rho$	$\overline{\boldsymbol{Client}} = \{Client_1, \ldots, Client_z\}$
Definition 12	smartcards	Badges	locations.	y Badges $Badge_\zeta$, $\zeta \in [1, y]$.	$Badge_\zeta$	$\overline{\boldsymbol{Badge}} = \{Badge_1, \ldots, Badge_y\}$
Definition 13	user passwords		states.	θ User Passwords Pwd_θ, $\theta \in [1, w]$.	Pwd_θ	$\overline{\boldsymbol{Pwd}} = \{Pwd_1, Pwd_2, \ldots, Pwd_w\}$
Definition 14	Badge PINs		states.	x Badge PINs PIN_λ, $\lambda \in [1, x]$.	PIN_λ	$\overline{\boldsymbol{PIN}} = \{PIN_1, PIN_2, \ldots, PIN_x\}$

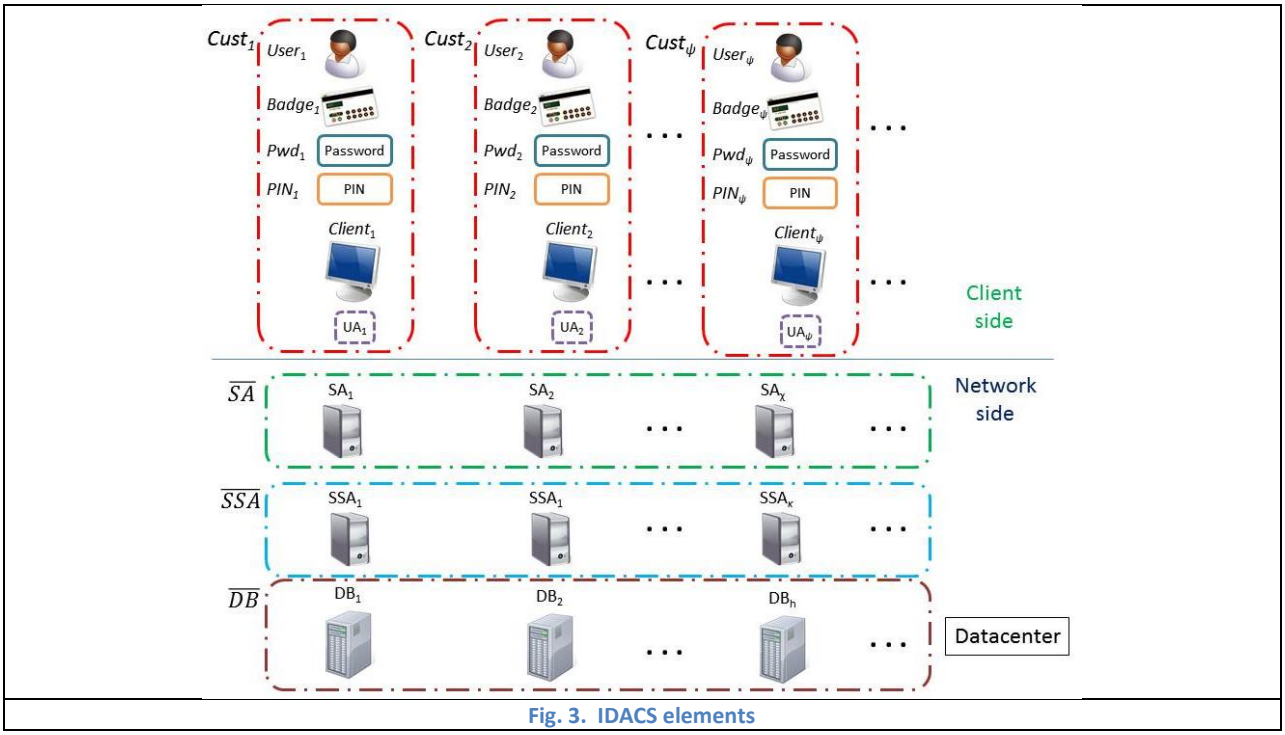


Fig. 3. IDACS elements

Property 1 : All $Pwd_\theta \in \overline{\boldsymbol{Pwd}}$ and all $PIN_\lambda \in \overline{\boldsymbol{PIN}}$ are stored in the brains of $User_\omega \in \overline{\boldsymbol{User}}$, and are not stored at any other location. However, cryptographic hashes of all $Pwd_\theta \in \overline{\boldsymbol{Pwd}}$ and all $PIN_\lambda \in \overline{\boldsymbol{PIN}}$ are stored at locations (SAs and SSAs) that are required to verify these Pwd_θ and PIN_λ. This space-separated relationship allows Pwd_θ and PIN_λ to be verified when they are provided by $User_\omega$, while providing no useful information (due to the one-way property of the cryptographic hash) to an attacker who gains access to a location's memory contents.

Definition 15 : Given the IDACS Network, when $User_\omega$ seeks to use $Client_\rho$ to communicate with the IDACS servers at time t, $Client_\rho$ downloads a unique *User Agent* software program UA_β from the network. This UA_β handles all communications between $Client_\rho$ and the IDACS servers. UA_β is considered a virtual location. UA_β is a function of $User_\omega$, $Client_\rho$, and time,

thus UA_β: f($User_\omega$, $Client_\rho$, t). UA_β is the entity that will be performing most of the operations on the client side in the IDACS Network, so the following definitions and procedures will reference a single UA_β.

Definition 16 : Given the IDACS Network, at time t there are c sets of $User_\omega$, $Client_\rho$, $Badge_\zeta$, Pwd_θ, PIN_λ, and UA_β (denoted as {$User_\omega$, $Client_\rho$, $Badge_\zeta$, Pwd_θ, PIN_λ, UA_β}) that are authorized to access the network. These combinations are termed Customers $Cust_\psi$, $\psi \in [1, c]$. $Cust_\psi$ is considered a state. Since $Cust_\psi$ represents a combination of the other parameters, $Cust_\psi$: f($User_\omega$, $Client_\rho$, $Badge_\zeta$, Pwd_θ, PIN_λ, UA_β, t).

Definition 17 : Given the locations defined in the IDACS network, some of the following definitions will depend on the state that describes the configuration and memory contents of a combination of certain locations. These states represent a combination of other states as defined in Definition 2, so they are termed super-states. The symbol $\mathcal{S}$ represents the super-state covering the entire IDACS system, with other symbols representing more narrowly-defined super-states that are subsets of $\mathcal{S}$, e.g. $\mathcal{S}Client_\rho$ represents the state of $Client_\rho$ in combination with UA_β.

$$\mathcal{S}Client_\rho\text{: f}(Client_\rho, UA_\beta, t)$$

The definition of $\mathcal{S}$ depends mainly on the memory contents of different locations and the results of the lookup transform as defined in Definition 24. Similar notation is used for $Badge_\zeta$, $\overline{\boldsymbol{Badge}}$, PIN_λ, $\overline{\boldsymbol{PIN}}$, Pwd_θ, $\overline{\boldsymbol{Pwd}}$, SA_χ, $\overline{\boldsymbol{SA}}$, SSA_κ, and $\overline{\boldsymbol{SSA}}$. These super-states represent the basis of the space-time separated and jointly evolving relationship in IDACS.

The locations, states, transforms, and notation that are defined here and in the following sections are summarized in Table 1 for easy reference.

3.2.3 IDACS Client-side Authentication and Access Control

3.2.3.1 IDACS Client-side Elements, Operations, and Definitions

The definitions given in the previous section form the basis for the description of the IDACS system. This section will build on those definitions and expand on the Client-side operations of IDACS. It details how the IDACS Network Access Control protocol is handled for Customer authentication and authorization to allow customers to access data or services residing on a DB.

Definition 18 : Given the set $\overline{\boldsymbol{Cust}}$, in order for $Cust_\psi$ to initiate communications with the IDACS servers (SAs and SSAs) and perform network actions (Read/Write/Execute a piece of data on DB_γ), $Cust_\psi$ must present a Client Security Ticket $Ticket_\psi$ to the IDACS network. $Ticket_\psi$ is considered a state. $Ticket_\psi$ is a function of both $Cust_\psi$ and time t; thus, $Ticket_\psi$:f($Cust_\psi$, t). $Ticket_\psi$ is the set containing the sets $\overline{\boldsymbol{OTP_\psi}}$ and $\overline{\boldsymbol{PID_\psi}}$, and the state Req_ψ (all defined in the following definitions); i.e. $Ticket_\psi = \{\overline{\boldsymbol{OTP_\psi}}, \overline{\boldsymbol{PID_\psi}}, Req_\psi\}$.

Definition 19 : Given $Ticket_{\psi}$, it requires a *Merchandise Request* Req_{ψ} to communicate the specifics of the desired network action. Req_{ψ} is considered a state. Req_{ψ} specifies the request type (Read/Write/Execute a piece of data on DB_{γ}), the unique PID for $Cust_{\psi}$, the Content(PID_{ε}) tied to the specified data (as defined in Definition 22), and the data itself. The mechanics of the formation of Req_{ψ} also depend on $\mathcal{S}Cust_{\psi}$; Req_{ψ}: f($\mathcal{S}Cust_{\psi}$, $\overline{\boldsymbol{PID_{\psi}}}$).

Definition 20 : Given $Ticket_{\psi}$, it requires a set $\overline{\boldsymbol{OTP_{\psi}}}$ of q *One-Time Passwords (OTP)* OTP_{χ}, $\chi \in [1, q]$. Since all OTP_{χ} are data structures, they are considered states. These OTP_{χ} are used for pairwise authentication between $Cust_{\psi}$ and each SA_{χ}. Each calculated OTP_{χ} is a function of the $Cust_{\psi}$ calculating it, the SA_{χ} which will be verifying it, and time t.; thus, OTP_{χ}:f($Cust_{\psi}$, SA_{χ}, χ, t). The set $\overline{\boldsymbol{OTP_{\psi}}}$ of OTP_{χ} for all SA_{χ}, which is calculated by the UA_{β} associated with $Cust_{\psi}$ is represented as $\overline{\boldsymbol{OTP_{\psi}}}$ = {OTP_1, OTP_2, …, OTP_q}. $\overline{\boldsymbol{OTP_{\psi}}}$: f($Cust_{\psi}$, $\mathcal{S}\overline{\mathbf{SA}}$, t). Algorithm 2 illustrates the procedure by which OTP_{χ} is calculated.

Definition 21 : Given $Ticket_{\psi}$, it also requires a set $\overline{\boldsymbol{PID_{\psi}}}$ of r *Psuedo IDs (PID)* PID_{ε}, $\varepsilon \in [1, r]$. Since all PID_{ε} are data structures, they are considered states. These PID_{ε} are used for access control; they verify the identity of $Cust_{\psi}$ as well as the permissions of $Cust_{\psi}$ to perform the requested network action in Req_{ψ}, and they identify the information associated with Req_{ψ} residing on DB_{γ}. Each calculated PID_{ε} is a function of the associated $Cust_{\psi}$ and Req_{ψ}, the index ε, and time t. Thus, PID_{ε}: f($Cust_{\psi}$, Req_{ψ}, ε, t). The set $\overline{\boldsymbol{PID_{\psi}}}$ of PID_{ε} calculated by the UA_{β} associated with $Cust_{\psi}$ to authorize with the network is represented as $\overline{\boldsymbol{PID_{\psi}}}$ = {PID_1, PID_2, …, PID_r}. $\overline{\boldsymbol{PID_{\psi}}}$: f($Cust_{\psi}$, Req_{ψ}, t). Algorithm 3 illustrates the procedure by which PID_{ε} is calculated.

Definition 22 : Given $\overline{\boldsymbol{PID_{\psi}}}$, one of the PID_{ε} in $\overline{\boldsymbol{PID_{\psi}}}$ is tied to the specific piece of data (merchandise) specified in Req_{ψ}. This particular PID_{ε} is called the *Content PID*; it is represented by Content(PID_{ε}). The Content PID indicates the data being accessed in a Read or Execute operation, or establishes a data PID for future reference in a Write operation. Permission is granted to different $Cust_{\psi}$ to access different pieces of data residing on DB_{γ}; checking the permissions of $Cust_{\psi}$ to access a requested piece of data is part of the IDACS Network Access Control mechanism. To protect Content(PID_{ε}) for data residing in $\overline{\boldsymbol{DB}}$ from attacks against relatively less-protected SAs, the information needed to calculate Content(PID_{ε}) resides only on the relatively better-protected SSAs; only SSAs are capable of verifying Content(PID_{ε}). This is reflected in the simulations in "3.4 Simulations".

Property 2 : Although $\overline{\boldsymbol{OTP_{\psi}}}$ and $\overline{\boldsymbol{PID_{\psi}}}$ are calculated similarly, they serve separate functions. The elements of $\overline{\boldsymbol{OTP_{\psi}}}$ are used for authentication to verify the identity of $Cust_{\psi}$, while the elements of $\overline{\boldsymbol{PID_{\psi}}}$ are used for access control to verify that

$Cust_\psi$ is allowed to perform the action specified in Req_ψ on the data specified by Content(PID_ε). $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ provides space-time separated and evolving authentication (Property 3), while $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ provides per-customer and per-data access control which enforces broader group-based access policies.

Notation: When a piece of data A is stored at a location or state B at time t, this is indicated by the notation A◊(B, t). However, the time parameter is often abstracted, so the notation is simplified to A◊B.

Definition 23 : Given $\overline{\boldsymbol{Client}}$, every $Client_\rho$ can store up to ς <u>cryptographic seeds</u> $Seed_\sigma$, $\sigma \in [1, \varsigma]$. $Seed_\sigma$ is considered to be a state. The set of all $Seed_\sigma \lozenge Client_\rho$ is represented as $\overline{\boldsymbol{Seed}} \lozenge Client_\rho = \{Seed_1 \lozenge Client_\rho, Seed_2 \lozenge Client_\rho, \ldots, Seed_\varsigma \lozenge Client_\rho\}$. These relationships are represented by $Seed_\sigma \lozenge Client_\rho$: f($Client_\rho$, t) and $\overline{\boldsymbol{Seed}} \lozenge Client_\rho$: f($Client_\rho$, t). All $Badge_\zeta$ can also store a set $\overline{\boldsymbol{Seed}} \lozenge Badge_\zeta$ of $Seed_\sigma \lozenge Badge_\zeta$, so $Seed_\sigma \lozenge Badge_\zeta$: f($Badge_\zeta$, t) and $\overline{\boldsymbol{Seed}} \lozenge Badge_\zeta$: f($Badge_\zeta$, t). Additionally, $Seed_\sigma$ can be derived from Pwd_θ and PIN_λ by applying the cryptographic hash function to a concatenation of Pwd_θ or PIN_λ with pseudo-random nonces and time-evolving sequence numbers. Thus, $Seed_\sigma \lozenge Pwd_\theta$: f($Pwd_\theta$, t), $\overline{\boldsymbol{Seed}} \lozenge Pwd_\theta$: f($Pwd_\theta$, t), $Seed_\sigma \lozenge PIN_\lambda$: f($PIN_\lambda$, t), and $\overline{\boldsymbol{Seed}} \lozenge PIN_\lambda$: f($PIN_\lambda$, t).

Definition 24 : Given the sets $\overline{\boldsymbol{Pwd}}$, $\overline{\boldsymbol{PIN}}$, $\overline{\boldsymbol{Badge}}$, or $\overline{\boldsymbol{Client}}$, each Pwd_θ, PIN_λ, $Badge_\zeta$, or $Client_\rho$ stores (q + r) *ordered* sets $\overline{\boldsymbol{Seed}}_{OTP,\chi}$ or $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$ each consisting of j seeds $Seed_\sigma \lozenge Pwd_\theta$, $Seed_\sigma \lozenge PIN_\lambda$, $Seed_\sigma \lozenge Badge_\zeta$, or $Seed_\sigma \lozenge Client_\rho$. Each set is needed to calculate one OTP_χ or one PID_ε, respectively. The <u>F-box(lookup)</u> transform takes the item type (*OTP* or *PID*), the index (χ or ε), the super-state $\mathcal{S}Client_\rho$, $\mathcal{S}Badge_\zeta$, $\mathcal{S}PIN_\lambda$, or $\mathcal{S}Pwd_\theta$ (which provides the seeds and states), and $\mathcal{S}Cust_\psi$ (which determines the order of the seeds in different $\overline{\boldsymbol{Seed}}_{OPT,\chi}$ and $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$) as inputs; and outputs the ordered set of $Seed_\sigma$ which corresponds to the item type and index. This transform is represented by

$$\overline{\boldsymbol{Seed}}_{OTP,\chi} \lozenge Client_\rho = \text{F-box}(\mathcal{L}\text{ookup}, \mathcal{S}Cust_\psi, \mathcal{S}Client_\rho, OTP, \chi)$$

where $\mathcal{S}Client_\rho$ can be replaced by $\mathcal{S}Badge_\zeta$, $\mathcal{S}PIN_\lambda$, or $\mathcal{S}Pwd_\theta$, and (*OTP*, χ) may be replaced by (*PID*, ε). For some combinations of inputs, the output set may be an empty set, i.e. j = 0 and $\overline{\boldsymbol{Seed}}_{OTP,\chi} \lozenge Client_\rho = \emptyset$.

Property 3 : The members of $\overline{\boldsymbol{Seed}}_{\boldsymbol{\chi}} \lozenge Client_\rho$, $\overline{\boldsymbol{Seed}}_{\boldsymbol{\varepsilon}} \lozenge Badge_\zeta$, etc. are not stored consecutively in their respective locations; they are stored randomly in that location's memory. Additionally, based on the IDACS state history and nonces, their positions in memory are changing in time with forward secrecy. This provides the space-time separation and the space-time joint evolution of IDACS. Because of this, the F-box($\mathcal{L}$ookup) transform is non-trivial for an attacker to break and gives the strength to Theorem 1.

Definition 25 : Given a group of *n* generic objects O_1, O_2, …, O_n, the *F-box(concatenate)* transform accepts this group of objects as input and outputs the objects concatenated into an ordered set. The generic objects may be individual objects, or they may be sets of objects. In equation notation, the "concatenate" is represented by **C**oncat. For example,

$$\overline{\boldsymbol{Seed}}_{OTP,\chi} = \text{F-box}(\mathbf{C}\text{oncat}, \overline{\boldsymbol{Seed}}_{OTP,\chi} \lozenge Client_{\rho}, \overline{\boldsymbol{Seed}}_{OTP,\chi} \lozenge Badge_{\zeta}, \overline{\boldsymbol{Seed}}_{OTP,\chi} \lozenge PIN_{\lambda}, \overline{\boldsymbol{Seed}}_{OTP,\chi} \lozenge Pwd_{\theta})$$

$$\overline{\boldsymbol{OTP}}_{\psi} = \text{F-box}(\mathbf{C}\text{oncat}, OTP_1, OTP_2, \ldots, OTP_n)$$

Definition 26 : The *F-box(random)* transform generates a random byte array. For this research, the array is typically 256 bits long (corresponding to the SHA-256 hash algorithm).

$$XV_1 = \text{F-box}(\mathfrak{m}\text{rand})$$

Definition 27 : Given a generic set of inputs, the *F-box(hash)* transform applies a cryptographic hash function (e.g. SHA-256) to a byte array representation of the inputs and outputs the resulting byte array.

$$\text{output} = \text{F-box}(\mathbf{H}\text{ash}, \text{inputs})$$

A specific instance of this transform operates as follows. Given an item type *OTP* or *PID* of index χ or ε, the transform accepts the item type(*OTP* or *PID*), the index (χ or ε), and the associated set of seeds i.e $\overline{\boldsymbol{Seed}}_{OTP,\chi}$ or $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$ as inputs. The output OTP_{χ} or PID_{ε} is calculated by applying the cryptographic hash to $\overline{\boldsymbol{Seed}}_{OTP,\chi}$ or $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$combined with well-known (system-wide for IDACS and publically known) values and a time-evolving sequence number. Different but well-known values and order of the seeds are used for each OTP_{χ} or PID_{ε}; thus, each OTP_{χ} or PID_{ε} is calculated differently, but the calculation method is well-known. The time-evolving sequence number is used to accomplish anti-replay functionality of the output.

$$OTP_{\chi} = \text{F-box}(\mathbf{H}\text{ash}, \mathcal{S}Cust_{\psi}, \overline{\boldsymbol{Seed}}_{OTP,\chi}, OTP, \chi)$$

Property 4 : Due to Property 3, the outputs of the F-box(**L**ookup) transform and the composition of $\overline{\boldsymbol{Seed}}_{OTP,\chi}$ and $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$ are drawn from space-separated elements that are time-evolving with forward secrecy. Additionally, when a OTP_{χ} or PID_{ε} is being calculated using $\overline{\boldsymbol{Seed}}_{OTP,\chi}$ or $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$ as inputs to F-box(**H**ash), a time-evolving sequence number is used as another of the inputs. As a result, the $\overline{\boldsymbol{OTP}}_{\psi}$ and $\overline{\boldsymbol{PID}}_{\psi}$ that depend on these values are also space-time separated and jointly evolving. An attacker who intercepts $\overline{\boldsymbol{OTP}}_{\psi}$, $\overline{\boldsymbol{PID}}_{\psi}$, or any $\overline{\boldsymbol{Seed}}_{OTP,\chi}$ or $\overline{\boldsymbol{Seed}}_{PID,\varepsilon}$will be unable to use them after the sequence number or any of the $Seed_{\sigma}$ have changed, as they will be invalid. Additionally, an attacker cannot use

an intercepted OTP_{χ} or PID_{ε} to obtain any information regarding the $Seed_{\sigma}$ used to calculate them (due to the one-way property of the cryptographic hash function).

Given Definition 17, Definition 24, and the related Property 3, the following Theorems may be formed. Both of these Theorems are proved in "3.3 Proofs for Theorem 1 and Theorem 2":

Theorem 1 : Given the F-box(Lookup) transform, which takes as inputs (a) a super-state $\mathcal{S}Client_{\rho}$, $\mathcal{S}Badge_{\zeta}$, $\mathcal{S}PIN_{\lambda}$, or $\mathcal{S}Pwd_{\theta}$ (which contain cryptographic seeds), (b) the super-state $\mathcal{S}Cust_{\psi}$, and (c) an *OTP* or *PID* index ((b) and (c) are used together to determine the identity and order of the seeds returned by the F-box(Lookup) transform). This transform returns an *ordered* subset of the seeds derived from (a). An attacker who wishes to recreate the F-box(Lookup) transform and has access to (c) and all or part of (a) but not (b) faces an NP-complete problem due to the *order* of the output seeds.

Theorem 2: Given the IDACS system and an attacker who is trying to calculate $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ without access to the super-states $\mathcal{S}Client_{\rho}$, $\mathcal{S}Badge_{\zeta}$, $\mathcal{S}PIN_{\lambda}$, $\mathcal{S}Pwd_{\theta}$, or $\mathcal{S}Cust_{\psi}$. Such an attacker must reassemble $\mathcal{S}Cust_{\psi}$ (which contains $\mathcal{S}Client_{\rho}$, $\mathcal{S}Badge_{\zeta}$, $\mathcal{S}PIN_{\lambda}$, and $\mathcal{S}Pwd_{\theta}$) in order to successfully calculate $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$. It is an NP-complete problem for the attacker to reassemble $\mathcal{S}Cust_{\psi}$ or any other super-state.

3.2.3.2 IDACS Client-side Algorithms

The Client-side operations for IDACS network authentication and authorization may now be detailed in terms of these definitions. Algorithm 1 outlines the entire IDACS Network Access Control procedure, and the Client-side operations are detailed in Algorithm 2 and Algorithm 3. The network-side operations will be detailed in the next section.

Notation: The notation A → B: C indicates that message C is being sent from party A to party B.

In Algorithm 1, UA_{β} first calculates the $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ (1) and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ (2) needed to authenticate the data request Req_{ψ} (3) with the IDACS network and packages them together into $Ticket_{\psi}$ (4). $Cust_{\psi}$ then sends $Ticket_{\psi}$ to a pre-determined SA_1 (5) to begin the network access control process. The network access control module defined in Algorithm 4 uses SAs and SSAs to authenticate $Ticket_{\psi}$ as many times as necessary (as defined by the authentication chain length, *N*) according to the specific IDACS implementation (7 and 8) using randomly generated XV_1 and XV_4 values (6) for the first iteration of the module. If the network access control module fails at any time, the data request is dropped (10). After the network access control module has been run several times, the final SSA to handle $Ticket_{\psi}$ sends Req_{ψ} to DB_{γ} for processing (11). Algorithm 1 (and all contained sub-algorithms) is outlined in Fig. 4. The $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ used in Algorithm 1 are calculated according to Algorithm 2 and Algorithm 3.

Algorithm 1. IDACS Network Access Control procedure

```
input: Cust_ψ, 𝒮SA, 𝒮SSA, 𝒮DB, data operation (Read/Write/Execute), data, SA_1, N
output: Cust_ψ, 𝒮SA, 𝒮SSA, 𝒮DB

        at UA_β
1           OTP_ψ = calculate_OTPs(𝒮Cust_ψ, Cust_ψ)
2           PID_ψ = calculate_PIDs(𝒮Cust_ψ, Cust_ψ)
3           Req_ψ = F-box(𝒞oncat, (Read/Write/Execute), PID(Cust_ψ), Content(PID_ε), data)
4           Ticket_ψ = F-box(𝒞oncat, OTP_ψ, PID_ψ, Req_ψ)

5       Cust_ψ → SA_1 : Ticket_ψ

6       XV_1 = F-box(ℳrand)      XV_4 = F-box(ℳrand)
7       for n = 1 to N
8           { XV_1, XV_4, SA_1, SSA_2, passed } = run_auth_chain(SA_1, XV_1, XV_4, Ticket_ψ)
9           if (passed = false)
10              exit alorithm
            end
        end

11      SSA_2 → DB_γ : Req_ψ
```

In Algorithm 2, if $\overline{OTP_\psi}$ is being calculated by $Cust_\psi$ (1), each individual OTP_χ (2) is calculated by gathering the relevant seeds from the relevant $Client_\rho$, $Badge_\zeta$, PIN_λ, and Pwd_θ (3), and hashing them together (4). The individual OTP_χ are concatenated together to generate the set $\overline{OTP_\psi}$ (5). If the calculation is being performed by SA_χ (7), then only OTP_χ is needed to be authenticated rather than the entire set $\overline{OTP_\psi}$. Thus, all of the relevant seeds are gathered and hashed together to generate OTP_χ (8). On the other hand, the entire set $\overline{PID_\psi}$ is generated by $Cust_\psi$ and the entire set is authenticated by each SA_χ or SSA_κ (excluding Content(PID_ε), which is only checked by SSA_κ). In Algorithm 3, $Cust_\psi$ gathers all of the relevant seeds from $Client_\rho$, $Badge_\zeta$, PIN_λ, and Pwd_θ (3) and hashes them together (4) for each individual PID_ε (1). Any SA_χ or SSA_κ gathers the seeds and hashes them together (6) to generate each individual PID_ε (1). Finally, all PID_ε are concatenated to form $\overline{PID_\psi}$ (7). The procedure for calculating a single OTP_χ at $Cust_\psi$ in Algorithm 2 is outlined in Fig. 5; the procedure for calculating a single PID_ε at $Cust_\psi$ is similar.

Algorithm 2. calculate_OTPs()

```
input: location, Cust_ψ
output: OTP_ψ or OTP_χ

1   if (location == 𝒮Cust_ψ)
2       for χ=1 to q
3           Seed_OTP,χ = F-box(ℒookup·𝒞, 𝒮Cust_ψ, OTP, χ)
4           OTP_χ = F-box(ℋash, 𝒮Cust_ψ, Seed_OTP,χ, OTP, χ)
        end
5       OTP_ψ = F-box(𝒞oncat, OTP_1, OTP_2, ..., OTP_q)
6       return OTP_ψ

7   else if (location == 𝒮SA_χ)
8       OTP_χ = F-box(ℒookup·ℋash, PID(Cust_ψ), 𝒮SA_χ, OTP, χ)
9       return OTP_χ
    end
```

Algorithm 3. calculate_PIDs()

```
input: location, Cust_ψ
output: PID_ψ

1   for ε=1 to r
2       if (location == 𝒮Cust_ψ)
3           Seed_PID,ε = F-box(ℒookup·𝒞, 𝒮Cust_ψ, PID, ε)
4           PID_ε = F-box(ℋash, 𝒮Cust_ψ, Seed_PID,ε, PID, ε)
5       else if (location == 𝒮SA_χ or 𝒮SSA_κ)
6           PID_ε = F-box(ℒookup·ℋash, PID(Cust_ψ), 𝒮location, PID, ε)
        end
    end

7   PID_ψ = F-box(𝒞oncat, PID_1, PID_2, ..., PID_r)
8   return PID_ψ
```

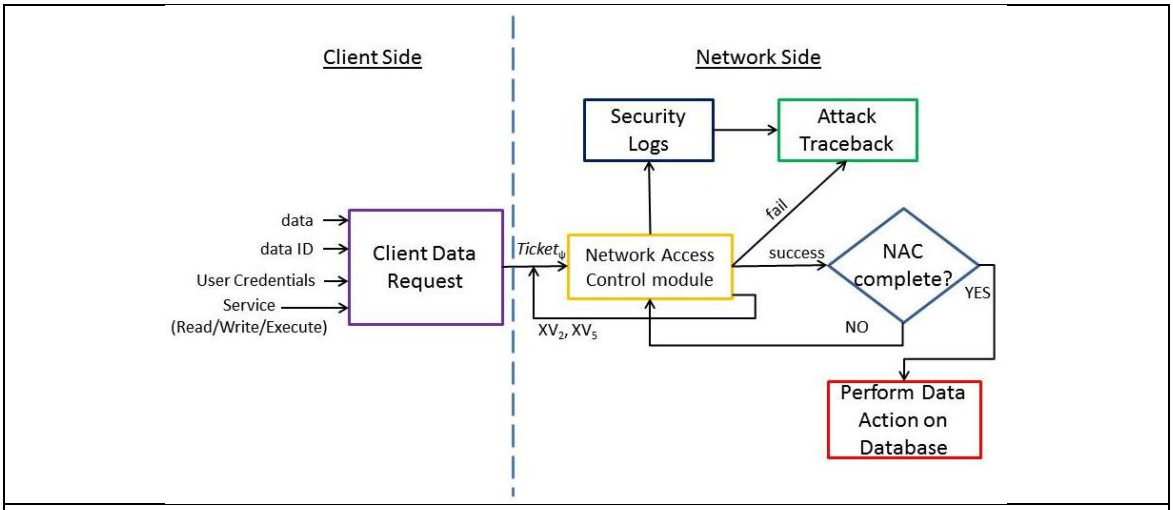


Fig. 4. IDACS Network Access Control top-level view

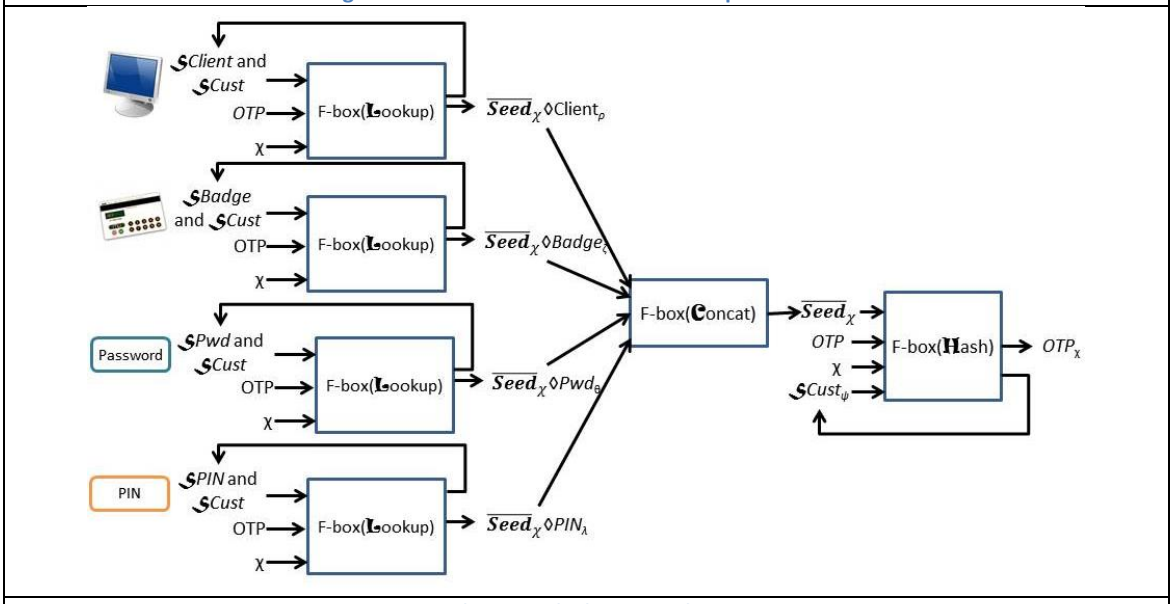


Fig. 5. Procedure to calculate a single OTP_χ at $Cust_\psi$

3.2.4 IDACS Network-side Authentication and Access Control

3.2.4.1 IDACS Network-side Elements, Operations, and Definitions

The previous section outlined the IDACS Client-side authentication and authorization procedures for gaining access to data stored on a DB. This section will discuss the corresponding Network-side procedures that verify $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ in order to authenticate $Cust_\psi$ and grant DB access.

Definition 28 : Given $\overline{\mathbf{SA}}$ and $\overline{\mathbf{SSA}}$, each pair of two machines from these sets share a cryptographic key. This key is used for encryption and cryptographic hash functions. A cryptographic key shared between machines A and B is represented as *Key*(A, B). These shared keys are referenced in Algorithm 4.

Definition 29 : Given $Ticket_{\psi}$, $\overline{\mathbf{SA}}$, and $\overline{\mathbf{SSA}}$, $Ticket_{\psi}$ must be authenticated ($\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$) and authorized ($\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$) by multiple SA_{χ} and SSA_{κ} (space separation and redundancy) both before it reaches $\overline{\mathbf{DB}}$ and before it returns to $Cust_{\psi}$. As $Ticket_{\psi}$ is passed through this authentication chain, there is also an authentication method for messages passed between the SA_{χ} and SSA_{κ} to verify the identity of the sending SA_{χ} or SSA_{κ}. *Xchain Values* are used in these messages for inter-machine authentication. Details on how these values are calculated are included in Algorithm 4. The notation for these values is XV_A, $A \in [1, 6]$ as described in Algorithm 4.

Definition 30 : Given $Ticket_{\psi}$ which is sent by $Cust_{\psi}$ to the IDACS network, the $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ contained in $Ticket_{\psi}$ must be verified by an authentication chain of multiple SA_{χ} and SSA_{κ} before it is sent to DB_{γ} (as detailed in Algorithm 1 and Definition 29). The order in which SA_{χ} and SSA_{κ} verify $Ticket_{\psi}$ is pseudo-random, but calculated by the *F-box(next-SA-SSA)* transform. This transform accepts $Ticket_{\psi}$, the current location SA_{χ} or SSA_{κ}, and $\mathcal{S}$SA or $\mathcal{S}$SSA as inputs and outputs index χ or κ for the next-hop SA or SSA (next-hop $SA_{\chi} \in \overline{\mathbf{SA}}$, next-hop $SSA_{\kappa} \in \overline{\mathbf{SSA}}$). The transform applies F-box(Hash) to $Ticket_{\psi}$ and then calculates the Hamming distance between F-box(Hash, $Ticket_{\psi}$)and $PID(SA_{\chi})$ or $PID(SSA_{\kappa})$ for $\chi \in [1, q]$ and $\kappa \in [1, n]$. The index χ or κ where $PID(SA_{\chi})$ or $PID(SSA_{\kappa})$ has the lowest Hamming Distance (excluding all SA_{χ} or SSA_{κ} already in the authentication chain) is the index of the next-hop SA or SSA. The F-box(next-SA-SSA) is represented in equation notation by

$$\chi = \text{F-box}(\text{Next}, Ticket_{\psi}, SA_{\chi}, \mathcal{S}\overline{\text{SA}})$$

$PID(SA_{\chi})$ or $PID(SSA_{\kappa})$ are shared among all SA_{χ} or SSA_{κ}, but they are not shared with $Cust_{\psi}$. The F-box(next-SA-SSA) transform may only occur at SA or SSA locations.

Definition 31 : Given $Ticket_{\psi}$ and XV values, the complete messages passed between multiple SA_{χ} and SSA_{κ} are termed *Network Security Tickets*, denoted TK_A, $A \in [1, 5]$. The details are shown in Algorithm 4. TK_A is a concatenation of the relevant $Ticket_{\psi}$ and Xchain values.

Definition 32 : Given network message B, any SA_{χ} or SSA_{κ} processing B will record a *Security Ticket Log Record* \B\ detailing the vital information regarding B (e.g. the time B was processed, the IP address of the $Cust_{\psi}$ that sent B, etc.) B may be any $Ticket_{\psi}$ or TK network messages. The logs residing on SA_{χ} or SSA_{κ} are part of $\mathcal{S}SA_{\chi}$ or $\mathcal{S}SSA_{\kappa}$.

Definition 33 : Given Definition 32, any SA or SSA that processes a network message (e.g. TK) records a log record \TK\ using the *F-box(insert-log-record)* transform. This transform accepts $\mathcal{S}SA_{\chi}$ or $\mathcal{S}SSA_{\kappa}$ and TK as inputs and outputs an updated

version of $\mathcal{S}SA_{\chi}$ or $\mathcal{S}SSA_{\kappa}$ which contains \TK\. The F-box(insert-log-record) transform is represented in equation notation by

$$\mathcal{S}SA_{\chi} = \text{F-box}(\mathcal{I}\text{nsert}, \mathcal{S}SA_{\chi}, TK)$$

Definition 34 : Given Definition 33, any SA_{χ} or SSA_{κ} may search its own log entries for a given \TK\ that matches certain input parameters such as time, IP address of sending $Cust_{\psi}$, etc. These input parameters are not rigidly defined, and may exist in many combinations. The *F-box(retrieve-log-record)* transform accepts $\mathcal{S}SA_{\chi}$ or $\mathcal{S}SSA_{\kappa}$ and a list of conditions as inputs and outputs one or more matching log records \TK\, or 'null' if no matching records are found. This transform is represented in equation notation by

$$\backslash TK\backslash = \text{F-box}(\mathcal{R}\text{trv}, \mathcal{S}SA_{\chi}, \{\text{conditions}\})$$

3.2.4.2 IDACS Network-side Algorithms

The network-side authentication and authorization process described in Algorithm 1 is carried out in the "run_auth_chain()" function described in Algorithm 4.

The initial inputs to Algorithm 4 are handled separately depending on if this is the first call of the function (Algorithm 1 (8) with n=1) or a subsequent call. For the first call of the function, SA_1 has been randomly selected by $Cust_{\psi}$, $Ticket_{\psi}$ has been sent from $Cust_{\psi}$ to SA_1 (Algorithm 1 (5) connected to Algorithm 4 (2)), and XV_1 and XV_4 have been randomly generated by SA_1 and SSA_1, respectively (Algorithm 1 (6) connected to Algorithm 4 (1) and (7)). For subsequent function calls, SA_1 and SSA_1 in the current Algorithm 4 function call are SA_2 and SSA_2 from the previous Algorithm 4 function call, and $Ticket_{\psi}$ resides at SA_1 as a consequence; XV_1 and XV_4 in the current Algorithm 4 function call are XV_2 and XV_5 from the previous Algorithm 4 function call, respectively (Algorithm 1 (8) connected to Algorithm 4 (1) and (3)) (see Fig. 6 and Fig. 7 for details on how consecutive calls to Algorithm 4 are linked).

Algorithm 4. run_auth_chain()

input: SA_1, XV_1, XV_4, $Ticket_\psi$
output: XV_2, XV_5, SA_2, SSA_2, passed

At beginning of algorithm, the following values reside at the indicated locations after the last iteration of this algorithm, or are sent to ($Ticket_\psi$) or generated at (XV_1 and XV_4) the indicated locations during the first iteration of the algorithm

1 XV_1 ◊ SA_1
2 $Ticket_\psi$ ◊ SA_1
3 XV_4 ◊ SSA_1

at SA_1

4 SA_1 = F-box(Insert, SA_1, $Ticket_\psi$)
5 if (check_OTP_PID(SA_1, $Ticket_\psi$, $Cust_\psi$) == false) return (passed = false)
6 SA_2 = F-box(Next, $Ticket_\psi$, SA_1, $\overline{SA}$) SSA_1 = F-box(Next, $Ticket_\psi$, SA_1, $\overline{SSA}$)
7 XV_2 = F-box(Hash, XV_1, *Key*(SA_1, SA_2))

8 SA_1 → SA_2: TK_1 = { $Ticket_\psi$, XV_1, XV_2 }
9 SA_1 → SSA_1: TK_2 = { $Ticket_\psi$, XV_2 }

at SSA_1

10 SSA_1 = F-box(Insert, SSA_1, TK_2)
11 if (check_OTP_PID(SSA_1, $Ticket_\psi$ ◊ TK_2, $Cust_\psi$) == false) return (passed = false)
12 SA_2 = F-box(Next, $Ticket_\psi$ ◊ TK_2, SSA_1, $\overline{SA}$) SSA_2 = F-box(Next, $Ticket_\psi$ ◊ TK_2, SSA_1, $\overline{SSA}$)
13 XV_3 = F-box(Hash, XV_2, *Key*(SSA_1, SA_2)) XV_5 = F-box(Hash, XV_4, *Key*(SSA_1, SSA_2))

14 SSA_1 → SA_2: TK_3 = { $Ticket_\psi$, XV_3, XV_5 }
15 SSA_1 → SSA_2: TK_4 = { $Ticket_\psi$, XV_4, XV_5 }

at SA_2

16 SA_2 = F-box(Insert, SA_2, TK_1, TK_3)
17 if (XV_2 ◊ TK_1 != F-box(Hash, XV_1 ◊ TK_1, *Key*(SA_1, SA_2)) or (XV_3 ◊ TK_3 != F-box(Hash, XV_2 ◊ TK_1, *Key*(SSA_1, SA_2))
18 report_and_trace_attack() return (passed = false)
end
19 SSA_2 = F-box(Next, $Ticket_\psi$, SA_2, $\overline{SSA}$)
20 XV_6 = F-box(Hash, XV_5, *Key*(SA_2, SSA_2))

21 SA_2 → SSA_2: TK_5 = { $Ticket_\psi$, XV_6 }

at SSA_2

22 SSA_2 = F-box(Insert, SA_2, TK_4, TK_5)
23 if (XV_5 ◊ TK_4 != F-box(Hash, XV_4 ◊ TK_4, *Key*(SSA_1, SSA_2)) or (XV_6 ◊ TK_5 != F-box(Hash, XV_5 ◊ TK_4, *Key*(SA_2, SSA_2)
24 report_and_trace_attack() return (passed = false)
end

25 return (passed = true)

The main body of Algorithm 4 is represented graphically in Fig. 6. First, SA_1 records a security log record of $Ticket_\psi$ (4). Next, SA_1 verifies the associated OTP_χ and also $\overline{PID_\psi}$ extracted from $Ticket_\psi$; if the verification fails, then the function returns "false" (5). The next-hop SA_2 and SSA_1 are determined (6), and XV_2 is calculated (7). TK_1 is formed and sent to SA_2 (8), and TK_2 is formed and sent to SSA_1 (9). SSA_1 mirrors this process; SSA_1 records a security log record of TK_2 (10) and verifies

$\overline{\boldsymbol{PID_\psi}}$ extracted from $Ticket_\psi$ which is extracted from TK_2 (11). SSA_1 then calculates the next-hop SSA_2 and SA_2 (12) and also calculates XV_3 and XV_5 (13). SSA_1 then forms TK_3 and sends it to SA_2 (14) and forms TK_4 and sends it to SSA_2 (15).

While SA_1 and SSA_1 verify OTP_χ and $\overline{\boldsymbol{PID_\psi}}$, SA_2 and SSA_2 verify the Xchain values (however, SA_2 and SSA_2 will also be verifying OTP_χ and $\overline{\boldsymbol{PID_\psi}}$ as SA_1 and SSA_1 in the next function call of Algorithm 4). SA_2 first records security log records for TK_1 and TK_3 (16). Next, the relationship between XV_1 and XV_2 and also the relationship between XV_2 and XV_3 are verified (17). If the XV relationships fail verification, the authentication process is stopped (18). The next-hop SSA_2 (19) and XV_6 (20) are both calculated. Finally, TK_5 is formed and sent to SSA_2 (21). SSA_2 verifies its received Xchain values in a similar fashion (22–24). If all of the authentication checks and Xchain verifications pass, the function returns successfully (25).

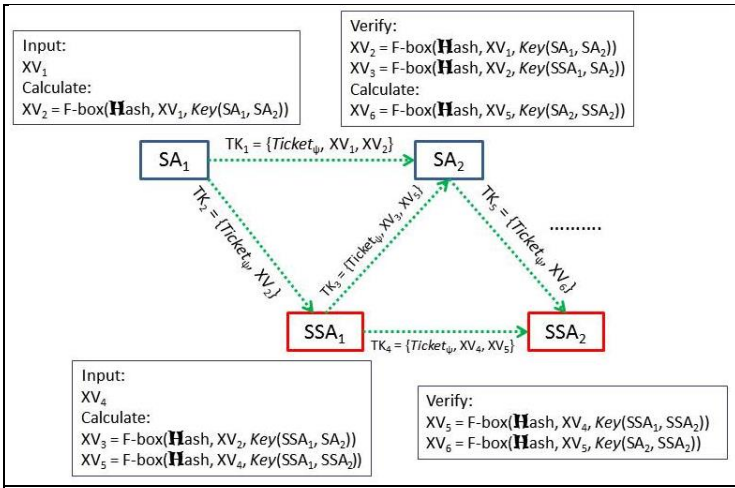


Fig. 6. Mutual authentication in authentication chain

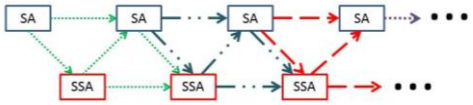


Fig. 7. Cascading the *run_auth_chain()* algorithm

Algorithm 5. check_OTP_PID()

input: location, $Ticket_\psi$, $Cust_\psi$
output: passed

1 if (location ϵ $\overline{\boldsymbol{SA}}$) and ($OTP_{location}$◊$Ticket_\psi$!= calculate_OTPs(location, $Cust_\psi$))
2 report_and_trace_attack() return (passed = false)
end
3 if($\overline{\boldsymbol{PID_\psi}}$ ◊ $Ticket_\psi$!= calculate_PIDs(location, $Cust_\psi$))
4 report_and_trace_attack() return (passed = false)
end

5 return (passed = true)

Algorithm 5 outlines the procedure used by SAs and SSAs to verify the $\overline{\boldsymbol{OTP_\psi}}$ and $\overline{\boldsymbol{PID_\psi}}$ contained in $Ticket_\psi$. The OTP_χ associated with SA_χ is verified by each SA_χ (1), and the entire $\overline{\boldsymbol{PID_\psi}}$ is verified by each SA_χ and SSA_ε (3) (excluding Content(PID_ε), which is only checked by SSA_κ). If either of these checks fail, the "report_and_trace_attack()" function is called to

identify the source of error (which is assumed to be an attacker with incomplete authentication credentials). The details of this algorithm are discussed in "4.4 Attack Traceback".

The following properties help to explain the operation and purpose of the Xchain values.

Property 5 : The procedure outlined in Algorithm 4 provides mutually-supported authentication between the SAs and SSAs authenticating $Ticket_{\psi}$. Fig. 6 graphically illustrates the XV portion of Algorithm 4 *run_auth_chain()*. During the first iteration of *run_auth_chain()*, XV_1 and XV_4 are randomly generated; in subsequent iterations, they are given the values of XV_2 and XV_5 from the previous iteration. SA_1 calculates XV_2 by hashing XV_1 with $Key(SA_1, SA_2)$, and sends both values to SA_2. SA_2 is able to verify XV_2 using its own copy of $Key(SA_1, SA_2)$, which verifies the identity of SA_1. SA_1 also sends XV_2 to SSA_1, which calculates XV_3 by hashing XV_2 with $Key(SSA_1, SA_2)$ and sends it to SA_2. SA_2 is able to verify the SA_1-SSA_1 connection as well as the identity of SSA_1 by verifying XV_3. Similarly, SSA_1 calculates XV_5 by hashing XV_4 with $Key(SSA_1, SSA_2)$ and sends both XV_4 and XV_5 to SSA_2 but only XV_5 to SA_2. SA_2 calculates XV_6 by hashing XV_5 with $Key(SA_2, SSA_2)$ and sends it to SSA_2. SSA_2 is then able to verify the identity of SSA_1 by verifying XV_5 and is able to verify the SSA_1-SA_2 link as well as the identity of SA_2 by verifying XV_6. The *run_auth_chain()* algorithm may be cascaded across *N* SAs and SSAs to accomplish the desired number of authentications (Fig. 7). In this way, the authentication chain provides mutually-supported space-separated authentication that is time-evolving between the SAs and SSAs.

Property 6: When the *run_auth_chain()* algorithm is cascaded, XV_2 and XV_5 of one iteration are, in fact, the XV_1 and XV_4, respectively, for the next iteration; cascaded iterations of the *run_auth_chain()* algorithm are seamlessly integrated (Fig. 7). This is demonstrated at (8) in Algorithm 1. In this way, consecutive iterations provide mutually-connected authentication for each other.

3.3 Proofs for Theorem 1 and Theorem 2

3.3.1 Proofs

Consider the following scenario: an attacker wishes to replicate the IDACS Network Access Control procedure for a legitimate $Cust_{\psi}$ in order to impersonate $Cust_{\psi}$ and gain access to $Cust_{\psi}$'s data residing on DB_{γ}. In order to impersonate $Cust_{\psi}$, the attacker requires correctly generated $\overline{\boldsymbol{OTP}}_{\boldsymbol{\psi}}$ and $\overline{\boldsymbol{PID}}_{\boldsymbol{\varepsilon}}$ for $Cust_{\psi}$. To accomplish this, the attacker requires two things: a) the cryptographic seeds residing on/derived from $Client_{\rho}$, $Badge_{\zeta}$, PIN_{λ}, and Pwd_{θ} (i.e. $\overline{\boldsymbol{Seed}}_{\sigma}\Diamond Client_{\rho}$, $\overline{\boldsymbol{Seed}}_{\sigma}\Diamond Badge_{\zeta}$, $\overline{\boldsymbol{Seed}}_{\sigma}\Diamond Pwd_{\theta}$, and $\overline{\boldsymbol{Seed}}_{\sigma}\Diamond PIN_{\lambda}$) and b) the order in which these seeds must be hashed to generate all OTP_{χ} and PID_{ε} (i.e., the output of the F-box(Lookup) transform in Algorithm 2 (3) and Algorithm 3 (3)). An attacker who is able to steal or clone a $Client_{\rho}$,

$Badge_\zeta$, Pwd_θ, or PIN_λ can gain access to (a) through memory scraping or other memory access strategies. However, in order to obtain (b), the attacker needs access to $\mathcal{S}Cust_\psi$, which is the critical input to the F-box(Lookup) transform. Now, $\mathcal{S}Cust_\psi$ is composed of $\mathcal{S}Client_\rho$, $\mathcal{S}Badge_\zeta$, $\mathcal{S}Pwd_\theta$, and $\mathcal{S}PIN_\lambda$; the entire $\mathcal{S}Cust_\psi$ does not reside with any one of these locations or states. Therefore, an attacker who does not possess all four of these items cannot recreate $\mathcal{S}Cust_\psi$ apart from a brute-force attack; such an attacker must resort to other means to recreate (b) (i.e. the output of the F-box(Lookup) transform).

The attacker faces an order-reassembly problem; this problem can be represented using graph theory. The group of seeds $\overline{\boldsymbol{Seed}}_\sigma$ is represented as a set of vertices $\mathbf{V}$ and a set of directed edges $\mathbf{E}$, where each $v\epsilon\mathbf{V}$ is connected to every other $v\epsilon\mathbf{V}$ by a pair of directed edges $e\epsilon\mathbf{E}$ with opposite directions. Each $e\epsilon\mathbf{E}$ is given an associated weight $\mathbf{W}(e)$, $0\leq\mathbf{W}(e)\leq1$ (Fig. 8 (a)). Each $e\epsilon\mathbf{E}$ with a tail connecting to v_1 and a head connecting to v_2 ($v_1,v_2\,\epsilon\mathbf{V}$), represents the possibility that v_2 follows v_1 in the output of the F-box(Lookup) transform, while $\mathbf{W}(e)$ represents the associated probability (the determination of $\mathbf{W}(e)$ is discussed in the following section). Presumably, the path connecting N vertices (where N is known to be the number of seeds output by the F-box(Lookup) transform) that has the highest sum $\mathbf{W}(e)$ of any path with N vertices will be the correct solution to the F-box(Lookup) transform (Fig. 8 (b)). The problem of finding the highest sum $\mathbf{W}(e)$ path is defined as the Maximum Weight Directed Path of Specified Length (MWDPSL) problem; this problem is proven NP-complete in "Appendix A". The inspiration for this proof is drawn from [29]. The NP-complete proof of the (MWDPSL) problem in turn proves Theorem 1.

Consider a second scenario. The same attacker does not have access to $Client_\rho$, $Badge_\zeta$, Pwd_θ, or PIN_λ (and therefore not $\mathcal{S}Cust_\psi$ either). This attacker must recreate $\mathcal{S}Client_\rho$, $\mathcal{S}Badge_\zeta$, $\mathcal{S}Pwd_\theta$, and $\mathcal{S}PIN_\lambda$ in order to gain access to both (a) and (b); therefore, the attacker must correctly reassemble the memory contents of $Client_\rho$ and $Badge_\zeta$ and an analagous representation of Pwd_θ, or PIN_λ (these memory contents are the definition of $\mathcal{S}Client_\rho$, $\mathcal{S}Badge_\zeta$, $\mathcal{S}Pwd_\theta$, and $\mathcal{S}PIN_\lambda$). Each of these items is represented by b memory locations, each of which is Σ bits long; therefore, each memory location contains one of 2^Σ possible values. This situation can be represented using an undirected "colored" graph [30]. The possible values for a given memory location can be represented by a group of 2^Σ vertices v of the same "color", $v\epsilon\mathbf{V}$, and each memory location can be represented by a different "color" group (represented as different shapes in Fig. 9). Each $v\epsilon\mathbf{V}$ is connected to every other $v\epsilon\mathbf{V}$ (except for v of the same "color") by an undirected edge $e\epsilon\mathbf{E}$, and each $e\epsilon\mathbf{E}$ has an associated weight $\mathbf{W}(e)$, $0\leq\mathbf{W}(e)\leq1$. Each edge represents the possibility that the two connected v are both present in the correct reconstruction of the memory contents, and the associated $\mathbf{W}(e)$ represents the probability (again, the manner in which $\mathbf{W}(e)$ is assigned is discussed in the next section). A path connecting one v of each "color" that has the highest sum $\mathbf{W}(e)$ will represent the correct reconstruction of the memory

contents (Fig. 9). This problem is defined as the Maximum Weight Path of Specified Length (MWPSL) problem; this problem is also proved NP-complete in "Appendix A". In turn, the proof of the MWPSL problem proves Theorem 2.

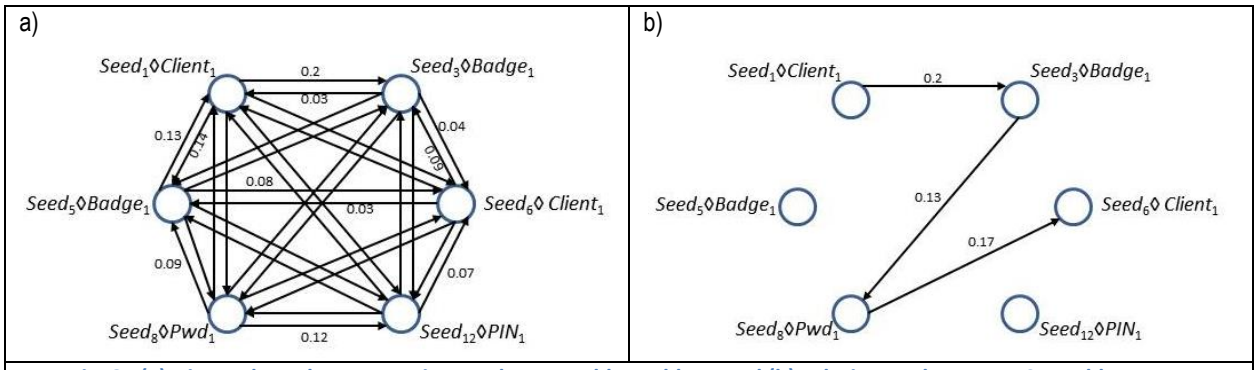


Fig. 8. (a) Directed graph representing seed reassembly problem, and (b) solution to the MWDPSL problem, N=4

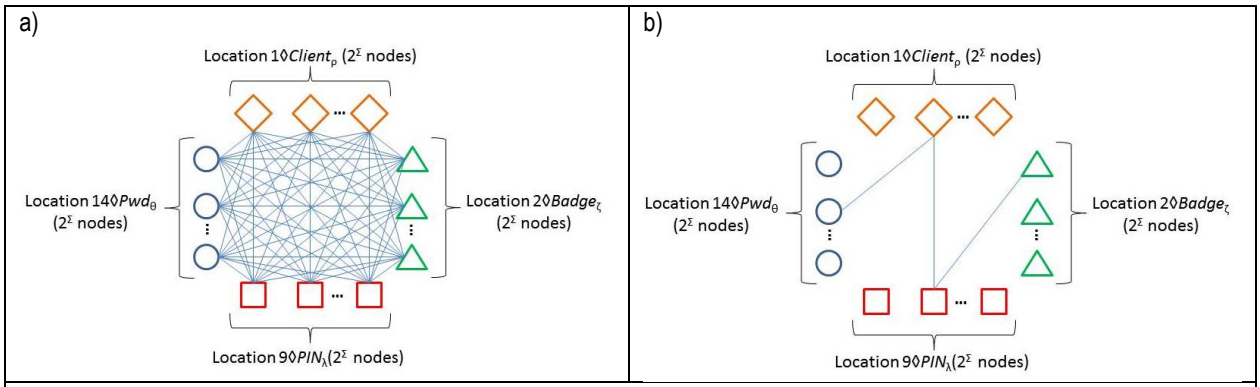


Fig. 9. (a) Graph representing memory reassembly problem, *b* = 4 (b) solution to the MWPSL problem, N = 4 (W(e) not shown)

3.3.2 Constraints on NP-completeness

The NP-completeness put forth in Theorem 1 and Theorem 2 points to a high level of security for IDACS, since NP-completeness is associated with an exponential increase in the problem solution complexity. However, NP-completeness speaks only to the worst-case (for the attacker) situation. It may be that the problem soultion can be found with significantly less than exponential complexity. Consider Fig. 8; the difficulty in finding the MWDPSL lies in the fact that the maximum weight path is not immediately apparent. If the graph in Fig. 8 contained a few very high-weight edges and the remaining edges had lower weights, this would provide a significant portion of the solution and greatly reduce the problem complexity. In the paper which originally presented the graph method for matching fragemnts of data [32], the authors discuss how this method can be used to reassemble scattered data file fragments. They note in their work that fragments from highly-patterned data files generate graphs with a few higher weight edges, resulting in significantly reduced probelm complexity, while data with more pseudo-random qualities generated graphs with more uniformly weighted graphs, resulting in higher problem complexity. This leads to the important question, what would a graph generated by a realistic IDACS situation look like?

According to the research in [32], data fragments (whether file fragments in that paper or $\overline{\boldsymbol{Seed}}$ in this paper) that are more "random" will have a low correlation with each other; when analyzed for likelihood of matching, they will result in a uniform distirbution of edge weights (Fig. 10). A strong Random Number Generator would be expected to generate this type of result. However, if the data fragments are highly patterned (e.g. the outputs of a poorly-designed hash function), the analysis will result in a graph with a few high-weight edges (Fig. 11). So which of these two situations most closely matches the analysis of the IDACS $\overline{\boldsymbol{Seed}}$?

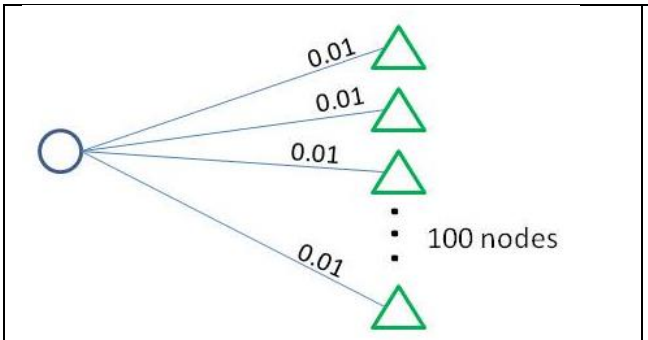


Fig. 10. Graph problem for uniform probability relationship (e.g. true Random Number Generator)

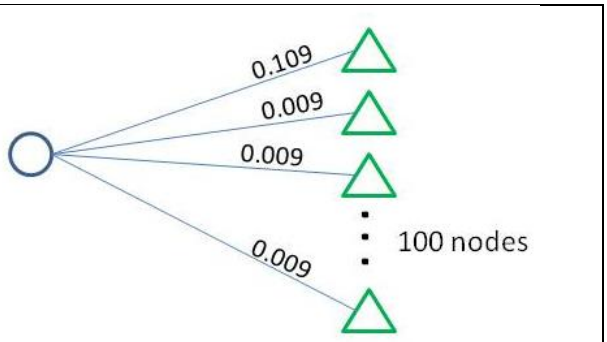


Fig. 11. Graph problem for highly correlated relationship (e.g. poorly designed hash function)

The National Institute of Standards and Technology (NIST) has provided a battery of tests [34] that analyze the outputs of Random Number Generators (RNGs) to measure their "randomness" by looking for patterns [35]. This battery of tests has also been used on ciphertext from various encryption algorithms to measure how closely it matches truly random data. This battery contains 15 individual tests, each of which measures different aspects of "randomness" in a set of data. Each test, when analyzing a data sample, asks this question: "If the algorithm that generated this data sample was truly random, what is the probability that this specific data sample could have been generated?" The test responds with a p-score for the analyzed data sample; this p-score is a probability in the range [0, 1]. NIST recommends interpreting these p-scores using a "significance level" of 0.01; if a data sample's p-score is above 0.01, then the data sample has passed the randomness test. Now, some data samples that are truly random will generate a failing p-score, which would be a "false negative" for randomness; this is due to inherent weaknesses in the tests [35].There are two ways to interpret the results of these tests. The first way is to look at the proportion, or percentage, of data samples with passing p-scores. According to the parameters in [35], for a set of tests run with 1000 data samples, a truly random RNG will have a minimal proportion of 0.9805068, i.e. a minimum 98.05% pass rate. The second way is to look at the distribution of the test p-scores. For a set of truly random data samples being subjected to a test, it is expected that the p-scores of the data samples should be evenly distributed. Evenness of distribution can be measured by

calculating P-value$_T$ based on the chi-square statistic for each test as discussed in [35]; if each test has P-value$_T$ ≥ 0.0001, then the p-scores are considered to be evenly distributed.

In order to determine the "randomness" of $\overline{Seed}$ that will be used in IDACS, the NIST battery of tests was applied to a number of SHA-256 cryptographic hash outputs designed to simulate the $\overline{Seed}$ that will be used by IDACS. The battery of tests was applied to 1000 data samples of sizes dictated by the NIST battery.

Of the 15 tests in the battery, two of the tests are run twice during the course of the battery. Results for both tests are reported here. Three of the tests are run a number of times; results for two randomly chosen instances of those tests are reported here. All other tests were run once, and the results are reported here. There are a total of 20 separate test results.

For the first analysis, the proportions of data samples that pass each test are presented in graph form in Fig. 12. It can be seen that all tests exceed the minimum pass proportion of 0.9805068.

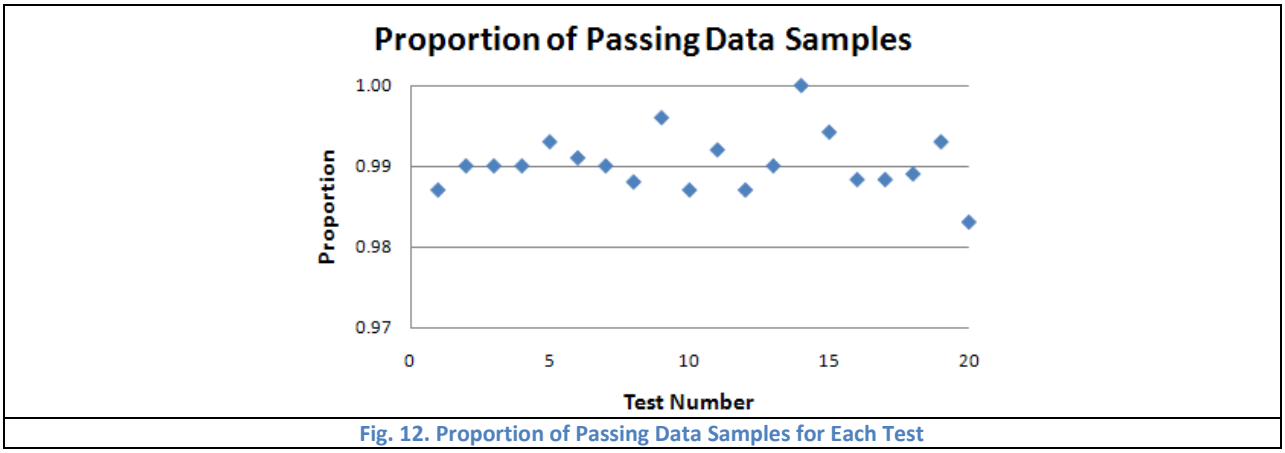


Fig. 12. Proportion of Passing Data Samples for Each Test

For the second analysis, the P-valueT for each of the tests is presented in Table 3. It can be seen that all tests exceed the minimum pass value of 0.0001.

Table 3. P-value$_T$ for each test

Test #	P-value$_T$		Test #	P-value$_T$		Test #	P-value$_T$
1	0.461612		8	0.378705		15	0.907498
2	0.328297		9	0.572847		16	0.345744
3	0.134944		10	0.873987		17	0.915241
4	0.788728		11	0.581082		18	0.078567
5	0.918317		12	0.444691		19	0.278461
6	0.605916		13	0.455937		20	0.614226
7	0.018668		14	0.052531			

3.3.3 Implications for IDACS

What are the implications of the previous sections for IDACS? Consider the attacker in the scenarios; an attacker who has access to cryptographic seeds but not $\mathcal{S}Cust_\psi$ (Theorem 1) OR no access to any memory locations at all (Theorem 2) faces the NP-complete reassembly problem. There is no known solution to these problems with a complexity polynomial to the

problem size (number of seeds or memory locations in the graph). A polynomial-time solution could exist for certain situations meeting special constraints; however, due to the demonstrated randomness of the $\overline{\boldsymbol{Seed}}$ used in IDACS, it is expected that the best algorithm will be of exponential complexity to the problem size. Having no special algorithms to aid him, and the attacker will be reduced to brute-force attacks. For a Theorem 1 situation, he must try every possible seed combination solution to F-box(Lookup) to guess the solution as shown in Fig. 8 (b). For a Theorem 2 situation, he must try every possible memory value to guess the solution as shown in Fig. 9 (b). Such attacks will be detected quickly, and the security log/forensics capabilities of IDACS will allow the system to identify which seeds, locations, and states have been compromised by the attacker. The attacker will be foiled even if the some (but not all) of {$Client_{\rho}$, $Badge_{\zeta}$, PIN_{λ}, Pwd_{θ}} are stolen. Furthermore, even if the attacker is able to guess the solution, because of Property 3, the identity, memory locations, and order of the cryptographic seeds evolve in time, presenting the attacker with totally new problems as shown in Fig. 8 (a) and Fig. 9 (a).

To get an idea of the real-world implications of this, consider a Theorem 1 situation where the attacker has access to all of the $Seed_{\sigma}$ needed to calculate $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$ (but without access to the F-box(Lookup) transform). For an IDACS system with a given q (number of SA_X, with the same number of OTP_X to be calculated) and a given r (number of PID_{ε} to be calculated) and a given number of $Seed_{\sigma}$ used to calculate each OTP_X and PID_{ε}, Table 4 shows how long it would take the attacker (on average), trying all possible permutations of $Seed_{\sigma}$ at 10^6 permutations per second, to find the ordering to correctly calculate $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$. Since Theorem 1 deals with the ordering of known $Seed_{\sigma}$ and Theorem 2 deals with generating these $Seed_{\sigma}$ (before ordering can even occur), it is accepted that the computation times for a Theorem 2 situation would be significantly greater than those shown in Table 4.

Table 4. Average time to for permutation orderings of $Seed_{\sigma}$ to generate correct OTPs and PIDs at 10^6 permutations per second

		# $Seed_{\sigma}$ per OTP/PID		
		8	12	16
# OTPs (SAs) + # of PIDs	6 + 8	$3.13 * 10^{174}$ years	$4.01 * 10^{294}$ years	$8.87 * 10^{422}$ years
	8 + 8	$6.11 * 10^{207}$ years	$5.63 * 10^{348}$ years	$1.36 * 10^{499}$ years
	10 + 8	$8.80 * 10^{241}$ years	$1.59 * 10^{404}$ years	$1.14 * 10^{577}$ years

3.4 Simulations

3.4.1 Simulation Setup

Simulations for this research were carried out using a model of an IDACS network built using MATLAB. The simulation network was built as demonstrated in Fig. 13. The network contains a variable number of Clients up to a maximum of 102,400. The SA barrier consisted of four SAs (with the possibility for future expansion). The network contained one or two SSAs and one

Database (both with the possibility for future expansion). Network links were built according to the bandwidths indicated in Fig. 13, and were full-duplex.

During all simulations, background traffic was introduced into the network in order to simulate normal operating conditions. It was determined that introducing network traffic on the slower network connections did not affect the simulation results (but made the simulation running time prohibitively long). Therefore, all background traffic was introduced between the SAs and the SSAs. Uniformly distributed background traffic equal to 80 Kbps/Client was divided equally between the SAs and sent from each SA to each SSA. An equal amount of traffic was also sent from each SSA to each SA. This rate of background traffic ranged from a one-way 80% load on a 10 Gbps link for a full-sized network (102,400 Clients) to a much smaller load for smaller networks (0.8% load for 1000 Clients). This rate of background traffic affected both SA/SSA security log size (which has a substantial effect on real-time forensics, which is discussed in "4.5.1 Attack Traceback Time Simulations") and packet transit time in the datacenter (due to network congestion). Additionally, realistic packet delay times for routers were obtained from router manufacturer documentation and incorporated into the simulation. Packet processing delays for Clients, SAs, SSAs, and Databases were estimated; when the IDACS prototype implementation is completed by the researchers, more precise packet processing times will be measured and incorporated into the simulation.

Each simulation consisted of two phases. In the first phase, each Attacker would build a set of compromised Slaves (a botnet) gathered from a pool of vulnerable Clients. The attacker would compromise the Clients (turning them into bots) by sending a Compromise packet to each Slave candidate. During the second phase, each Attacker would send out a specified

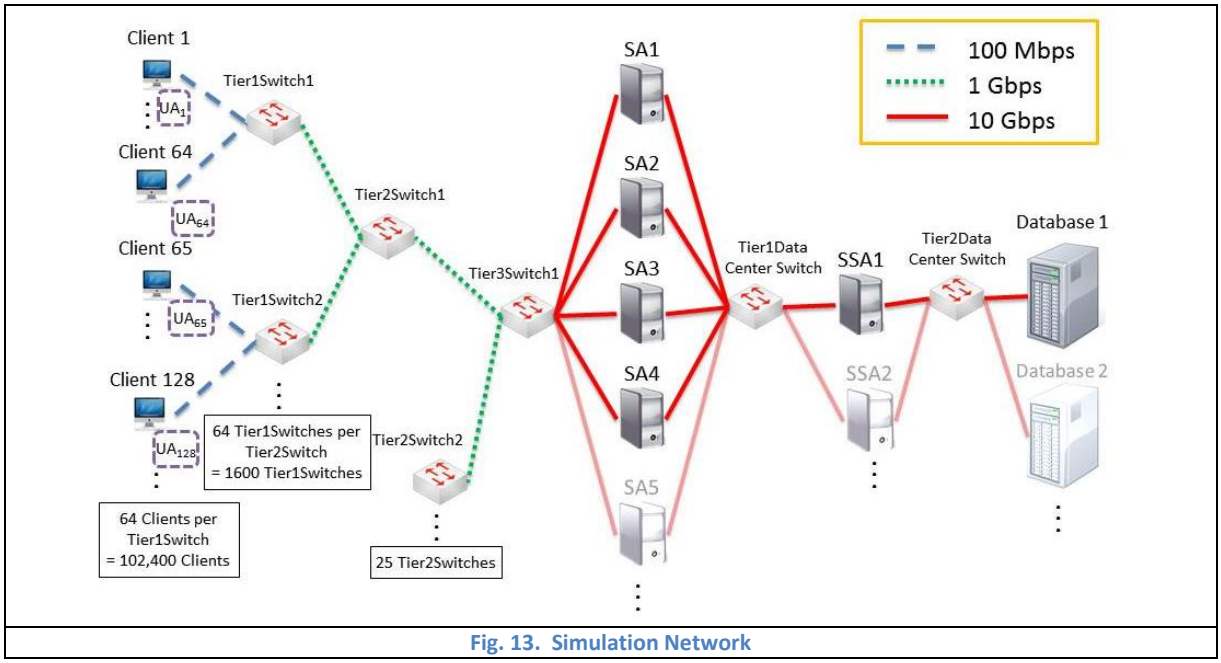


Fig. 13. Simulation Network

number of Read and Write attacks using a random-length Attack Chain of chained Slaves (the details of the attack scenarios used in this simulation are discussed in "4.2 Attack Vectors"). The start times for these attacks were uniformly distributed over a 20 millisecond period. These attack packets would be checked for $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$ by the SAs and the SSAs according to normal IDACS operations. If the packet failed any of these checks, the packet would be dropped as an attack. If an attack packet was able to bypass all of the security checks and successfully carry out a Database Read or Write operation, the attack was considered successful. Background traffic was present in the network during both of these phases.

3.4.2 Simulation Parameters

In the simulation, if the attack packet did not possess the proper $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$, any SA or SSA would detect the attack 100% of the time. In reality, however, some attacks would go undetected due to various attack methods (zero-day attacks, SQL injection, buffer overflow, etc.) Therefore, SAs and SSAs in the simulation were classified as "fully compromised" and "partially compromised". A "fully compromised" SA or SSA would pass a failed $\overline{\boldsymbol{OTP}_{\psi}}$ or $\overline{\boldsymbol{PID}_{\psi}}$ check as successful 100% of the time; this situation represents an SA or SSA that is fully controlled by an attacker. A "partially compromised" SA or SSA would pass a failed $\overline{\boldsymbol{OTP}_{\psi}}$ or $\overline{\boldsymbol{PID}_{\psi}}$ check 50% of the time; this represents a "normal", or uncontrolled by an attacker, SA or SSA. The rationale behind setting a "partially compromised" SA or SSA to a 50% fail rate is twofold. First it simulates zero-day attacks, etc.; second, it demonstrates the strength of the IDACS system, even under "poor" conditions and makes more visible the effect of other variables on network performance. In a realistic situation, it is expected that the failure rate of "normal" machines will be much less than 50%.

Additional probability variables were also used to govern other factors in the simulation. During chained attacks (in which an Attacker uses a chain of Slaves (bots) to launch an attack; this is discussed in "4.2 Attack Vectors"), the attacker was given an 80% probability of stealing the cryptographic seeds needed to calculate $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$and an 80% chance of an attack chain packet (prior to the final leg of the chained attack) passing a failed permissions check based on Content(PID_{ε}).

3.4.3 Attack Detect Rate

All tests were based on 1000 attacks for a given test case; 500 Read attacks and 500 Write attacks. Some tests used 1 SSA, and some used 2 SSAs. The first set of tests (Fig. 14) demonstrates the performance of the IDACS system under casualties ("fully compromised" SAs). It shows that even with multiple SAs compromised, the attack detect ratio is still very high. 1 SSA was used in these tests.

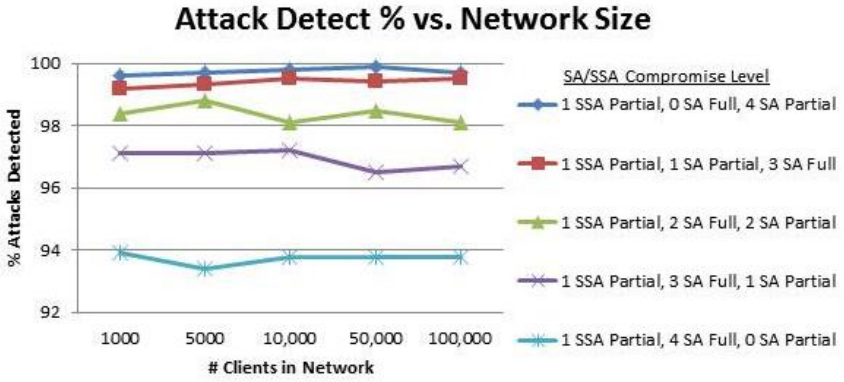


Fig. 14. Attack Detect Ratio vs. Network Size and Number of SAs "fully compromised"

As expected, Fig. 14 shows that the Attack Detection Ratio is fairly constant across network sizes. However, the Attack Detection Ratio is affected by the number of "fully compromised" SAs. When no SAs are "fully compromised", the system performs very well, with an average detection ratio above 99.5%. With one or two SAs "fully compromised", the detection ratio is still fairly high. Thus, it can be seen that the system provides an excellent defense against attacks, even under heavy casualties.

The second set of tests (Fig. 15) was performed to test the system under SSA "full compromise". The simulated network in these tests contained two SSAs; one was "fully compromised" and the other was "partially compromised". By comparing Fig. 14 and Fig. 15, it can be easily seen that the compromise of an SSA has greater effect on the Attack Detection Ratio than the compromise of an SA. This is because the simulation specified the chances of an attacker obtaining $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$ for a Slave were fairly high (80% chance), thus making permissions checks based on the Content(PID_{ϵ}) (which were only performed at SSAs) the primary mode of detecting attacks. Thus, the loss of an SSA has a greater effect on system security. However, even with an SSA and up to 2 SAs "fully compromised", the Attack Detect Ratio was still above 94%. This test shows that protecting the SSAs should be a top priority in any implementation of this system.

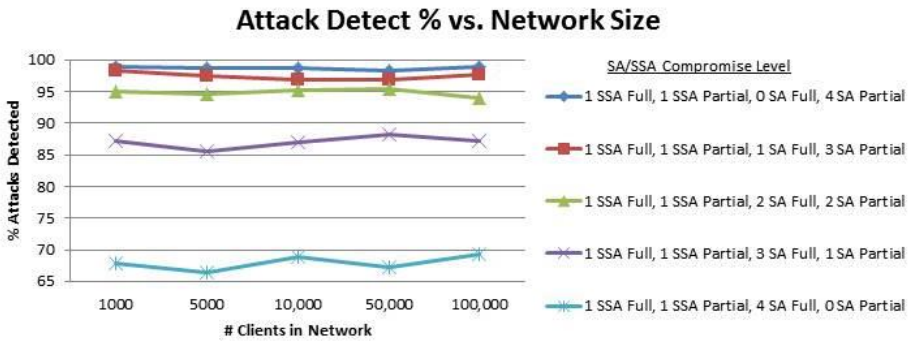


Fig. 15. Attack Detect Ratio vs. Network Size and Number of SA/SSAs "fully compromised"

One of the main features of IDACS is the real-time forensics capability. Through log examination and correlation, IDACS is able to trace back and correctly identify the origin of an attack, whether the attack is launched directly by the attacker or

indirectly using a botnet of legitimate IDACS users. This simulation also addressed these capabilities; the results will be presented and discussed in “4.5.1 Attack Traceback Time Simulations”.

4 IDACS Network Attack Detection, Prevention, and Traceback

4.1 Introduction

In today's network security environment, it is critically important to detect and prevent network intrusions. However, it is almost equally important to trace network attacks to their origins and identify the culprits and their methods. This allows the guilty parties to be held liable for their actions; it also allows network administrators to focus their resources once they know the weak spots in their defenses. IDACS provides real-time digital forensics capabilities that can identify network attackers as well as their collaborators, and even traitors within IDACS itself. This section will detail those capabilities possessed by IDACS and discuss how they can be used to detect, block, and trace attacks to their origins. Additionally, simulations will demonstrate the ability of IDACS to detect attacks and self-heal even when the network contains a high percentage of insider traitors.

4.2 Attack Vectors

When an attacker wishes to defeat the IDACS Network Access Control Protocol in order to gain access to protected data or services residing within the IDACS datacenter, there are several general attack vectors available for this task.

1) Forge legitimate $Cust_\psi$ credentials ($Client_\rho$, $Badge_\zeta$, Pwd_θ, and PIN_λ) in order to impersonate a legitimate $Cust_\psi$
2) Steal/hack credentials for a legitimate $Cust_\psi$
3) Hack and gain control over one or more SAs and/or SSAs in order to manipulate the authentication process

Attack vector 1) requires brute-force guessing of $\mathcal{S}Client_\rho$, $\mathcal{S}Badge_\zeta$, $\mathcal{S}Pwd_\theta$, and $\mathcal{S}PIN_\lambda$; this is infeasible according to Theorem 2. Attack vector 2) may be more effective, although the space-separation of $Client_\rho$, $Badge_\zeta$, Pwd_θ, and PIN_λ makes it more difficult for an attacker to collect them all and acquire a botnet of complete $Cust_\psi$. By using attack vector 3), an attacker can use a botnet of SAs and SSAs to bypass $\overline{\boldsymbol{OTP_\psi}}$ and $\overline{\boldsymbol{PID_\psi}}$ checks and even manipulate the correct authentication chain path. Attack vector 3) can be accomplished by hacking ***loyal*** SAs and SSAs, turning them into ***traitor*** machines; these traitor machines possess all of the $Seed_\sigma$ used by that particular IDACS machine. Additionally, it may be possible to use a hostile machine to impersonate, or ***spoof***, legitimate SAs and SSAs, although a spoofed machine would not possess the $Seed_\sigma$ associated with the legitimate machine. The most effective attack scenario combines vectors 2) and 3) in an attempt to access the IDACS datacenter.

If an attacker is able to use attack vector 1) possibly combined with 3) to control a single $Cust_\psi$, he will most likely attempt to access the IDACS datacenter directly (Fig. 16). However, if the attacker uses 2) to build a botnet of traitor $Cust_\psi$, he may choose to send $Ticket_\psi$ through multiple traitor $Cust_\psi$ (Fig. 17). In this way, the attacker is able to accomplish several

objectives. First, the attacker takes advantage of the credentials owned by the traitor $Cust_\psi$ in order to send a legitimate $Ticket_\psi$. Second, in the case that the attack is detected, he masks his identity from the IDACS forensics suite. However, even in this situation, once an attack is detected, the IDACS real-time forensics will be able to identify the attacker through the methods described in "4.4 Attack Traceback".

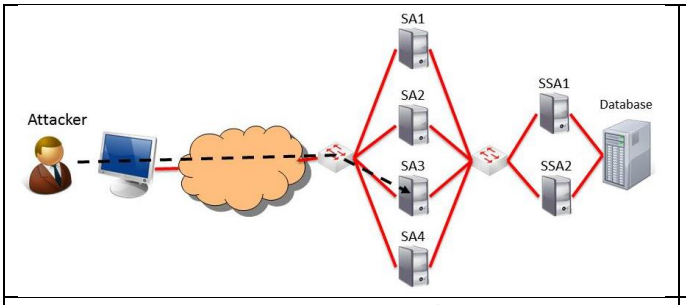


Fig. 16. Direct Attack

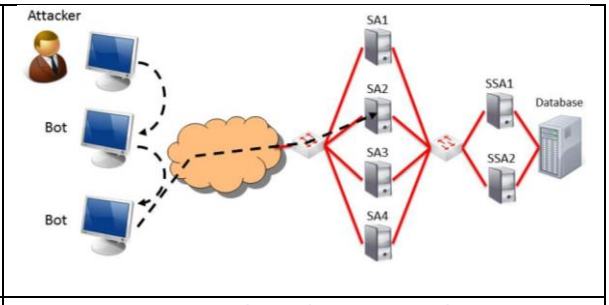


Fig. 17. Chained Botnet Attack

By means of attack vectors 2) and 3), any $Cust_\psi$, SA_χ, or SSA_κ can be turned into a traitor machine. When this happens, the machine becomes a Byzantine actor (i.e. a malicious system actor that actively works to defeat the correct operation of the system). A great deal of research has been done on the subject of Byzantine actors [34] [35] [36], and it is of great interest to be able to prove that a given system is Byzantine-resistant, able to operate correctly in the presence of a given number of Byzantine actors.

4.3 Attack Detection and Prevention

The incorporation of the space-separated time-evolving relationship into the IDACS Network is based on a simple principle, which affects its real-time forensics capabilities:

Principle 1: Any $Cust_\psi$, SA_χ, or SSA_κ in IDACS can be hacked and turned into a traitor/Byzantine actor. Any Customer ($Cust_\psi$), authenticating machine (SA_χ or SSA_κ), or real-time forensics machine (SSA_κ) can be turned into a traitor/Byzantine actor.

This principle is the reason for the decentralized approach of separating authentication capabilities in space and time. With a design that keeps this principle in mind, IDACS is able to detect and prevent almost all illegal $Ticket_\psi$ that are passed to it. In fact, IDACS is demonstrably secure against any illegal $Ticket_\psi$ under certain conditions. The following Claims are based on a certain set of assumptions outlined below, which enforce the parameters of network communication.

Assumption 1: Any $Cust_\psi$ can only communicate with $\overline{\boldsymbol{SA}}$.

Assumption 2: Any member of $\overline{\boldsymbol{SA}}$ can communicate with any $Cust_\psi$, any member of $\overline{\boldsymbol{SSA}}$, and any other member of $\overline{\boldsymbol{SA}}$.

Assumption 3: Any member of $\overline{\boldsymbol{SSA}}$ can communicate with any member of $\overline{\boldsymbol{SA}}$ or $\overline{\boldsymbol{DB}}$, and any other member of $\overline{\boldsymbol{SSA}}$.

Assumption 4: Any member of $\overline{\boldsymbol{DB}}$ can communicate with any member of $\overline{\boldsymbol{SSA}}$.

Assumption 5: An attacker who is forming $Ticket_\psi$ has access to all $Seed_\sigma$ stored on traitor SA_χ or SSA_κ.

Assumption 6: A spoofed SA_χ or SSA_κ does not have access to the $Seed_\sigma$ stored on the machine it is spoofing.

Assumption 7: Any DB_γ that receives a $Ticket_\psi$ can verify whether or not the SSA_κ that sent it was the correct SSA_κ at the end of the calculated authentication chain.

Assumption 8: When processing $Ticket_\psi$, IDACS performs $\overline{\boldsymbol{OTP_\psi}}$ or $\overline{\boldsymbol{PID_\psi}}$ checks on both the approach from $Cust_\psi$ to $\overline{\boldsymbol{DB}}$, and on the return from $\overline{\boldsymbol{DB}}$ to $Cust_\psi$. However, the authentication chain path for the return is based on the reply message $Ticket'_\psi$, which is different from $Ticket_\psi$.

Assumption 9: Any attack $Ticket_\psi$ falls into one of two categories: a) contains incorrect $\overline{\boldsymbol{OTP_\psi}}$ or $\overline{\boldsymbol{PID_\psi}}$, or b) contains correct $\overline{\boldsymbol{OTP_\psi}}$ and $\overline{\boldsymbol{PID_\psi}}$, but is attempting to access data or service that the originating $Cust_\psi$ does not have permissions to access.

Based on these assumptions, certain Claims about the attack detection and prevention capability of IDACS can be made. Recall that N is the Authentication Chain Length; there are N SAs and N SSAs in the approach authentication chain and N SAs and N SSAs in the return authentication path.

Claim 1: A $Ticket_\psi$ with incorrect $\overline{\boldsymbol{OTP_\psi}}$ will be detected with up to $2N$ traitor SSAs and $(2N - 1)$ traitor SAs in the approach and return authentication chain paths if the authentication chain path is not manipulated.

Justification for Claim 1: According to Assumption 5, if any SA_χ or SSA_κ is a traitor, then the attacker will have access to the $Seed_\sigma$ necessary to calculate $\overline{\boldsymbol{PID_\psi}}$ correctly. Thus, $\overline{\boldsymbol{PID_\psi}}$ checks will pass at each SA_χ or SSA_κ even if they are not traitors. However, if there is even one loyal SA_χ in the authentication chain, the attacker does not have access to the $Seed_\sigma$ needed to calculate OTP_χ. This incorrect OTP_χ will be detected by the loyal SA_χ, and the attack will be detected and prevented (Fig. 18).

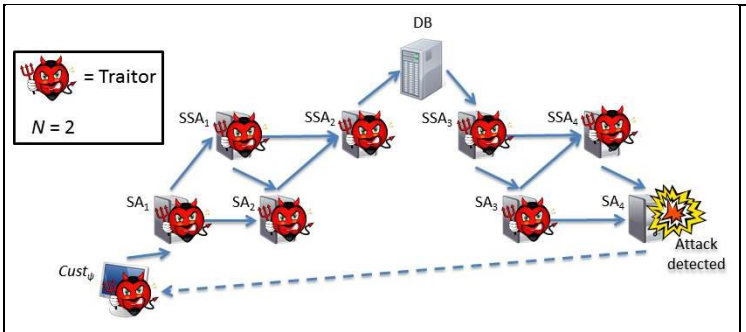


Fig. 18. Illustration of Claim 1

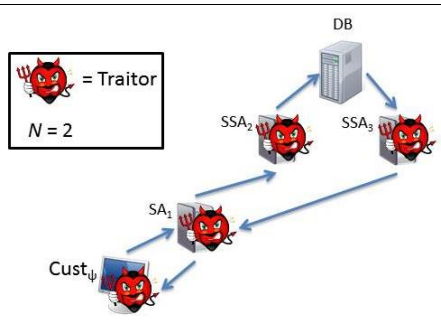


Fig. 19. Illustration of Claim 2

However, the strength of IDACS illustrated by Claim 1 is qualified by Claim 2.

Claim 2: A $Ticket_{\psi}$ with incorrect $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ or $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ is *not guaranteed* to be detected with one traitor SA and two traitor SSAs in IDACS if the authentication chain path is manipulated.

Justification for Claim 2: Under certain circumstances, IDACS cannot guarantee detection of an attack $Ticket_{\psi}$ with one traitor SA and two traitor SSAs in IDACS if authentication chain path manipulation is allowed. The attacker is allowed to choose the first SA in the authentication chain, so he chooses a traitor SA. Since this SA is a Byzantine actor, it calculates the authentication chain path based on $Ticket_{\psi}$ and checks to see whether the last SSA in the authentication chain is also a traitor. If it is, then the SA passes $Ticket_{\psi}$ to this SSA, which then passes $Ticket_{\psi}$ to a DB_{γ} (Fig. 19). This action bypasses the $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ checks that would be performed by (potentially) loyal SAs and SSAs in the authentication chain. It is necessary for the last SSA in the authentication chain to be a traitor, because according to Assumption 7, DB_{γ} will also validate the authentication chain for the sending SSA. When DB_{γ} forms the return ticket $Ticket'_{\psi}$, it will calculate a return authentication chain where the first SSA in the path cannot (by the rules) be the same as the last SSA in the approach path. If this SSA happens to be a traitor also, then it sends $Ticket'_{\psi}$ directly to the first traitor SA, which sends it to the attacker's $Cust_{\psi}$.

Claim 1 and Claim 2 address a) in Assumption 9; similar claims can be made to address b) in Assumption 9.

Claim 3: A $Ticket_{\psi}$ with correct $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ but seeking to access data/services for which $Cust_{\psi}$ is not granted permissions will be detected with up to $(4N - 1)$ traitor SAs and SSAs in the approach and return authentication chains if the authentication chain is not manipulated.

Justification for Claim 3: Since $Ticket_{\psi}$ contains correct $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$, the attack will not be detected on those grounds. However, each SA and SSA also checks the data/service targeted by $Ticket_{\psi}$ to see if $Cust_{\psi}$ has permissions on it. If only one SA or SSA in the approach or return authentication chain is loyal, then a non-permitted $Ticket_{\psi}$ will be detected, and the attack will be prevented.

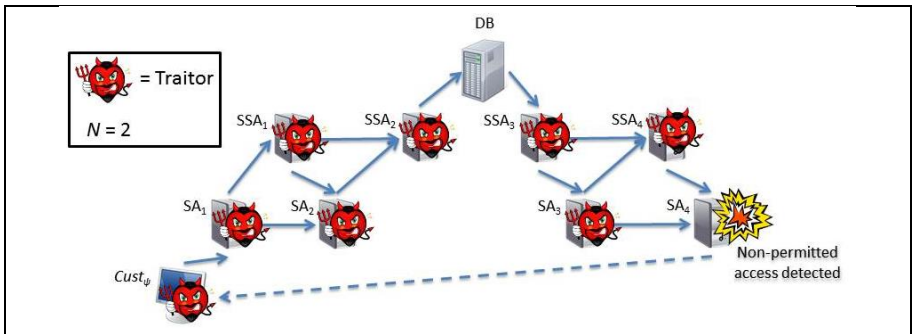


Fig. 20. Illustration of Claim 3

Claim 4: A $Ticket_{\psi}$ with correct $\overline{\boldsymbol{OTP_{\psi}}}$ and $\overline{\boldsymbol{PID_{\psi}}}$ but seeking to access data/services for which $Cust_{\psi}$ is not granted permissions is not guaranteed to be detected with 1 traitor SA and 2 traitor SSAs in IDACS if the authentication chain is manipulated.

Justification for Claim 4: The justification for Claim 4 is identical to the justification for Claim 2.

There is also one final Claim that can be made regarding spoofed IDACS machines.

Claim 5: A spoofed SA_{χ} or SSA_{κ} will be detected as soon as it communicates with a loyal SA_{χ} or SSA_{κ}.

Justification for Claim 5: According to Assumption 6, a spoofed SA_{χ} or SSA_{κ} does not have access to the $Seed_{\sigma}$ of the machine it is spoofing, including the $Seed_{\sigma}$ needed to calculate XV when communicating with other SA_{χ} or SSA_{κ}. Therefore, it will be unable to correctly calculate the requisite XV; this situation will be detected immediately by a loyal SA_{χ} or SSA_{κ}.

4.4 Attack Traceback

When an attack is detected by IDACS, it falls into one of several categories, with each category having corresponding root causes. If an attack is detected based on a OTP_{χ} or $\overline{\boldsymbol{PID_{\psi}}}$ failure, this is because the attacker possesses an incomplete subset of the set { $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, PIN_{λ} }; additionally, if prior SAs or SSAs correctly authenticated the attack packet based on OTP_{χ} and $\overline{\boldsymbol{PID_{\psi}}}$, they may be controlled or spoofed by the attacker. If the attack is detected base on an XV failure, this is because the attacker is spoofing one or more of the SA/SSAs. Each of these situations is handled differently by IDACS.

When an attack is detected in Algorithm 4, the function "report_and_trace_attack()" (Algorithm 6) is called to invoke the IDACS real-time digital forensics suite. The inputs to Algorithm 6 are $\overline{reasons}$, $\overline{which\_PID\_failed}$, TK_A, TK_B, and current_location. $\overline{reasons}$ indicates the reason the attack was detected; it may contain one or more of the following values: OTP_fail, PID_fail, and XV_fail. $\overline{which\_PID\_failed}$ contains a list of which (if any) of the PID_{ε} failed. TK_A and TK_B are the two TKs received by the detecting SA or SSA (if applicable). "current_location" indicates the identity of the SA or SSA detecting the attack. When "report_and_trace_attack()" is called, these inputs are packaged into an attack report (3). If the attack was detected by an SA (1), the report is sent to an SSA for processing (2 and 3). If the attack is detected at an SSA (4), then the attack is processed by that SSA (5). To process the attack, the SSA calls different forensics subroutines based on the reasons for the attack detection. If the attack failed due to OTP_{χ} or $\overline{\boldsymbol{PID_{\psi}}}$ (6), then "trace_attack()" is called to identify the root attacker,

the bot chain used in the attack, and any suspicious packet types that may have been used by the attacker to compromise other bots (7); "identify_bots()" is called to identify possibilities for traitor $Client_{\rho}$ controlled by the attacker (8); and "identify_compromised_items()" is called to determine which members of { $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, PIN_{λ} } have been stolen by the attacker based on correlation with $\overline{which\_PID\_failed}$ (9). If an attack was detected by a failed OTP_{χ}, $\overline{\boldsymbol{PID_{\psi}}}$, or XV (10), then "identify_bad_SA_SSA()" is called to determine which SAs or SSAs (if any) are traitor or spoofed. All of these subroutines are discussed in the following sections.

Algorithm 6. report_and_trace_attack()

inputs : $\overline{reasons}$, $\overline{which\_PID\_failed}$, TK_A, TK_B, current_location
outputs : none

```
1     if ( current_location ∈ SA )
2         dest_SSA = random SSA
3         current_location → dest_SSA: {reasons, which_PID_failed, TK_A, TK_B, PID(current_location) }
4     else if ( current_location ∈ SSA )
5         dest_SSA = current_location
      end

      at dest_SSA:

6         if (OTP_fail ∈ reasons) or (PID_fail ∈ reasons)
7             {sourceTraitorCust, bot_chain, suspicious_pkt_type} = trace_attack(TK_A)
8             traitorCustBotnet = identify_bots(sourceTraitorCust, suspicious_pkt_type, SA, SSA)
9             traitorCustItems = identify_compromised_items(which_PID_failed, TK_A)
          end
10        if (XV_fail ∈ reasons) or (OTP_fail ∈ reasons) or (PID_fail ∈ reasons)
11            traitorSAsSSAs = identify_bad_SA_SSA(reasons, TK_A, TK_B)
          end
```

Fig. 21 presents a block diagram representation of Algorithm 6, showing the relationship between int inputs, outputs and different functions that are called within the algorithm. The following table provides an overview of the traceback algorithms provided by the IDACS real-time digital forensics suite and what types of attacks they are able to detect:

Table 5. Traceback functions and what they detect

Traceback function	What it detects
trace_attack()	Root traitor $Cust_{\psi}$ and other bot $Cust_{\psi}$ used in attack chain
identify_bots()	Attacker's controlled botnet of traitor $Cust_{\psi}$
identify_compromised_items()	Which cryptographic seeds have been leaked, and by whom { $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, PIN_{λ} }
identify_bad_SA_SSA()	Which SAs and SSAs are spoofed bad XV), or traitors (clearing packets with incorrect $\overline{\boldsymbol{OTP_{\psi}}}$ or $\overline{\boldsymbol{PID_{\psi}}}$)

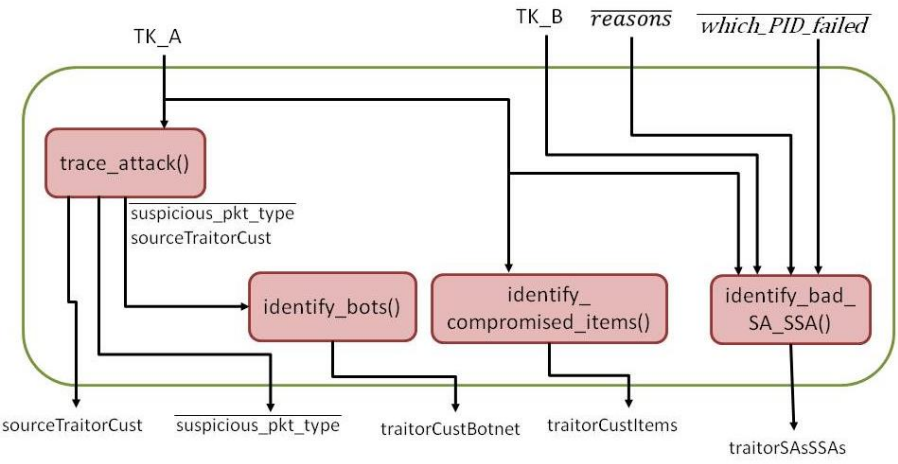


Fig. 21. Block diagram of "report_and_trace_attack()"

When reading the following sections about the different digital forensics functions used by IDACS, it is important to keep the following Property in mind:

Property 7 : The different design elements of IDACS (the distribution of *PID* seeds, the design of Xchain values, the design of security log records, etc.) are carefully crafted to facilitate the real-time digital forensics capabilities of IDACS. Therefore, IDACS is able to provide high-speed forensic services in real-time with minimal overhead.

4.4.1 Log Correlation to Identify Attacking Clients

When an attack is detected by IDACS, the real-time digital forensics suite is able to trace the attack to the root attacker by correlating the security log records on IDACS machines. In a fully-realized IDACS system, all data packets (including Client-Client packets such as are used in attack chains) are required to pass through the SA barrier. Even in a less-complete IDACS system, the $Client_{\rho}$ still maintain security logs for all of the data packets they send and receive. These security logs are the key component to the attacker traceback capability.

As an example, consider the detected attack packet log record on the right in Fig. 22. This record was generated for a data packet that was detected to be an attack due to OTP_{χ} or PID_{ε} failure. IDACS begins from the assumption that this packet was part of an attack chain (Fig. 17), and begins to trace the attack chain back to the root attacker. The trace is based on the log record items TIME, SOURCE/DESTINATION_IP_ADDRESS, PARENT/CURRENT_UA_PID, and CONTENT_PID. The SSA running the trace searches its own logs as well as the logs of other SSAs, SAs, and Clients (if necessary) for the “parent” packet that directly precedes the detected packet in the attack chain. The trace searches for a packet that was logged before the attack was logged (TIME < 14830.528934) where the IP Address of the machine that sent the attack packet is the same as the destination IP address of the "parent" packet (SOURCE_IP_ADDRESS for the attack packet = DESTINATION_IP_ADDRESS for

the "parent" packet); the DB-side data targeted by the Content(PID_ε) in the “parent” packet is the same as in the attack packet (CONTENT_PID = 34876105); and the parent UA for the detected attack packet is the same as the current UA for the “parent” packet (PARENT_UA_PID for the attack packet = CURRENT_UA_PID for the “parent” packet). Such a "parent" attack packet is detected (left side of Fig. 22). The investigating SSA then compares the CURRENT_UA_PID and PARENT_UA_PID in the "parent" packet record. If they are the same, then the machine at SOURCE_IP_ADDRESS is flagged as the root attacker; if they are different, the machine at SOURCE_IP_ADDRESS is flagged as a traitor $Client_p$, and the traceback continues.

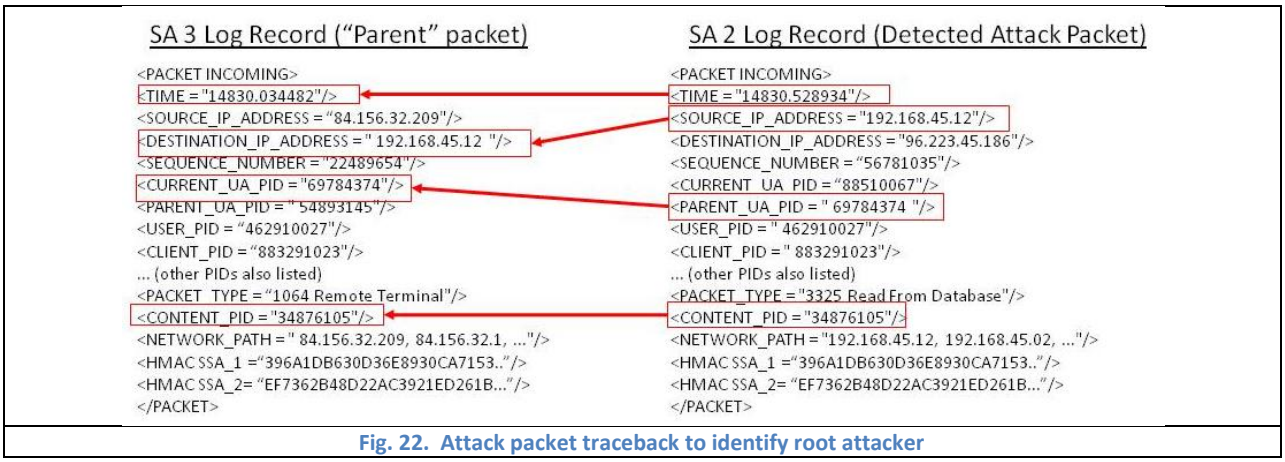


Fig. 22. Attack packet traceback to identify root attacker

The details of this traceback are show in the "trace_attack()" function defined in Algorithm 7. The SSA executing the traceback receives the log record \TK\ of the detected attack packet as input. The critical trace parameters are extracted from this record: the $Cust_\psi$ who sent TK is marked as the candidate for the attacker (1), and the time (2), Parent UA(PID_ε) (3), and Content(PID_ε) (4) are isolated. Additionally, the $\overline{\boldsymbol{bot\_chain}}$ and $\overline{\boldsymbol{suspicious\_pkt\_types}}$ outputs are initialized (5 and 6). Until the source attacker has been identified (7 and 8), the SSA initiates a search of all SSA, SA, and Client logs (10) searching for a "parent" packet logged before the current attack packet with parameters on the Destination IP address, Current UA(PID_ε), and Content(PID_ε) (9). If the F-box($\mathbb{R}$trv) transform (10) returns no hits on the "parent" packet (11), the traceback has failed to detect the root attacker; however, a partial list of bots used in the attack chain can be returned (12). If a "parent" packet with matching Current UA(PID_ε) and Parent UA(PID_ε) is discovered (13), then the $Cust_\psi$ that sent this packet is identified as the root attacker (15). Additionally, since this packet was used in an attack chain, this packet’s type is flagged as suspicious (16) and is used in

Algorithm 8 to identify candidate members of the attacker’s controlled botnet. Otherwise, the Client that sent the "parent" packet is marked as one of the bots controlled by an attacker (17), the search parameters are reset based on the "parent" packet (18 to 21), this packet’s type is flagged as suspicious (16), and the search continues (8). Once the root attacker has been identified,

the function returns the identity of the root attacker, the bots that were identified in the attack chain, and the suspicious packet types (23).

Algorithm 7. trace_attack()

```
inputs: \TK\
outputs: sourceAttacker, bot_chain, suspicious_pkt_type

1       sourceAttacker = Cust_ψ ◊ \TK\
2       sourceTme = current time
3       sourceParentUAPID = Parent_UA(PID_ε) ◊ \TK\
4       sourceContentPID = Content(PID_ε) ◊ \TK\
5       bot_chain = null
6       suspicious_pkt_types = null
7       sourceFound = false

8       while (sourceFound == false)
9           parameters = {time ◊ \Ticket_ψ\ < sourceTime,
                          destination_IP ◊ \Ticket_ψ\ = = sourceAttacker,
                          Current_UA(PID_ε) ◊ \Ticket_ψ\ == sourceParentUAPID,
                          Content(PID_ε) ◊ \Ticket_ψ\ == sourceContentID}
10          source\Ticket_ψ\ = F-box(Retrv, SSA, SSA, parameters)

11          if(source\Ticket_ψ\ == null)
12              fail;    return {null, bot_chain}
13          else if((Parent_UA(PID_ε) ◊ source\Ticket_ψ\) == (Current_UA(PID_ε) ◊ source\Ticket_ψ\))
14              sourceFound =true
15              sourceAttacker = Cust_ψ ◊ source\Ticket_ψ\
16              suspicious_pkt_types = F-box(Concat, suspicious_pkt_types, packet_type(source\Ticket_ψ\))
            else
17              bot_chain = F-box(Concat, bot_chain, Cust_ψ ◊ source\Ticket_ψ\)
18              sourceAttacker = Cust_ψ ◊ source\Ticket_ψ\
19              sourceTime = time ◊ source\Ticket_ψ\
20              sourceParentUAPID = Parent_UA(PID_ε) ◊ source\Ticket_ψ\
21              sourceContentID = Content(PID_ε) ◊ source\Ticket_ψ\
22              suspicious_pkt_types = F-box(Concat, suspicious_pkt_types, packet_type(source\Ticket_ψ\))
            end
        end

23      return { sourceAttacker, bot_chain, suspicious_pkt_types }
```

4.4.2 Log Correlation to Identify Traitor Client Botnet

Immediately following the call of "trace_attack()" to identify the root traitor Client and the traitor Client bots in the attack chain, the SSA running the real-time digital forensics suite will run the "identify_bots()" function to identify all traitor Client bots controlled by the attacker, even those in a dormant state. In the example shown in Fig. 22, it was seen that the root traitor Client and other traitor Clients in the attack chain used "Remote Terminal" packets to carry out attack activities. Therefore, the digital forensics suite designates "Remote Terminal" packets, especially those sent by the root Traitor Client, to be suspicious. The SSA running the real-time digital forensics suite searches through the security log records of all SAs, SSAs, and Clients to identify suspicious network traffic. In the example shown in Fig. 23, the digital forensics suite searches for "Remote Terminal"

packets in the security logs that were sent by the root traitor Client (SOURCE_IP_ADDRESS = 75.128.32.146). This search yields a number of Clients that are strong candidates for being Traitors. This list of potential Traitor Clients is added to those identified in the attack chain in "trace_attack()"; these Clients may be quarantined and examined in-depth until they can be healed and returned to service. It should be noted that this algorithm is capable of detecting dormant traitor Clients even beyond those that were used in the attack.

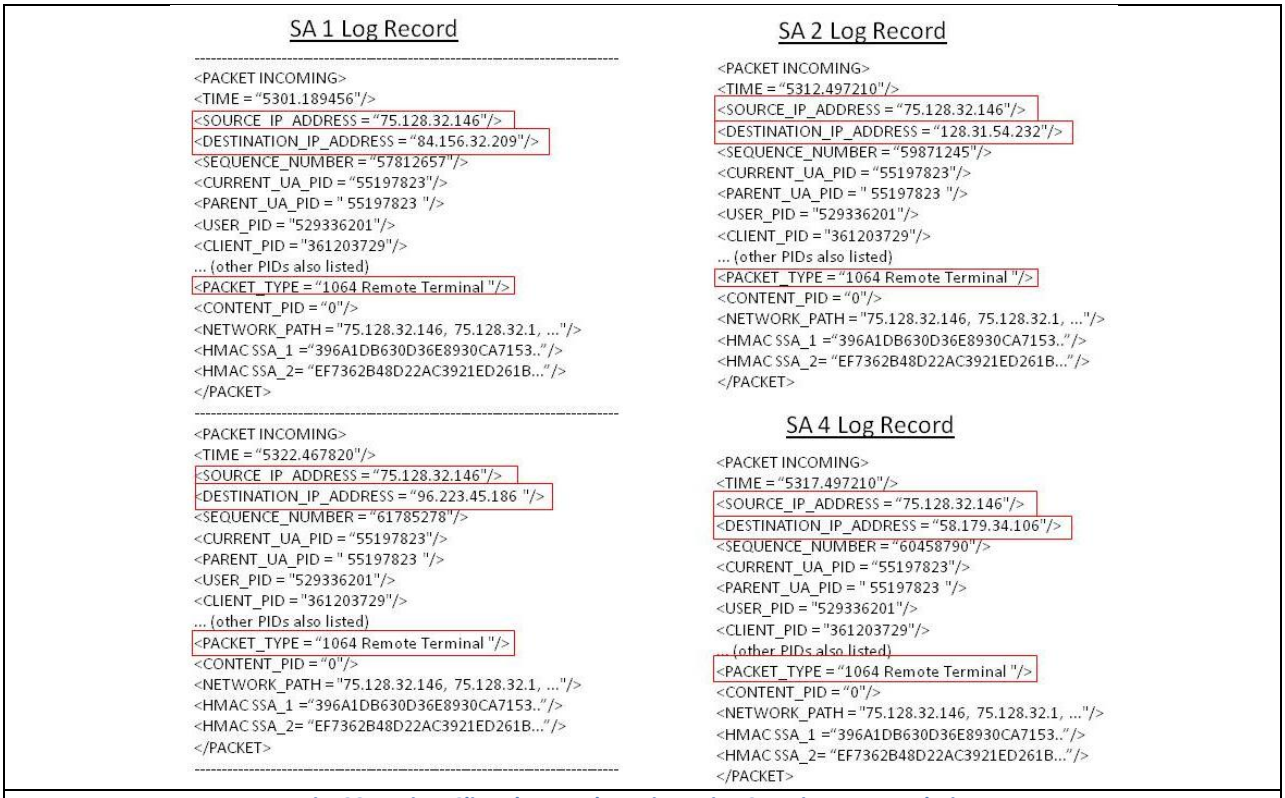


Fig. 23. Traitor Client botnet detection using Security Log correlation

Algorithm 8 details the "identify_bots()" algorithm. The function receives the identity of the root Traitor Client, any suspicious packet types as determined by the "trace_attack()" algorithm, and $\overline{SA}$ and $\overline{SSA}$ as inputs. It creates a set of parameters (1) to be the input to the F-box($\mathcal{R}$trv) transform; the packets sought by the F-box($\mathcal{R}$trv) transform must be among the suspicious packet types and must have originated from the attacker detected by "trace_attack()". The function then loops through every SA (2) and every SSA (5), searching for log records that meet the search parameters (3 and 6). For any security log records that match the conditions, the $Cust_{\psi}$ that were targeted by those packets are added to the list of possible traitor Clients controlled by the attacker (4 and 7).

Algorithm 8. identify_bots()

inputs: attacker-$Cust_{\psi}$, $\overline{\boldsymbol{suspicious\_pkt\_type}}$, $\overline{\boldsymbol{SA}}$, $\overline{\boldsymbol{SSA}}$
outputs: $\overline{possible\_traitors}$

1 $\overline{parameters}$ = { packet type(\\$Ticket_{\psi}$\\) ∈ $\overline{\boldsymbol{suspicious\_pkt\_type}}$,
source IP(\\$Ticket_{\psi}$\\) == source IP(attacker-$Cust_{\psi}$)}

2 for index=1 to χ
3 temp = F-box(Retrv, SSA_{χ}, $\overline{parameters}$)
4 $\overline{possible\_traitors}$ = F-box(Concat, $\overline{possible\_traitors}$, destination_IP ◊ temp)
end

5 for index=1 to κ
6 temp = F-box(Retrv, SSA_{κ}, $\overline{parameters}$)
7 $\overline{possible\_traitors}$ = F-box(Concat, $\overline{possible\_traitors}$, destination_IP ◊ temp)
end

8 return $\overline{possible\_traitors}$

4.4.3 PID/OTP Correlation to Identify Client-Side Security Items

The space-time separated and jointly evolving relationship built into IDACS can be used to assist the real-time digital forensics capabilities. The elements of $\overline{\boldsymbol{PID_{\psi}}}$ are calculated based on seeds drawn from $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, and PIN_{λ}. However, not every PID_{ε} needs to draw seeds from each of these sources; the locations of seeds used to calculate individual PID_{ε} can be tailored to meet the security requirements of the system. In addition to the distributed storage of the seeds, certain bits are removed from the seeds themselves and stored in a physically different location. These bits are called Xbits; they will be discussed in more detail "5.2.1 Separation of Encryption Keys". The Xbits are stored on the SAs in the IDACS Network, and they must be recombined with the cryptographic seeds to correctly calculate the elements of $\overline{\boldsymbol{PID_{\psi}}}$. Fig. 24 demonstrates how different PID_{ε} can be calculated using different combinations of cryptographic seeds and xbits; Type A PID_{ε} are calculated using only seeds from $Client_{\rho}$, Type B PID_{ε} are calculated using only seeds from $Client_{\rho}$ and the associated Xbits, Type F PID_{ε} are calculated using seeds and associated Xbits from both $Client_{\rho}$ and PIN_{λ}, etc. This division accomplishes two purposes. First, separating the seeds across different locations increases the complexity for an attacker to correctly construct $\overline{\boldsymbol{PID_{\psi}}}$, as discussed previously. Second, a seed separation and combination such as indicated in Fig. 24 can be used as a forensics tool. If an attack is detected by an SA or SSA due to $\overline{\boldsymbol{PID_{\psi}}}$ failure, an analysis of which PID_{ε} failed and which PID_{ε} were formed correctly can indicate which Client-side items and Network-side SAs or SSAs are Traitors or have had their memory

compromised. For example, if a Type D PID_{ε} was formed correctly, it may be assumed that the attacker owns the seeds derived from PIN_{λ} as well as the associated Xbits; if a Type F PID_{ε} was formed correctly, it may be assumed that the seeds stored on $Client_{\rho}$ and derived from PIN_{λ} as well as the associated Xbits are owned by the attacker.

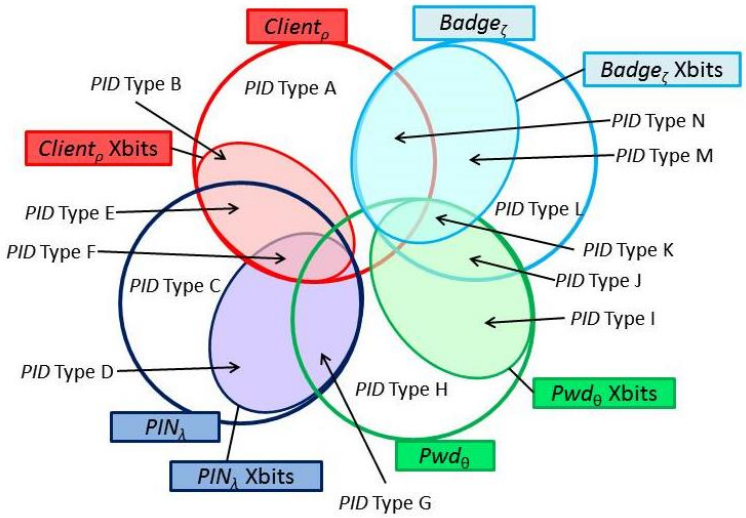


Fig. 24. Space-separated combinations of seeds used to calculate PID_{ε}

An example of the "identify_compromised_items()" algorithm corresponding to Fig. 24 is shown in Algorithm 9. The $Cust_{\psi}$ that sent the detected attack packet TK_1 is marked as a Traitor (1), with the $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, and PIN_{λ} contained in $Cust_{\psi}$ and the SAs storing their corresponding Xbits all possibly controlled/cloned by an attacker. Based on what type of PID_{ε} were formed correctly in the attack packet TK_1, certain elements of $Cust_{\psi}$ and $\overline{\boldsymbol{SA}}$ are marked as Traitor (4 – 11). The following properties hold true for this forensics capability:

Property 8 : Due to the seed distribution and PID_{ε} formation (as discussed generically in Property 7, the checks performed in "identify_compromised_items()" are performed very quickly with very little overhead.

Property 9 : The seed distribution shown in Fig. 24 is very flexible, and can be adjusted on a per-Client basis to meet the security needs of the particular Client or IDACS implementation.

Algorithm 9. identify_compromised_items()

inputs: $\overline{which\_PIDs\_failed}$, TK_A
outputs: traitor_items

```
1       traitor_C = Cust_ψ ◊ TK_A
2       traitor_items = null

3       switch (PID_ε ∉ which_PIDs_failed)
4           case Type A, B, E, F, or N:  traitor_items = F-box(Concat, traitor_items, Z ◊ compromised_C)
5           case Type C, D, E, F, or G:  traitor_items = F-box(Concat, traitor_items, X ◊ compromised_C)
6           case Type G, H, I, J, or K:  traitor_items = F-box(Concat, traitor_items, W ◊ compromised_C)
7           case Type J, K, L, M, or N: traitor_items = F-box(Concat, traitor_items, Y ◊ compromised_C)

8           case B, E, or F: traitor_items = F-box(Concat, traitor_items, SA storing xbits for Z ◊ traitor_C)
```

```
9           case D, F, or G: traitor_items = F-box(Concat, traitor_items, SA storing xbits for X ◊ traitor_C)
10          case I, J, or K: traitor_items = F-box(Concat, traitor_items, SA storing xbits for W ◊ traitor_C)
11          case K, M, or N: traitor_items = F-box(Concat, traitor_items, SA storing xbits for Y ◊ traitor_C)
        end

12      return traitor_items
```

4.4.4 TK Correlation to ID Traitor SA/SSAs

Fig. 25 graphically illustrates the handling of the XV used in the "run_auth_chain()" algorithm outlined in Algorithm 4. Multiple iterations of Algorithm 4 are called by Algorithm 1 (7 and 8) in order to form the complete authentication chain of SAs and SSAs, as shown in Fig. 26. The iterations are linked, with the SA_2 and SSA_2 of one iteration becoming the SA_1 and SSA_1 of the next iteration (Fig. 25 and Fig. 26). In a given iteration, SA_1 and SSA_1 perform OTP_X and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ checks and generate XV, while SA_2 and SSA_2 verify the XV to verify the identity of SA_1 and SSA_1. At SA_2, if the XV_2 verification fails, it indicates that SA_1 is being by an attacker; if the XV_3 verification fails, this indicates that SSA_1 is being spoofed by an attacker. In the same way, at SSA_2 if the XV_5 verification fails, it indicates that SSA_1 is being spoofed; if XV_6 fails, it indicates that SA_2 is being spoofed. These relationships are discussed more extensively in "3.2.4.2 IDACS Network-side Algorithms".

Additionally, each SA_2 and SSA_2 in a given iteration of Algorithm 4 are the SA_1 and SSA_1 for the next iteration, and will also be performing OTP_X and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ checks in the next iteration. If an SA or SSA finds that the $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ fails the check, but a previous SA or SSA indicated that the $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$ had passed the check (by passing $Ticket_\psi$ along the authentication chain), then this is highly indicative that the previous SA or SSA is a traitor (having passed verifiably bad $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$). In the same way, if an SA_X finds that OTP_X fails the check, but a previous SA passed $Ticket_\psi$ along the authentication chain, this may be indicative that the previous SA is a traitor. This cannot be directly verified, since only the previous SA possesses the seeds to verify his OTP; however, the forensics engine can be configured on the assumption that it is statistically unlikely that the seeds to correctly calculate a given OTP could be obtained without obtaining the seeds for all OTPs.

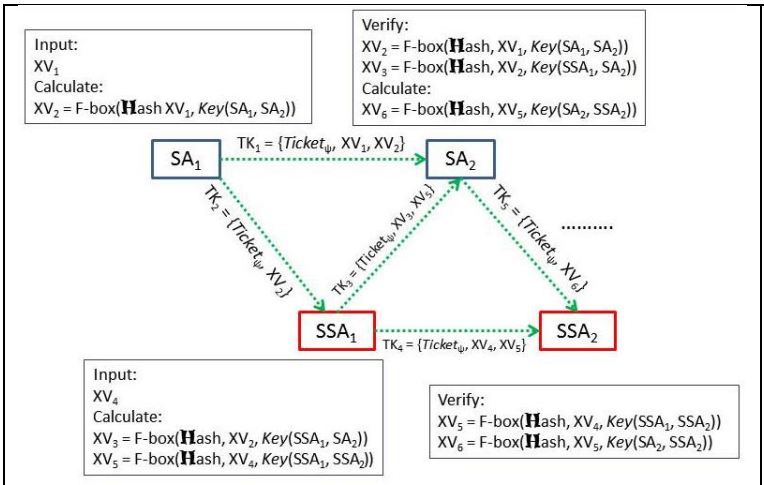


Fig. 25. Mutual Authentication in authentication chain

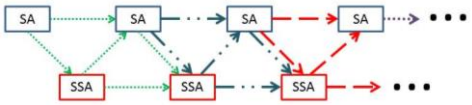


Fig. 26. Cascading the run_auth_chain() algorithm

Due to the design of $\overline{\boldsymbol{OTP_\psi}}$, $\overline{\boldsymbol{PID_\psi}}$, and the XV relationships, traitor or spoofed SAs or SSAs can be detected and isolated very quickly. Algorithm 10 illustrates how the digital forensics suite performs this detection.

Property 10 : The relationships between the XV values (as discussed generically in Property 7) and also $\overline{\boldsymbol{OTP_\psi}}$ and $\overline{\boldsymbol{PID_\psi}}$ are carefully designed to allow traitor or cloned SAs/SSAs to be detected quickly in real time with very little overhead.

Algorithm 10. identify_bad_SA_SSA()

```
inputs: reasons, TK_A, TK_B
otputs: compromised_machines

1     traitor_spoofed_machines = null
2     sourceA = origin of TK_A
3     sourceB = origin of TK_B

4     if (TK_A_XV_fail ∈ reasons) OR (TK_A_PID_fail ∈ reasons) OR (TK_A_OTP_fail ∈ reasons)
5        traitor_spoofed_machines = F-box(Concat, traitor_spoofed_machines, sourceA)
      end
6     if (TK_B_XV_fail ∈ reasons) OR (TK_B_PID_fail ∈ reasons) OR (TK_B_OTP_fail ∈ reasons)
7        traitor_spoofed_machines = F-box(Concat, traitor_spoofed_machines, sourceB)
      end

8     return traitor_spoofed_machines
```

4.5 Simulations

4.5.1 Attack Traceback Time Simulations

The simulations discussed in "3.4 Simulations" were also used to simulate the attack traceback time. When an attack was detected (i.e. (4), (11), (17), or (23) in Algorithm 4), this point in time was recorded as T1. After the attack had been reported to an SSA and the traceback to identify the root attacker was completed (i.e the "trace_attack()" algorithm called at (7)

of Algorithm 6 completes), this point in time was recorded as T2. After the attacker's botnet was identified (i.e. the "identify_bots()" function called at (8) of Algorithm 6 completes), this point in time was recorded as T3. The statistics of interest in this situation were the (T2 - T1) time and the (T3 - T1) time. The (T2 - T1) time is termed the "Attack Traceback Time", since it represents the time it takes for the root attacker to be identified after the attack is detected. The (T3 - T1) time is termed the "Botnet Detection Time", since it represents the time it takes for the root attacker's botnet to be detected after the attack is detected. It should be noted here that the "Botnet Detection Time" will always be greater than the "Attack Traceback Time", since (T3 - T1) = (T2 - T1) + (T3 - T2).

The IDACS Network measured in Fig. 27 through Fig. 29 consisted of 4 "partially compromised" SAs and either 1 or 2 "partially compromised" SSAs. Fig. 27 shows the average attack traceback time for an IDACS network with 1 SSA. Fig. 27 shows that the attack traceback for this simulated IDACS network is extremely fast, with both the root attacker and its botnet identified in less than 3.5 milliseconds even for a network of 100,000 Client Devices. Additionally, the attack traceback time grows logarithmically with the network size; this is because the simulation uses $\log_2(x)$ to calculate security log search times, since there currently exist search algorithms that are better than $\log_2(x)$. Because the attack traceback time is so short, an IDACS Netowork can alert a system administrator and begin network healing procedures before the attacker even realizes that the attack has been detected.

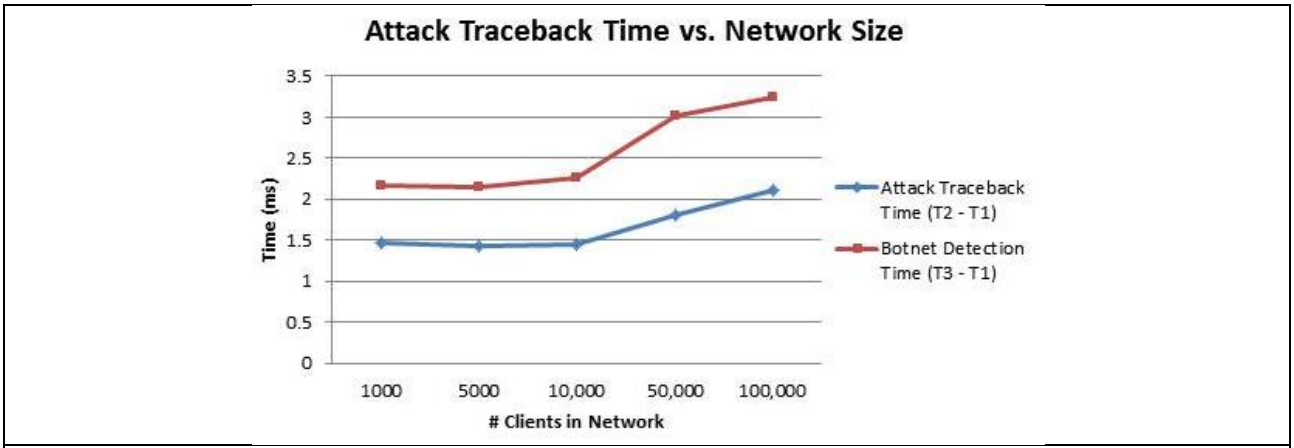


Fig. 27. Attack Traceback Time

An additional benefit of the IDACS system is that attack traceback time can be improved by scaling the network. Fig. 28 and Fig. 29 show the Attack Traceback Time and Botnet Detection Time for an IDACS network, one with 1 SSA, and one with 2 SSAs. These figures show that the traceback times can be dramatically improved by expanding the network side of the IDACS system; this is because the traceback duties are spread across multiple SSAs, resulting in a lighter workload for each machine. These findings provide significant incentive to expand fielded IDACS systems.

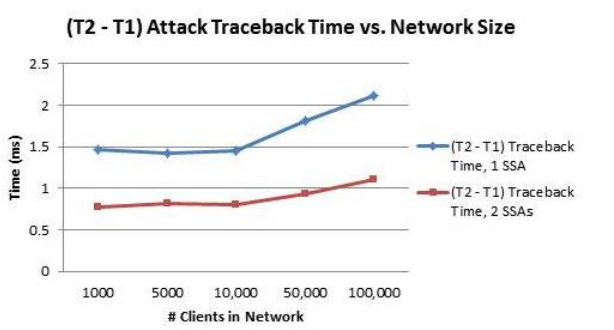


Fig. 28. Attack Traceback Time, 1 SSA vs. 2 SSAs

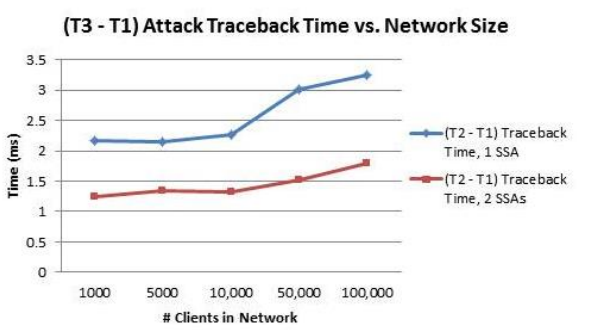


Fig. 29. Botnet Detection Time, 1 SSA vs. 2 SSAs

4.5.2 Attack Detection, Traceback, and Remediation Simulations

In addition to the simulations described in "3.4 Simulations", a second simulation suite performed for this research addresses the effects of the attack traceback combined with quarantine and healing for Byzantine traitor agents. Given an IDACS network under attack by parties that are able to steal Client authentication items and turn SAs and SSAs into traitors, how well will the attack traceback protect the IDACS datacenter from illegal (no permission) access? The simulation results presented here attempt to answer that question.

4.5.2.1 Assumptions for Simulation

In order to fully test the capabilities of the IDACS network against real-world threats and attacks, it was necessary to construct attack scenarios based on the latest and most lethal real-world attack vectors. Therefore, these simulations were carried out under the assumption that all attempts to hack SAs and SSAs and turn them into traitors would be accomplished using zero-day attacks. Since zero-day attacks have not been previously observed, it is virtually impossible to defend against them, and they will almost always be successful. Additionally, through the use of metamorphic evolution techniques, it is possible to generate endless variants of these zero-day turn-traitor-attacks, each of which has a unique signature. This method can be used to defeat security systems that use signature-based scans to detect known turn-traitor-attacks. Since both of these attack methods are widely in use today, they will both be considered in this simulation.

This simulation is based on a number of assumptions, each of which is justifiable according to real-world conditions. The first assumptions on which this simulation is based are as follows:

Assumption 10: Previously unobserved zero-day turn-traitor-attacks used to gain control over network machines will require a relatively long time (weeks) for a patch that successfully secures that attack's entry point to be issued.
Assumption 11: A zero-day attack used to gain control over network machines, once detected and analyzed, can be blacklisted with a signature-scanning security system very quickly. A zero-day turn-traitor-attack or a metamorphic variation thereof, once detected and blacklisted, will be detected and blocked 100% of the time thereafter.

Assumption 10 is reasonable; it is seen to be true across almost all computer security vulnerabilities that are being discovered today. Assumption 11 reflects the strengths of signature-based scanners, although their strength may be slightly overstated in order to simplify this simulation. Based on Assumption 11, it follows that attacker behavior will reflect this reality.

Assumption 12: A zero-day turn-traitor-attack or a metamorphic variation thereof, once detected and blocked, will not be reused by the attacker.

Based on the relative importance of $Cust_\psi$, $\overline{\boldsymbol{SA}}$, and $\overline{\boldsymbol{SSA}}$ in IDACS, they are accorded different levels of protection against theft ($Cust_\psi$) or outside zero-day turn-traitor-attacks ($\overline{\boldsymbol{SA}}$ and $\overline{\boldsymbol{SSA}}$). $Cust_\psi$ are used by human users in the field, so the elements of $Cust_\psi$ ($Client_\rho$, $Badge_\zeta$, Pwd_θ, and PIN_λ) are (relatively) easy to steal, although it may be difficult to steal a complete set. On the other hand, $\overline{\boldsymbol{SA}}$ and $\overline{\boldsymbol{SSA}}$ reside inside protected network datacenters, so they are relatively more difficult to gain control over. Thus, the assumptions:

Assumption 13: Completely turning a Client into a traitor (with access to $Client_\rho$, $Badge_\zeta$, Pwd_θ, and PIN_λ) through theft or coercion is difficult. An exception would be in an active battlefield scenario, where a number of human users (soldiers) could be captured and coerced into turning over all of the elements of $Cust_\psi$.

Assumption 14: $Cust_\psi$ are easier to turn into traitor bots than SAs, and SAs are easier to turn into traitor bots than SSAs.

Finally, any attacker, being intelligent and wishing to maximize his chances of success, will not launch attempts to access the IDACS datacenter until he has a certain chance of success. Thus,

Assumption 15: An attacker will not launch access-DB-attacks against the IDACS datacenter until he controls a certain number of traitor $Cust_\psi$, SA_χ and SSA_κ.

4.5.2.2 Simulation Parameters

This simulation was implemented in MATLAB, and examined an IDACS network consisting of 500 $Cust_\psi$, 40 SAs, 20 SSAs, and 10 DB (reference Fig. 3). Unlike the first simulation, this simulation did not consider the details of network transmission speeds or packet processing times. All packets are considered to be transmitted from one machine to another in one clock cycle, and all packets are processed in one clock cycle, with one clock cycle per queued packet.

The simulation consists of two phases. In Phase 1, the attacker uses turn-traitor-attacks to build a botnet of traitor $Cust_\psi$, SAs, and SSAs for use in IDACS access-DB-attacks. According to Assumption 15, the attacker builds a botnet consisting of traitor SAs equaling 60% of all SAs in IDACS, and traitor SSAs equaling 60% of all SSAs in IDACS (15% of the traitor SAs and SSAs were spoofed machines). Additionally, the attacker builds a botnet consisting of four traitor $Cust_\psi$ for each traitor SA and SSA (this number was experimentally determined to provide a sufficient number of traitor $Cust_\psi$ to launch a sufficient number of access-DB-attacks for the duration of the simulation). In accordance with Assumption 13, this simulation is assumed to

represent an active battlefield situation, so 15% of the traitor $Cust_{\psi}$ have full access to their authentication credentials ($Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, and PIN_{λ}). Once a sufficient number of bots have been obtained, the attacker launches Phase 2.

In Phase 2, the attacker sends a burst of a high number of access-DB-attacks. The logic behind the burst is that an attacker maximizes his chance of successful datacenter accesses if he sends them quickly; detected and prevented access-DB-attacks will result in the detection and quarantine of traitor $Cust_{\psi}$, SAs, and SSAs. By sending a burst of access-DB-attacks, the attacker makes full use of these bots before they are detected and quarantined, and the advantage gained from Assumption 15 begins to slip away. During Phase 2, access-DB-attacks are launched at an average rate of one attempt per 15 clock cycles (the actual start times of the access-DB-attacks are randomized using a normal distribution over the complete period of Phase 2). If an illegal data center access is detected and traced to one or more traitor machines based on the methods discussed in "4.4 Attack Traceback", that machine is quarantined (removed from the IDACS network) and healed over a period of 100 clock cycles, and then returned to IDACS as a loyal $Cust_{\psi}$, SA_{χ} or SSA_{κ}. During Phase 2, the attacker continues to attack $Cust_{\psi}$, SA_{χ} and SSA_{κ} and turn them into traitors, thus replenishing the botnet even as bots are detected and quarantined. In accordance with Assumption 14, one new $Cust_{\psi}$ is turned traitor every 150 clock cycles, one new SA_{χ} is turned traitor every 300 clock cycles, and one new SSA is turned traitor every 600 clock cycles (this is on average; the actual times the machines turn traitor are randomized over the period of Phase 2 using a normal distribution). In accordance with Assumption 15, if the percentage of traitor SAs or SSAs in IDACS fell below 10%, the attacker stopped launching access-DB-attacks until both of those numbers rose above 10%. As long as there was any traitor $Cust_{\psi}$ available, access-DB-attacks would continue. All traitor $Cust_{\psi}$ *without* access to complete authentication credentials ($Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, and PIN_{λ}) launched access-DB-attacks against data/services that particular $Cust_{\psi}$ had permissions to access (the access-DB-attack was illegal due to incorrect $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ or $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$), but all traitor $Cust_{\psi}$ *with* access to complete authentication credentials launched access-DB-attacks against data/services that particular $Cust_{\psi}$ did not have permissions to access (since a traitor $Cust_{\psi}$ with access to complete authentication credentials is indistinguishable from a loyal $Cust_{\psi}$, access of data/services for which that $Cust_{\psi}$ has correct permissions cannot be detected; therefore, this situation was not addressed in this simulation).

In this simulation, the botnet-building activity of Phase 1 was compressed into a period of 100 clock cycles (in reality, this botnet building could occur in a "low-and-slow" turn-traitor-attack strategy over the course of weeks or months). Phase 2 activity was simulated over a period of 4000 clock cycles. The length of both the approach and return authentication chains was 4 (N=4), as would be expected in a fielded IDACS implementation.

In order to address the question of zero-day turn-traitor-attacks with metamorphic variants, the simulation was divided into three scenarios. In Scenario 1, whenever an access-DB-attack is detected and prevented, one or more traitor $Cust_{\psi}$, SA_{χ} or SSA_{κ} is identified. This traitor is quarantined and healed, but no attempt is made to analyze the zero-day attack used to turn that machine into a traitor. Therefore, the same zero-day turn-traitor-attack can be used again to turn other machines into traitors during Phase 2. In Scenario 2, there are 20 different zero-day turn-traitor-attacks used to turn machines into traitors. When a traitor machine is identified, the IDACS forensics suite analyzes the zero-day turn-traitor-attack that was used to turn this machine into a traitor. A signature for the zero-day turn-traitor-attack is identified and added to each $Cust_{\psi}$, SA_{χ} and SSA_{κ}'s blacklist in accordance with Assumption 11, and will not be successful in turning any more machines into traitors during Phase 2. Therefore, the attacker will stop using that zero-day turn-traitor-attack according to Assumption 12. In Scenario 3, the attacker begins with 20 different zero-day turn-traitor-attacks and 20 metamorphic variants of each zero-day attack. Each analyzed and blacklisted metamorphic turn-traitor-attack variant will no longer be used, but many other metamorphic variants will be available. In short, Scenario 1 represents the "simplified case" situation, Scenario 2 represents the "best case" situation, and Scenario 3 represents the "realistic case" situation. Results from these three Scenarios are presented in the following section.

4.5.2.3 Simulation Results

Each of these three scenarios was simulated 10 times, and the results averaged together. This was done to gain a better view of broad trends and mask random variations in different simulation runs.

The purpose of this simulation was to demonstrate how well IDACS could protect the datacenter from illegal access. This can be measured in two ways. First, the number of successful illegal accesses over the period of the simulation indicates the success of the IDACS defense. Second, the number of undetected traitor $Cust_{\psi}$, SAs, and SSAs remaining in IDACS over the course of the simulation demonstrate how effectively IDACS is detecting, quarantining, and healing traitor machines.

Fig. 30 shows the percentage of the active SAs and SSAs in IDACS that are traitors for Scenario 1, or the "simplified case" situation (because there is no attack analysis or blocking in this scenario). Note that this graph counts only SAs and SSAs that are "active", i.e. not under quarantine; this measurement was chosen to reflect the network situation faced by an attacker trying to pass an illegal datacenter access through IDACS. After the botnet-building period represented by the first 100 clock cycles of the simulation, access-DB-attacks commence, leading to the discovery and quarantine of traitor SAs and SSAs. Initially both traitor SAs and SSAs are detected at a high rate, leading to the increased "sanitation" of the IDACS Network. When the percentage of traitor SAs in the system drops below the 10% mark, access-DB-attacks are no longer launched until additional SAs are turned into traitors; since fewer access-DB-attacks are being launched, it leads to traitor SSAs being identified

at a lower rate (since the "first line" of SAs will trigger illegal access detection, leading to fewer SSAs having the opportunity to be detected), and with the slow addition of new traitor SSAs, the percentage of traitor SSAs rises to a slightly higher level. The percentage of both traitor SAs and SSAs stabilizes to a relatively low equilibrium point.

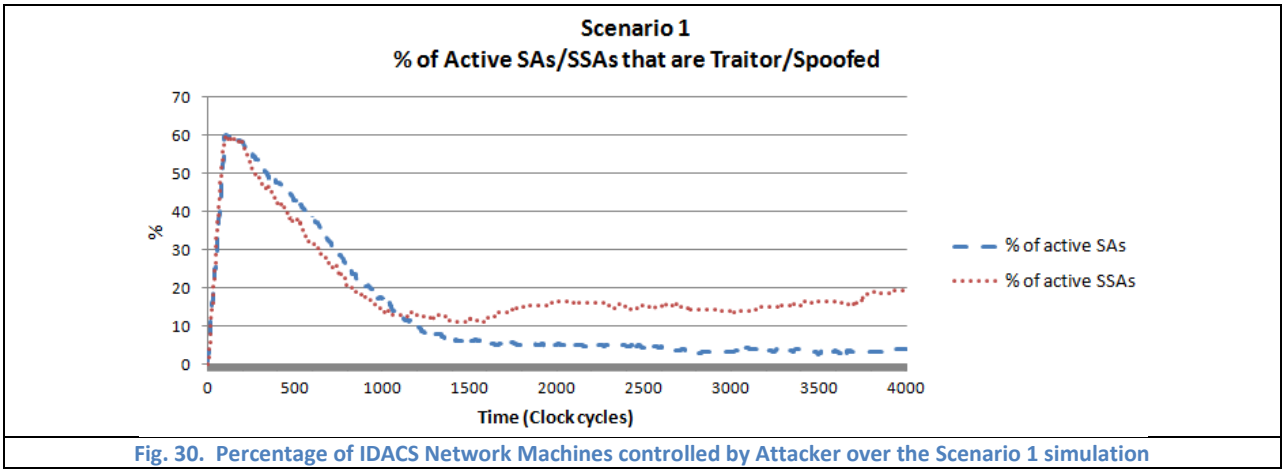


Fig. 30. Percentage of IDACS Network Machines controlled by Attacker over the Scenario 1 simulation

Fig. 31 shows the percentage of active SAs and SSAs that are traitors for Scenario 2, or the "best case" situation (because there is attack analysis and blocking with a limited number of unique zero-day turn-traitor-attacks, so eventually no more new SAs or SSAs can be turned into traitors). As in Scenario 1, the percentage of traitor SAs and SSAs both drop as traitors are identified and quarantined. Once all 20 unique zero-day turn-traitor-attacks have been analyzed and blocked, no new SAs or SSAs can be turned into traitors; therefore the percentage of traitor SAs and SSAs both drop to levels below the 10% cutoff, and remain level as no new traitors are added and no more traitors are identified (since access-DB-attacks are not launched below the 10% cutoff). Scenario 2 demonstrates better performance than Scenario 1; however, the scenario most reflective of a real-world situation is Scenario 3.

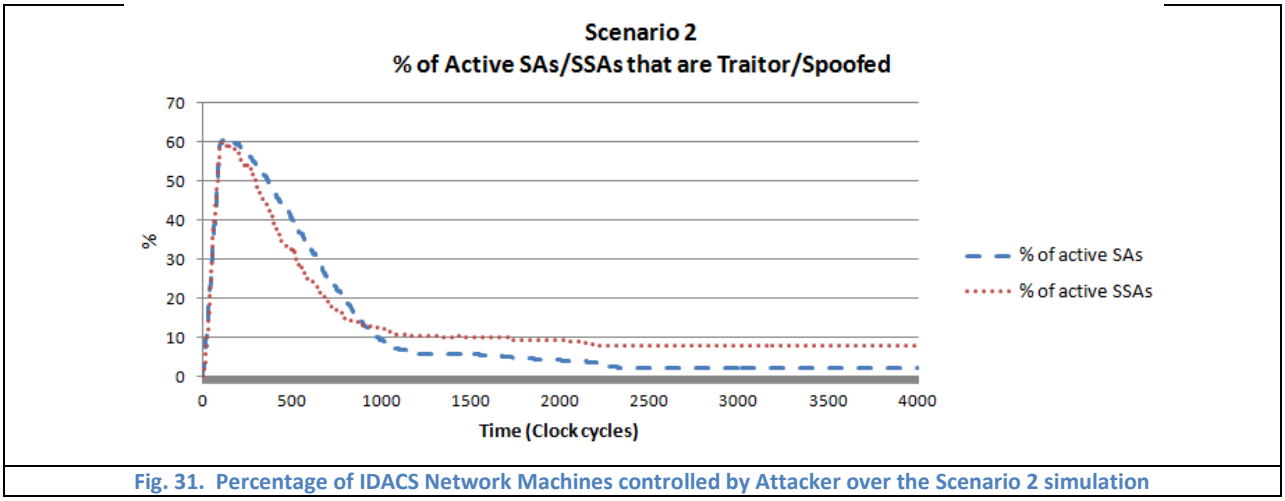


Fig. 31. Percentage of IDACS Network Machines controlled by Attacker over the Scenario 2 simulation

Fig. 32 shows the percentage of active SAs and SSAs that are traitors for Scenario 3, or the "realistic case" situation (because there is attack analysis and blocking, but with metamorphic turn-traitor-attack variants, there is a large supply of new, unidentified turn-traitor-attack variants). Since this scenario gives the attacker a large number of attack variants, the performance of Scenario 3 in Fig. 32 is similar to the performance of Scenario 1 in Fig. 30. However, some detected zero-day turn-traitor-attack variants are re-used before the attacker realizes they have been detected and blocked, and are subsequently blocked by IDACS; therefore, the performance in Scenario 3 is somewhat better than Scenario 1. This can be observed in the lower equilibrium point of percentage of traitor SSAs towards the end of the simulation period. Since Scenario 3 most closely reflects the "real-world" situation faced by a fielded IDACS system, Fig. 32 demonstrates that IDACS will provide an exceptional defense against an attacker with a botnet of Byzantine SAs and SSAs at his disposal.

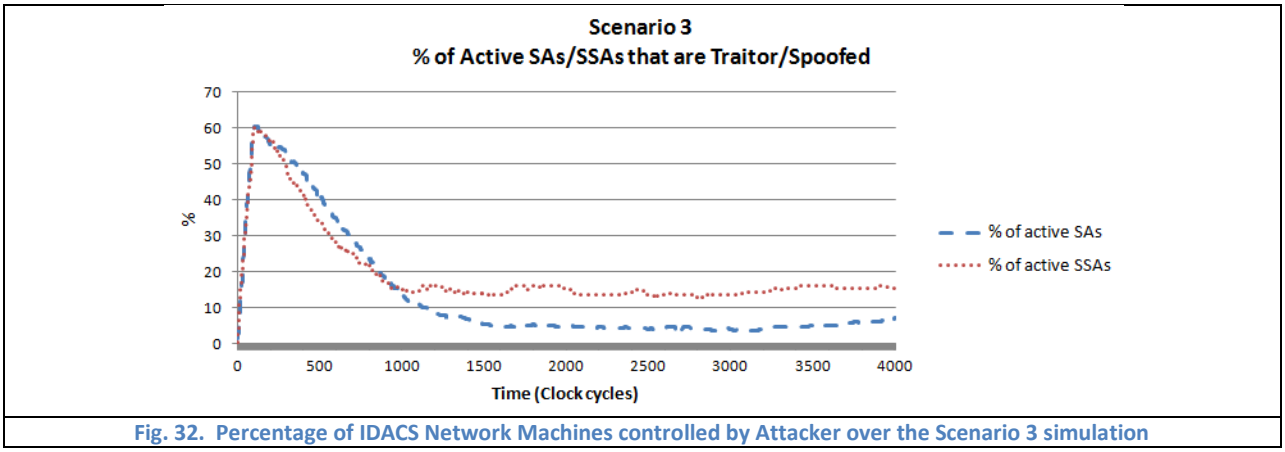


Fig. 32. Percentage of IDACS Network Machines controlled by Attacker over the Scenario 3 simulation

Fig. 33 shows the percentage of active $Cust_{\psi}$ that are controlled by the attacker across the simulation for the different Scenarios. This graph shows that Scenario 3 performs better than Scenario 1 for "sanitizing" the set of $Cust_{\psi}$; this is because as zero-day turn-traitor-attack variants are detected and blocked, but still re-used before the attacker is aware they have been blocked, fewer new $Cust_{\psi}$ are being tuned into traitors. Additionally, since more traitor SAs and SSAs are added in Scenario 3 than in Scenario 2, more access-DB-attacks are launched, allowing more traitor $Cust_{\psi}$ to be detected in the end. This graph also demonstrates that a real-world IDACS system will have excellent performance in identifying and sanitizing traitor $Cust_{\psi}$.

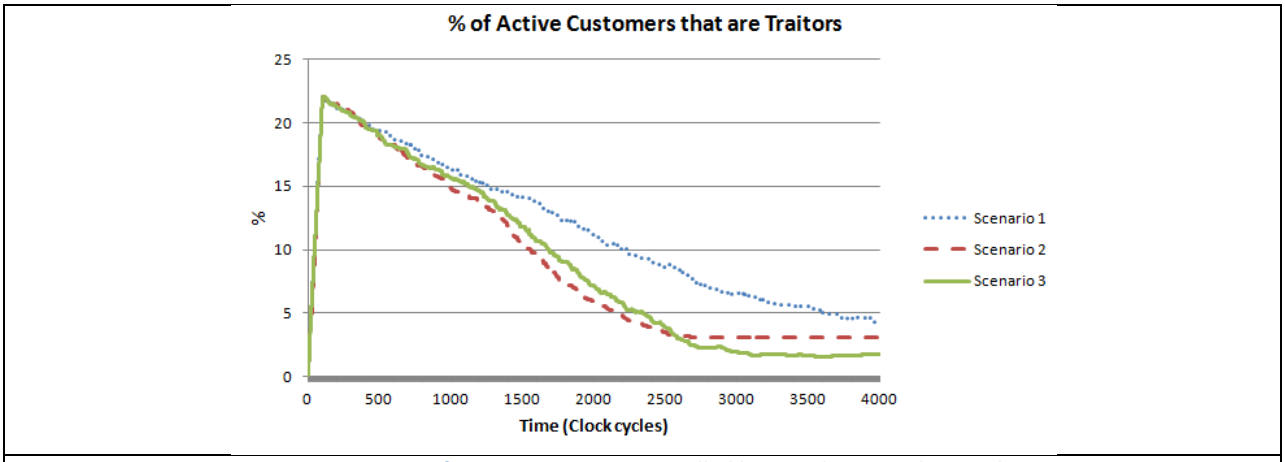


Fig. 33. Percentage of IDACS Customers controlled by Attacker over the simulation

Fig. 34 shows the average number of access-DB-attacks that were successfully passed through IDACS with the help of traitor SAs and SSAs. This chart shows that the most successful time period for the attacker is when he has the maximum number of traitor SAs and SSAs at his disposal (as expected), but that the success rate drops to almost zero as more traitor SAs and SSAs are identified and quarantined. Since Scenario 3 is most representative of a real-world fielded IDACS system, the performance results shown in this graph are encouraging. "5.3 Simulations" will present additional simulations demonstrating how confidential data stored in IDACS can be protected even in the presence of a low number of successful illegal datacenter accesses.

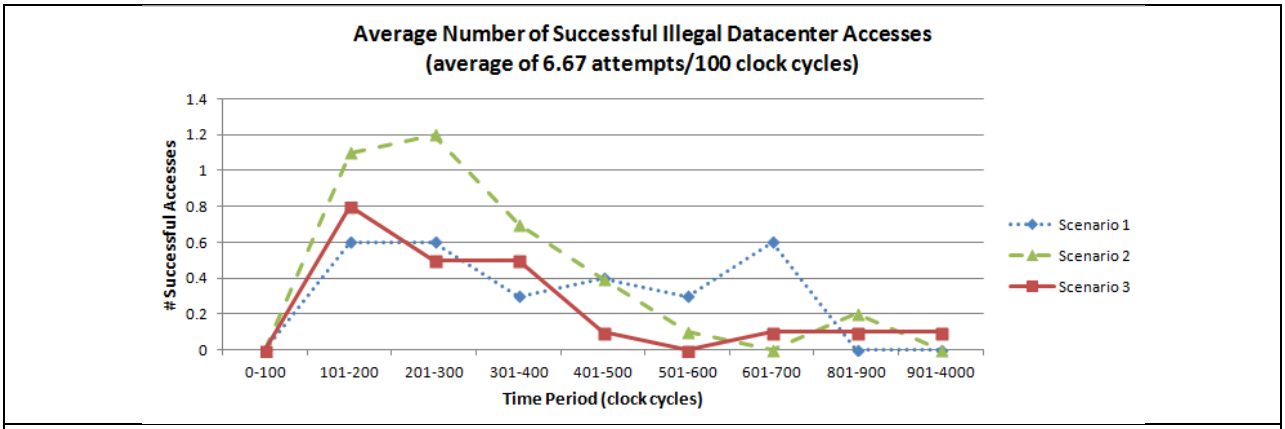


Fig. 34. Average Number of Successful Illegal Datacenter Access Attempts over the Simulation Time Period

5 IDACS Protection of Encrypted Data

5.1 Introduction

In the age of WikiLeaks and the rise of insiders leaking confidential information, it is critical to provide data networks with defenses against this sort of malicious insider behavior. Additionally, with employees losing laptops and mobile devices containing proprietary information with alarming frequency, it is important to minimize the effects of memory-scraping and unauthorized information access. IDACS leverages the space-time separated and jointly-evolving relationship to defend against these types of leaks of at-rest data. It also provides detection and accountability for the sources of data leaks. By separating encrypted data into pieces that are useless by themselves and storing them in separate and time-changing locations, IDACS can greatly increase the security of stored data. This section introduces the principles and methods by which IDACS provides this data security, and it will provide proofs for the mathematical strength of these methods. Additionally, simulations will demonstrate the real-world effectiveness of such a system, even in the presence of a high number of insider traitors.

5.2 Space-Time Protection of Encrypted Data

5.2.1 Separation of Encryption Keys

Definition 23 discusses the concept of cryptographic seeds $Seed_\sigma$ and explains that $Seed_\sigma$ are spread across different locations (i.e. $Seed_\sigma \Diamond Client_\rho$, $\overline{\boldsymbol{Seed}} \Diamond Client_\rho$, etc.). Algorithm 2 and Algorithm 3 demonstrate how these $Seed_\sigma$ are collected from across these locations and combined in order to calculate $\overline{\boldsymbol{OTP}_{\boldsymbol{\psi}}}$ and $\overline{\boldsymbol{PID}_{\boldsymbol{\psi}}}$. The cryptographic keys used to encrypt data residing on $Client_\rho$, similarly to these $Seed_\sigma$, are split into pieces and spread across different locations. However, these cryptographic keys have an additional space-separation protection; certain bits are removed from these cryptographic keys and stored on the SAs in the IDACS Network. In order to decrypt data residing on $Client_\rho$, these bits must be retrieved and reassembled with the cryptographic keys. These bits may be formally defined:

Definition 35: The cryptographic keys used to encrypt data residing on $Client_\rho$ have a certain number of bits removed and stored in a different location. These bits are termed *Xbits*, corresponding to the relevant $Cust_\psi$.

When the Xbits are removed from the cryptographic keys, one bit is removed from each byte of the cryptographic key (Fig. 35). These bits are removed from pseudo-random locations in each byte; the locations from which bits are removed (and conversely, the locations where the Xbits should be reinserted when the cryptographic key is being reassembled) are calculated based on the value of $\mathcal{S}Cust_\psi$; the locations of the removed Xbits will be different for each cryptographic key. As a consequence of this

arrangement, an attacker who manages to steal the contents of $Cust_{\psi}$ will still be unable to decrypt data residing on $Client_{\rho}$ without retrieving the corresponding Xbits from the IDACS Network side.

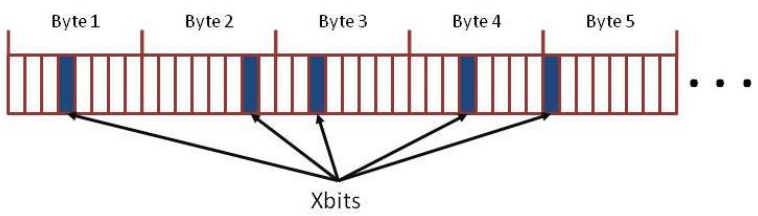


Fig. 35. Xbits removed from the cryptographic keys

Each time the cryptographic key is used for encryption or decryption, $Client_{\rho}$ reforms the cryptographic keys and calculates new Xbit locations base on the updated version of $\mathcal{S}Cust_{\psi}$. As such, an attacker who manages to derive the Xbit insertion locations for a given cryptographic key at time *t* will not possess the correct Xbit insertion points at a later time *t'* after $\mathcal{S}Cust_{\psi}$ has been adjusted. Thus, the space-separated time-evolving relationship is used to protect the integrity of cryptographic keys. This difficulty faced by the attacker is summarized in the following Theorem, which is proved in "5.2.4 Mathematical Proofs".

Theorem 3 : An attacker who possesses a cryptographic key and the corresponding Xbits, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the Xbit insertion points, faces an NP-complete problem to determine the Xbit insertion points.

5.2.2 Separation of Encrypted Data

5.2.2.1 Concept

The IDACS system is designed to store and protect data or services in the IDACS datacenter. However, this design can also be used to help protect data stored on Client devices. Data stored on $Client_{\rho}$ is encrypted using encryption keys stored across multiple locations ($Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, and PIN_{λ}); this guarantees that an attacker must have access to all of these items in order to decrypt the data. Additionally, pieces of the ciphertext are removed and stored in the IDACS datacenter (Fig. 36). These pieces must be retrieved from the datacenter in order to correctly decrypt the data stored on $Client_{\rho}$.

Definition 36: When encrypted data is stored on $Client_{\rho}$, segments of data are removed from the ciphertext and stored in a physically different location. These removed segments are called *Xslices*.

All of the Xslices that are removed from a Client-side ciphertext are stored in the IDACS datacenter (Fig. 36). The Xslices may be stored in a single contiguous block (Storage Method 1), or they may be split and stored across multiple locations (Storage Method 2).

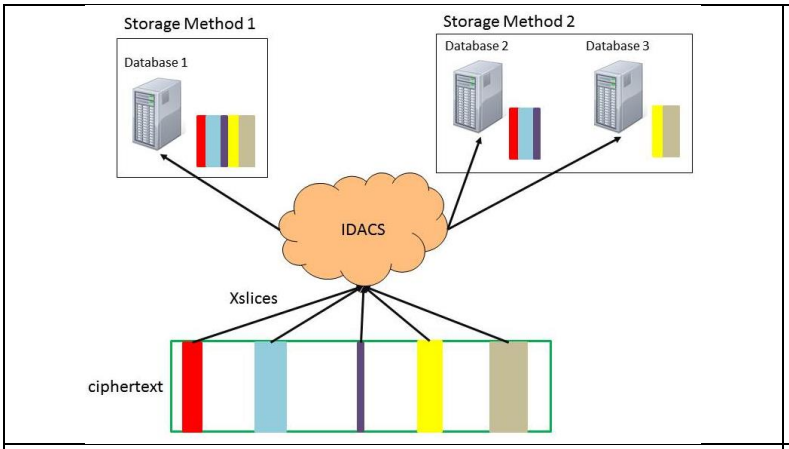


Fig. 36. Xslices removed from Client-side ciphertext and stored in IDACS datacenter

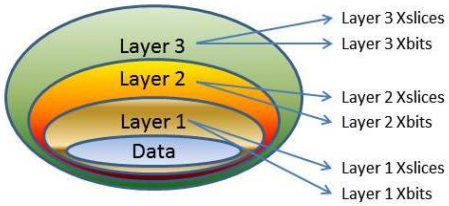


Fig. 37. Multiple layers of encryption

In order to decrypt data stored on $Client_\rho$, one must have access to the ciphertext stored on $Client_\rho$; all of the locations storing pieces of the encryption keys ($Client_\rho$, $Badge_\zeta$, Pwd_θ, and PIN_λ); the Xbits for the encryption keys, which are stored across multiple SAs in IDACS; and the Xslices that are stored across multiple DB in the IDACS datacenter. Additionally, each time the data stored on $Client_\rho$ is decrypted to be viewed, the value of $\mathcal{S}Cust_\psi$ is updated, and the data is re-encrypted with new cryptographic keys that have new Xbits, and new Xslices are removed from the ciphertext and stored at new locations in the datacenter. By combining space-separation and time-evolving charactersitics, this IDACS encryption scheme can achieve a much higher level of security than simple encryption.

As a final security measure, the encryption/Xbits/Xslices can be applied in multiple layers to protect high-sensitivity data (Fig. 37). In addition to providing more complex security, this provides additional options for space-time evolving protections. At certain time intervals, the top level of encryption can be re-processed with new encryption keys/Xbits/Xslices, while the lower levels are left untouched; in this way, the security is time-evolving with a minimum of effort.

5.2.2.2 Implementation

The location and length of the Xslices in the ciphertext are pseudorandom; they are calculated based on $\mathcal{S}Cust_\psi$, according to Definition 37 and Definition 38. This pseudorandomness contributes to the strength of the IDACS encryption, as addressed in Theorem 4 through Theorem 6.

Definition 37 : Given $\mathcal{S}Cust_\psi$, a block of ciphertext to have Xslices removed, and the PID of that data block, the *F-box(data-block-offset)* transform returns the length between the beginning of the block of data or the end of the previous xslice, and the beginning of the next xslice. This transform also updates $\mathcal{S}Cust_\psi$ so that the next call to the transform will return the length of the next sub-block. This transform must produce the same sequence of lengths for consecutive

transform calls for a given data block PID after $\mathcal{S}Cust_\psi$ has been reinitialized, so that data blocks may be disassembled and reassembled. The sub-block lengths are determined based on the cryptographic hash of secret seeds stored in $\mathcal{S}Cust_\psi$. This transform is represented by

local_data_block_length = F-box($\mathcal{O}$ffset, $\mathcal{S}Cust_\psi$)

Definition 38 : Given $\mathcal{S}Cust_\psi$, a block of data to have xslices removed, and the PID of that data block, the *F-box(xslice-length)* transform returns the length of the next xslice to be removed from the data block. This transform also updates $\mathcal{S}Cust_\psi$ so that the next call to the transform will return the length of the next xslice. This transform must produce the same sequence of lengths for consecutive transform calls for a given data block PID after $\mathcal{S}Cust_\psi$ has been reinitialized, so that data blocks may be disassembled and reassembled. The xslice lengths are determined based on the cryptographic hash of secret seeds stored in $\mathcal{S}Cust_\psi$. This transform is represented by

local_xslice_length = F-box($\mathcal{X}$Lth, $\mathcal{S}Cust_\psi$)

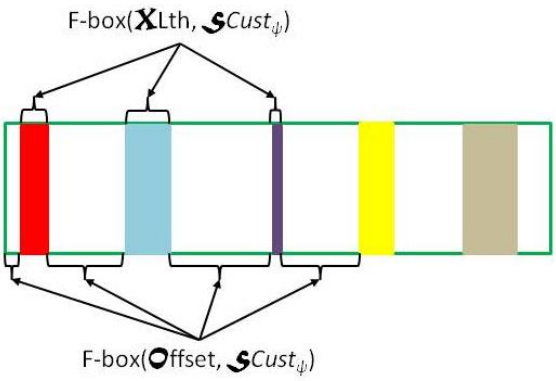


Fig. 38. Using F-box($\mathcal{O}$ffset) and F-box($\mathcal{X}$Lth) transforms to remove Xslices

Fig. 38 demonstrates how these transforms are used to divide the ciphertext into a block of Xslices and a block of ciphertext. This process is formalized in Algorithm 11, with the addition of another F-box transform defined in Definition 39. Finally, the entire multiple-layer encryption process illustrated in Fig. 37 is formalized in Algorithm 12, which also references Definition 40.

Definition 39 : Given an input string or byte array, the *F-box(substring)* transform returns a substring or sub-array based on specified indices. The transform is represented by

local_xslice = F-box($\mathcal{SS}$tring, data_block, offset, length)

The "data_block" parameter is the input string or byte array, the "offset" parameter is the index indicating where in "data_block" the desired substring begins, and the "length" parameter indicates the length of the desired substring.

Definition 40 : Given a block of data and $\mathcal{S}Cust_{\psi}$, the *F-box(encrypt)* transform encrypts the block of data using encryption keys provided by $\mathcal{S}Cust_{\psi}$ and returns the ciphertext along with the updated version of $\mathcal{S}Cust_{\psi}$. This transform is represented by

$$\{ \mathcal{S}Cust_{\psi}, \text{ciphertext} \} = \text{F-box}(\mathcal{E}\text{ncrypt}, \mathcal{S}Cust_{\psi}, \text{data\_block})$$

Algorithm 11. remove_xslices()

```
inputs : SCust_ψ, data_block
outputs : SCust_ψ, data_block', xslices

1     pointer = index of 1st byte of data_block
2     xslices = empty string
3     data_block' = empty string

4     while (pointer < data_block.length) do

5         local_data_block_length = F-box(Offset, SCust_ψ)
6         local_xslice_length = F-box(XLth, SCust_ψ)
7         local_data_block = F-box(SString, data_block, pointer, local_data_block_length)
8         local_xslice = F-box(SString, data_block, [pointer + local_data_block_length], local_xslice_length)

9         xslices = F-box(Concat, xslices, local_xslice)
10        data_block' = F-box(Concat, data_block', local_data_block)

11        pointer = pointer + local_data_block_length + local_xslice_length

      end
```

Algorithm 12. encrypt_data()

```
inputs : SCust_ψ, data_block, num_layers
outputs : SCust_ψ, ciphertext, xslices

1     ciphertext = data_block
2     xslices = empty string

3     for index = 1 to num_layers
4         { SCust_ψ, ciphertext } = F-box(Encrypt, SCust_ψ, ciphertext)
5         { SCust_ψ, ciphertext, temp_xslices } = remove_xslices(SCust_ψ, ciphertext)
6         xslices =  F-box(Concat, xslices, temp_xslices)
      end
```

5.2.2.3 Theoretical Implications

The use of Xslices in the IDACS Client-side data encryption scheme leads to several theoretical implications which demonstrate the security of this encryption scheme. First, consider the situation where the Xslices extracted from a given piece of data's ciphertext are stored in a contiguous block in a single location (Storage Method 1 in Fig. 36). Alternatively, when $Cust_{\psi}$ requests these Xslices from the IDACS datacenter, they are returned to $Cust_{\psi}$ in a single block of data. An attacker who manages to retrieve the ciphertext and the block of corresponding Xslices, but does not possess $\mathcal{S}Cust_{\psi}$ and thus cannot perform the F-box(Offset) or F-box(XLth) transforms in Algorithm 11, is faced with the problem of determining 1) where the ciphertext splits for the insertion of Xslices, and 2) where the contiguous block of Xslices splits into individual Xslices. The difficulty of 1) is addressed in Theorem 4, and the difficulty of 2) is addressed in Theorem 5.

Theorem 4 : An attacker who possesses a ciphertext block requiring Xslice insertion, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the Xslice insertion points, faces an NP-complete problem to determine the Xslice insertion points.

Theorem 5 : An attacker who possesses a block of concatenated Xslices extracted from a ciphertext, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the lengths of and separated individual Xslices, faces an NP-complete problem to separate the individual Xslices.

A second situation, where individual Xslices are stored across multiple DB in the IDACS datacenter (Storage Method 2 in Fig. 36), presents a similar problem. Consider an attacker who is able to retrieve all the individual Xslices associated with a certain piece of Client-side data's ciphertext. However, without access to $\mathcal{S}Cust_{\psi}$, the attacker is unable to determine the correct order in which these Xslices should be arranged for reinsertion into the ciphertext. This problem is addressed in the following Theorem.

Theorem 6 : An attacker who possesses all Xslices extracted from a ciphertext, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the order in which these Xslices should be re-inserted into the ciphertext, faces an NP-complete problem to correctly order the individual Xslices.

The preceding Theorems are proved in "5.2.4 Mathematical Proofs".

5.2.3 Data Segmentation

The previous section described how Xslices are used to protect the confidentiality of encrypted Client-side data. This section will examine how the used of distributed Xslices can be combined with data segmentation gain additional security by

minimizing the level of decrypted data exposure and minimize the damage caused by an attacker who is able to successfully pass several illegal IDACS datacenter access requests.

5.2.3.1 Single File

The standard approach to file encryption and decryption is to decrypt an entire protected file at the time of access. Unfortunately, this exposes the entire contents of the protected file to an attacker who can steal a $Client_p$ on which a currently decrypted file is being viewed. The concept of the space-time evolving relationship can be used to minimize this risk. Rather than encrypting a file in a single "block", the file can be divided into multiple "segments" (e.g. one page of the file equates to one segment). A "navigation" file associated with the encrypted data file is formed; this navigation file contains metadata regarding each segment in the data file, such as where each segment begins and ends in the ciphertext, which Xslices are used in that segment, and where those Xslices are inserted into the ciphertext (Fig. 39). When a user wants to view part of the encrypted file, the contents of the navigation file are presented as a "Table of Contents". The user selects the segment he wishes to view, and $Cust_\psi$ requests the Xslices for the specified segment, inserts them into the ciphertext, and decrypts that particular segment. The remaining segments in the file are not decrypted unless they are specifically accessed later.

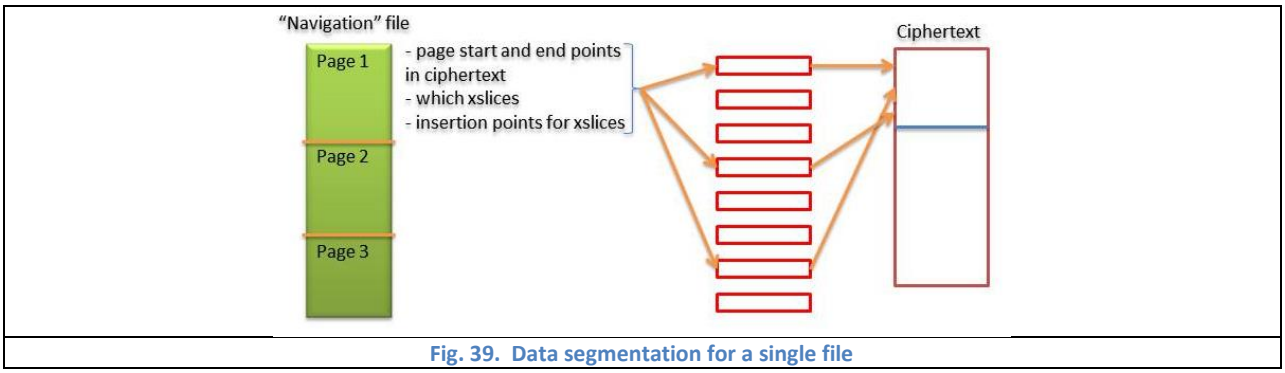


Fig. 39. Data segmentation for a single file

Segmenting an encrypted data file in this manner enhances data security in several ways. First, in the event that a $Client_p$ being used to decrypt and view data is stolen, the amount of decrypted data residing on $Client_p$ is limited. Additionally, if an attacker is able to force a few illegal IDACS datacenter access requests through IDACS, the encrypted data that attacker can recover is limited to a few file segments rather than the same number of files.

Segmenting data on the single file-level provides benefits in terms of both security and performance, which are summarized in Table 6. If a single segment of a file is decrypted, the number of Xslices retrieved from the datacenter as well as the amount of decrypted plaintext exposed is less than if the encrypted file were non-segmented. Additionally, the time required to complete this operation is constant (O(1)) rather than linear (O(x)). If all segments of the file are decrypted, then there is no relative advantage over a non-segmented approach.

Table 6. Comparison of segmented vs. non-segmented data encryption for file of length *x*.

	Non-segmented file	Segmented file (decrypting one segment)	Segmented file (decrypting entire file)
Performance			
Time to retrieve Xslices	Request Xslices: O(1) Send XSlices: O(*x*) Insert Xslices: O(*x*) Total: O(x)	Request Xslices: O(1) Send XSlices: O(1) Insert Xslices: O(1) Total: O(1)	Request Xslices: O(*x*) Send XSlices: O(*x*) Insert Xslices: O(*x*) Total: O(x)
Time to decrypt	O(*x*)	O(1)	O(*x*)
Total time	O(*x*)	O(1)	O(*x*)
Security			
Xslices exposed	O(*x*)	O(*1*)	O(*x*)
Decrypted plaintext exposed	O(*x*)	O(*1*)	O(*x*)

The results shown in Table 6 may be used as justification for the following:

Claim 6: Segmenting encrypted files and decrypting and issuing Xslices one segment at a time increases security and performance if one or a few pages are decrypted, but has no effect on security or performance if an entire file is decrypted.

5.2.3.2 File Directory Tree

In the previous section, it was shown how data segmentation could be used to protect a single encrypted data file; the same concept can also be used to protect and encrypt a file directory tree. Consider the file directory tree shown in Fig. 40. The different levels of folders in Fig. 40 correspond to the "navigation" file in Fig. 39; they are files that do not contain actual data, but only pointers to the actual data files. The actual data files that are the leaf nodes of this tree correspond to the ciphertext segments in Fig. 39. In Fig. 40, each Zone is a separate file with its own Xbits and Xslices residing in the IDACS datacenter, and each leaf node data file also contains its own Xbits and Xslices. Using this tiered encrypted File Directory Tree is ideal for a situation where encrypted data is maintained on the Client-side with Xbits and Xslices stored on IDACS Databases. Decrypting the File Directory Tree requires that Xbits and Xslices be retrieved to decrypt each successive Zone. If an attacker is able to pass a few access-DB-attacks, he may be able to gather some information on the structure of the File Directory Tree, but it may not be enough to recover any of the actual data files. Additionally, this structure obfuscates information regarding the size,

quantity, and organization of the data files in the File Directory Tree by minimizing the amount of file pointers and file data that are exposed during a single file access, as will be seen in the following example. Obfuscating this information also minimizes the number of targets an attacker can address.

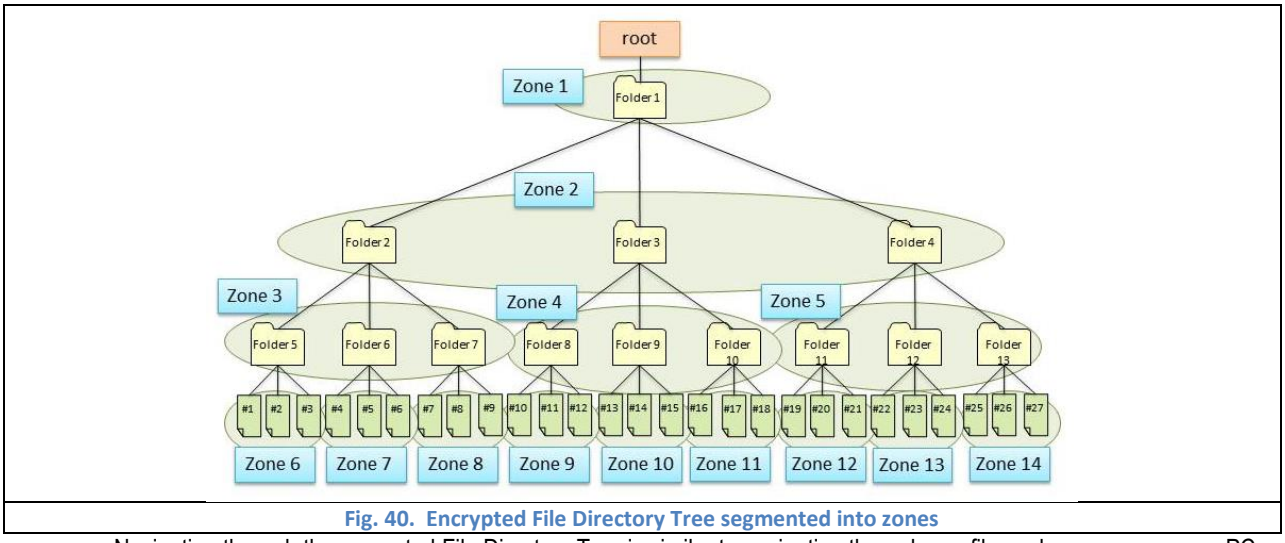


Fig. 40. Encrypted File Directory Tree segmented into zones

Navigating through the encrypted File Directory Tree is similar to navigating through any file explorer program on a PC. Consider Fig. 41; a user initially possesses encrypted pointer information regarding the "root" of the File Directory Tree. The user requests the Xbits and Xslices to decrypt the "root" file residing on $Cust_{\psi}$, revealing the contents of Zone 1 (Folder 1). The user then requests the Xbits and Xslices to decrypt the Folder 1 pointer file (Zone 2), revealing the children folders of Folder 1 (Folders 2, 3, and 4). The user proceeds through Zone 5 and Zone 14 to reach the target File 25. Through this process, only information regarding folders and their children along the direct path to the target (File 25) are revealed; information regarding unexplored folders is not revealed to the user.

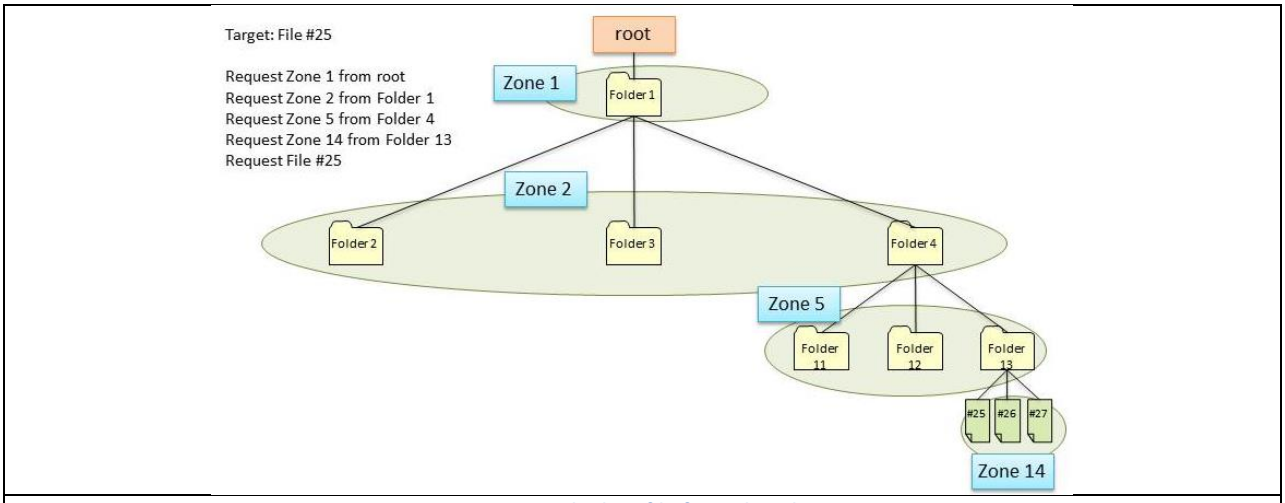


Fig. 41. Retrieving a single data file from the File Directory Tree

Table 7 illustrates the performance of a non-segmented File Directory Tree compared to the segmented version. The performance of the segmented version exceeds that of the non-segmented version if a single file is retrieved; however, the performance of the segmented version drops if all of the files in the Directory Tree are retrieved. For its application in IDACS, this tradeoff in performance is considered acceptable in return for the corresponding increase in security, which is demonstrated in Table 8.

Table 7. Comparison of performance of segmented vs. non-segmented File Directory Trees containing *x* data files

	Non-segmented File Directory Tree	Segmented File Directory Tree
Time to locate a single file	Request File Directory Tree: O(1) Send File Directory Tree: O(*x*) Decrypt File Directory Tree: O(*x*) Total: O(*x*)	Request Zone: O(1) Send Zone: O(1) Decrypt Zone: O(1) Repeat for the depth of the File Directory Tree: O(log *x*) Total: O(log *x*)
Time to decrypt a single file	O(1)	O(1)
Total for a single file	O(*x*)	O(log *x*)
Total for all files in File Directory Tree	O(*x*) (because File Directory Tree only needs to be retrieved once)	O(*x* log *x*) (because potentially entire depth of File Directory Tree must be retrieved for each file)

Table 8 compares the security provided (in terms of how much file data and pointers are exposed) for non-segmented and segmented File Directory Trees. If a single file is accessed, the segmented version provides higher security by not exposing the file data and pointers for non-accessed files; of course, this advantage is lost if all of the files in the directory tree are accessed. In either case, the segmented version provides a higher level of security by forcing more authentication and permissions checks by a factor of log *x*. Since the user must potentially navigate the depth of the File Directory Tree for each file accessed, retrieving Xbits and Xslices from the IDACS datacenter for each zone accessed, the segmented version forces more $\overline{\boldsymbol{OTP}_{\psi}}$ and $\overline{\boldsymbol{PID}_{\psi}}$ checks, making a traitor $Cust_{\psi}$ controlled by an attacker more likely to be detected.

Table 8. Comparison of security of segmented vs. non-segmented File Directory Trees containing x data files

	Non-segmented File Directory Tree	Segmented File Directory Tree
For a single file access		
How many files exposed	O(1)	O(1)
How many file pointers (leaf nodes) exposed	O(*x*) (all file pointers exposed)	O(1) (only files in folder containing target file)
How many folder pointers exposed (by decrypting zones)	O(*x*) (all folder pointers exposed)	O(log *x*)
How many authentication/ permissions checks	O(1)	O(log *x*)
For all files in Directory Tree accessed		
How many files exposed	O(*x*)	O(*x*)
How many file pointers (leaf nodes) exposed	O(*x*) (all file pointers exposed)	O(*x*) (all file pointers exposed)
How many folder pointers exposed (by decrypting zones)	O(*x*) (all folder pointers exposed)	O(*x*) (all folder pointers exposed)
How many authentication/permissions checks	O(*x*)	O(*x* log *x*)

The results displayed in Table 7 and Table 8 may be taken as justification for the following claims.

Claim 7: Segmenting the File Directory Tree and allowing a user to decrypt one zone at a time, as compared to a File Directory Tree system that provides the entire directory tree at once, for a single file access in a tree containing *x* files,

a) Improves the efficiency of retriving a single file from O(*x*) to O(log *x*)

b) Improves security by reducing the number of exposed file pointers from O(*x*) to O(1)

c) Improves security by increasing the number of required authentication/permissions checks from O(1) to O(log *x*)

Claim 8: Segmenting the File Directory Tree and allowing a user to decrypt one zone at a time, as compared to a File Directory Tree system that provides the entire directory tree at once, for accessing every file in a tree containing *x* files,

a) Reduces the efficiency of retrieving all files from O(*x*) to O(*x* log *x*)

b) Improves security by increasing the number of required authentication checks from O(x) to O(x log x)

5.2.4 Mathematical Proofs and Analysis

This section provides the mathematical proofs for Theorem 3 through Theorem 6.

5.2.4.1 Theorem 3, Theorem 4, and Theorem 5

First, a short review of the Theorems to be proved:

Theorem 3 : An attacker who possesses a cryptographic key and the corresponding Xbits, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the Xbit insertion points, faces an NP-complete problem to determine the Xbit insertion points.

Theorem 4 : An attacker who possesses a ciphertext block requiring Xslice insertion, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the Xslice insertion points, faces an NP-complete problem to determine the Xslice insertion points.

Theorem 5 : An attacker who possesses a block of concatenated Xslices extracted from a ciphertext, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the lengths of and separated individual Xslices, faces an NP-complete problem to separate the individual Xslices.

All three cases represent a "splitting" problem, where a block of data must be split at certain points in order to re-insert extracted information (Theorem 3 and Theorem 4) or to separate the extracted information into pieces for re-insertion (Theorem 5). In essence, the problem requires the attacker to recreate the sequence of outputs from repeated calls to the F-box($\mathcal{X}$Lth) or F-box($\mathcal{O}$ffset) transforms, as demonstrated in Fig. 38. Although the outputs of these transforms are calculated based on the value of $\mathcal{S}Cust_{\psi}$, in a fielded IDACS system, the lengths of individual Xslices or the data blocks in the ciphertext between Xslices are

chosen from a finitely long list of known possible lengths. In the case of Xbit insertion, there are a finite number of possible insertion points in each byte for the corresponding Xbit (8 possible positions). Therefore, solving these problems requires recreating a sequence of numbers (lengths) drawn from a known, finite list.

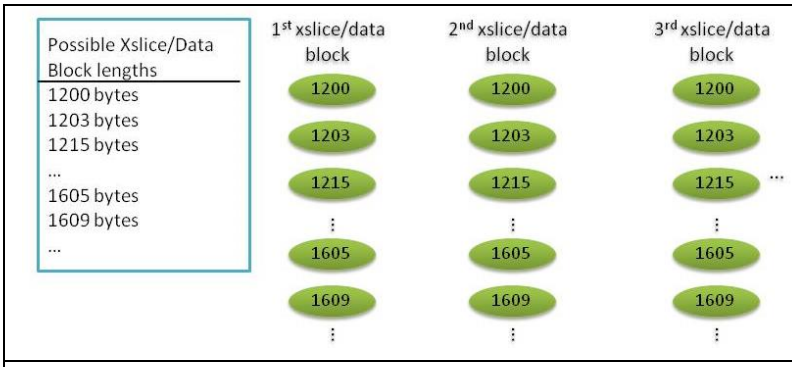


Fig. 42. Xslice/Data Block splitting problem

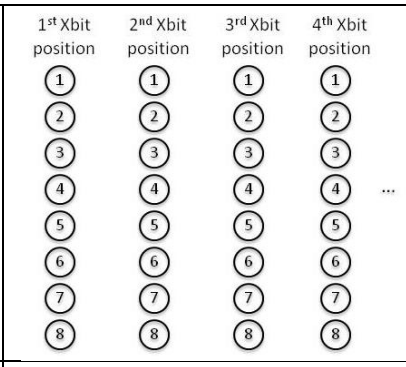


Fig. 43. Xbits splitting problem

Fig. 42 and Fig. 43 demonstrate these problems graphically. To solve these problems, one member from each column must be selected, with the final sequence of selections representing the actual division of Xslices/Data Blocks or insertion points for Xbits. These problems can be represented in terms of graph theory. Let the options in each column be represented by a set of vertices $\mathbf{V}$. Consistent with the concept of "graph coloring", each vertex in the same column is assigned the same color, with different colors assigned to each column (represented by shapes in Fig. 44). Each $v \epsilon \mathbf{V}$ is connected by an directed edge e, $e \epsilon \mathbf{E}$, to every other v in an adjacent column (in Fig. 44, only a few e are shown for the sake of simplicity). Each $e \epsilon \mathbf{E}$ has an associated edge weight $\mathbf{W}(e)$, $0 \leq \mathbf{W}(e) \leq 1$, where $\mathbf{W}(e)$ represents the probability that the $\{v_1, v_2\}$ connected by e, with their respective values and colors, are both present in a path which contains one v of each color that represents the correct sequence Xslice lengths etc. (the method for determining these weights will be discussed in "5.2.4.3 Randomness in Xslices") Theoretically, the path containing one v of each color that has the highest sum $\mathbf{W}(e)$ of all such paths (i.e the Maximum Weight Path) should represent the correct sequence of Xslice lengths etc. (Fig. 45)

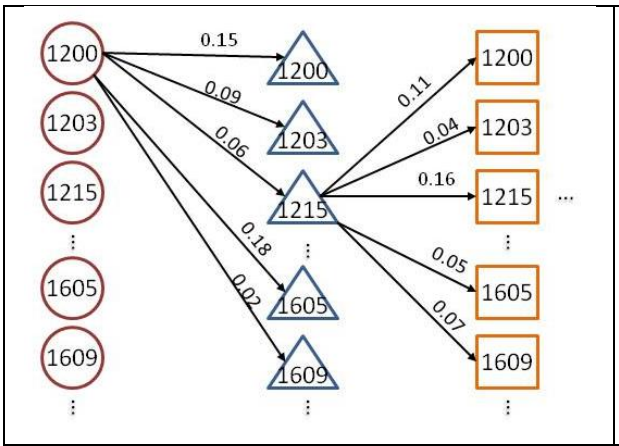


Fig. 44. Splitting problem represented in terms of graph theory.

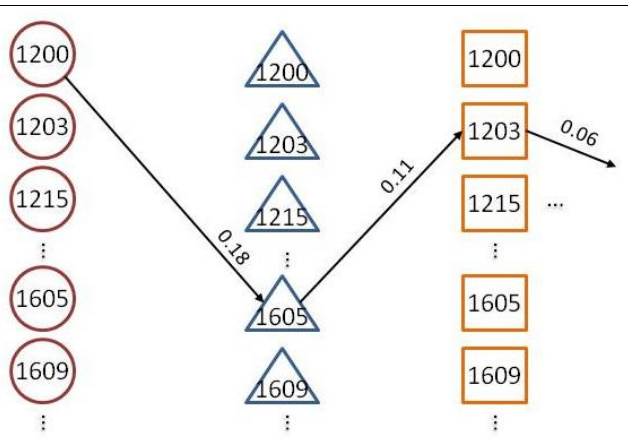


Fig. 45. Maximum Weight Path.

Now, solving the problem posed in Theorem 3, Theorem 4, or Theorem 5 is equivalent to solving this Maximum Weight Path problem. This problem may be formalized by specifying that a path of length Z (where Z is the number of columns) must be found. This is now the Maximum Weight Directed Path of Specified Length (MWDPSL) problem, which is proved NP-Complete in "Appendix A". Thus, the NP-Completeness of Theorem 3, Theorem 4, and Theorem 5 is proved.

5.2.4.2 Theorem 6

Theorem 6 : An attacker who possesses all Xslices extracted from a ciphertext, but does not possess the $\mathcal{S}Cust_{\psi}$ necessary to determine the order in which these Xslices should be re-inserted into the ciphertext, faces an NP-complete problem to correctly order the individual Xslices.

The proof for Theorem 6 is identical to the proof for Theorem 1, with the $Seed_{\sigma}$ in Theorem 1 replaced by the Xslices in Theorem 6. The Xslice ordering problem in Theorem 6 may also be represented by the Maximum Weight Path of Specified Length (MWPSL) problem, which is proved NP-Complete in "Appendix A". Thus, the NP-Completeness of Theorem 6 is proved.

5.2.4.3 Randomness in Xslices

While Theorem 3, Theorem 4, Theorem 5, and Theorem 6 have been proved NP-complete, the value of this proof must be qualified. NP-completeness speaks only to the worst-case complexity of a given decision problem (which is that the complexity grows exponentially with the problem size); there may be other factors that can significantly reduce the complexity of a problem. If the edge weights in the graph are relatively uniform (Fig. 46), then the complexity of finding the Maximum Weight Path is close to the worst-case scenario; however, if the edge weights are not relatively uniform (Fig. 47), then a simple algorithm (or a human analyst) can significantly reduce the complexity of finding the Maximum Weight Path by picking out the high-weight edges that are more likely to be part of the solution.

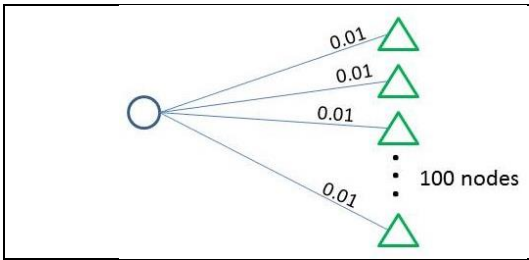


Fig. 46. Graph with uniform edge weight distribution

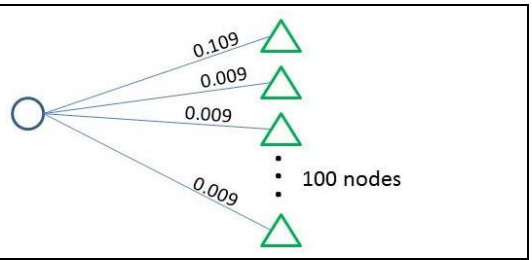


Fig. 47. Graph with non-uniform edge weight distribution

Consider the example of reassembling fragmented data as discussed in [31]. This paper discusses a method of reassembling fragmented data by using the Maximum Weight Path problem with the weights of edges between data fragment nodes based on recognized patterns in the data. Therefore, highly-patterned data will result in stronger pattern recognition, which will result in a graph with a few high-weight edges. Therefore, highly-patterned data will result in a Maximum Weight Path

reassembly problem that has a complexity significantly less than the worst-case exponential. In [31], the authors discuss this phenomenon, and show through their experiments that highly-patterned data does indeed lead to faster and more accurate file reassembly. So the question is, does the fragmented, distributed ciphertext (Xslices and their associated ciphertext) that is present in IDACS produce a uniform or non-uniform edge weight distribution in the graph?

In order to answer this question, it is necessary to analyze the "randomness" of the type of ciphertext fragments that will be present in IDACS in order to judge what type of edge weight distribution a Maximum Weight Path model applied to these fragments would generate. This analysis was performed using a software package created by the National Institute of Standards and Technology (NIST) [32], which provides a battery of tests that analyzes the outputs of Random Number Generators (RNGs) to measure their "randomness" by looking for patterns in the outputs [33]. The battery consists of 15 individual tests, each of which measures different aspects of "randomness" in the data. Each of these tests ask the question: "If the algorithm that generated this data sample was truly random, what is the probability that this specific data sample could have been generated?" The individual tests respond with a p-score in the probability range [0, 1]. The NIST standard [33] recommends using a passing score, or "significance level", of 0.01. Some truly random data samples will fail the tests and generate a "false positive" for randomness due to weaknesses in the test; therefore, two types of statistics are recommended for analyzing the test p-scores.

The first statistic looks at the proportion, or percentage, of tests with passing p-scores According to the parameters in [33], for a set of tests run with 1000 data samples, a truly random RNG will have a minimal proportion of 0.9805068, i.e. a minimum pass rate of 98.05%. The second statistic looks at the distribution of the p-scores. For a set of truly random data samples, it is expected that the p-scores should be evenly distributed. The evenness of the distribution can be measured by calculating the $P\text{-value}_T$ for each test based on the chi-square statistic discussed in [33]; if each test has a P-valueT ≥ 0.0001, then the p-scores are considered to be evenly distributed.

To measure the randomness of ciphertext blocks, the NIST battery was applied to two sets of data. The first set of data consisted of samples of normal AES ciphertext blocks (representing a segment of AES ciphertext with the correct Xslice re-inserted), and the second set consisted of samples of two normal AES ciphertext blocks encrypted with two different AES keys back-to-back with each other (representing a segment of AES ciphertext with an incorrect Xslice re-inserted). This test was designed to determine whether there was any discernible difference between the "pattern" (or randomness) of a correctly- and incorrectly- re-inserted Xslice. Both data sets consisted of 1000 samples, each of which was 10^6 bits ($1.25 * 10^5$ bytes) long. The samples in the first data set consisted of plaintext encrypted using AES in CBC mode, using a unique key for each sample.

The samples in the second data set consisted of the same plaintext encrypted using AES in CBC mode, but each sample was split into two halves, each of which was encrypted using a unique key).

The NIST battery of tests consists of 15 individual tests. Two of these tests are run twice during the course of the battery; results for both of the tests are reported here. Three of the tests are run a number of times; results for two randomly selected instances of those tests are reported here. All other tests were run once; the results are reported here. In total, 20 separate test results are reported.

Fig. 48 shows the proportion of passing NIST tests for the first data set (Fig. 48 (a)), which represents a "matched" Xslice and ciphertext, and the second data set (Fig. 48 (b)), which represents a "mismatched" Xslice and ciphertext (or two concatentated Xslices or ciphertext segments that were not adjacent in the original ciphertext). It can be seen that all but one test result pass the minimum proportion requirements; more importantly, both data sets are demonstrated to be "random", and there is no distinguishable difference between the proportions for the two data sets.

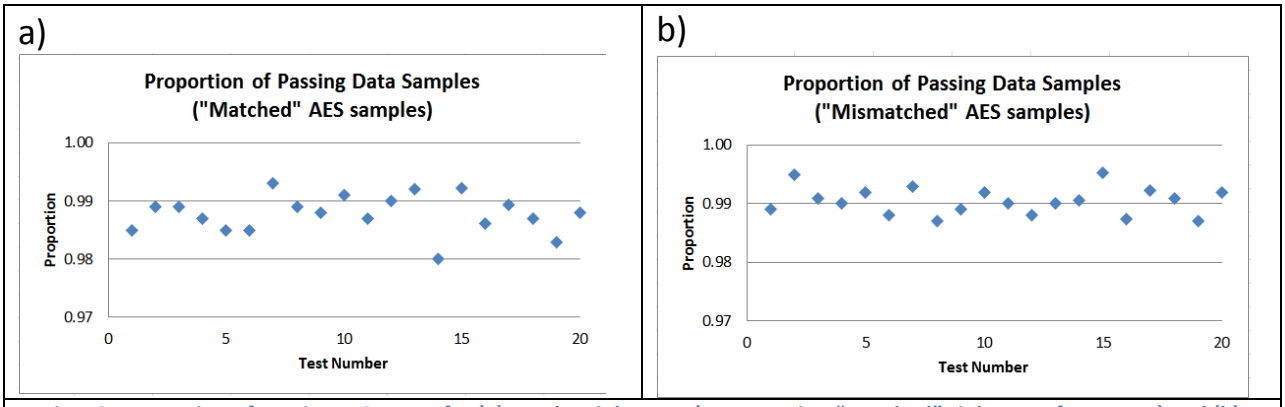


Fig. 48. Proportion of passing NIST tests for (a) regular ciphertext (representing "matched" ciphertext fragments) and (b) dual-AES key ciphertext (representing "mismatched" ciphertext fragments)

Table 9 lists the P-value$_T$ for all of the NIST tests for both the "matched" and "mismatched" ciphertext fragments. It can be seen that all tests for both data sets pass the minimum score of 0.0001. Thus, the two data sets may be considered equally "random". Therefore, it can be concluded that both matching and mismatching ciphertext fragments would generate uniform edge weights in a weighted graph. This indicates that there is no discernible difference (pattern-wise) between adjacent ciphertext blocks (ciphertext joined with associated Xslices) and non-adjacent blocks (concatentated Xslices or ciphertext with Xslices removed), and that the graphs generated to solve Theorem 3 through Theorem 6 would have uniform edge weights, maximizing the effect of the NP-complete property.

Table 9. P-value$_T$ for NIST tests for "matched" ciphertext fragments and "mismatched" ciphertext fragments

Test #	Matched Ciphertext	Mismatched Ciphertext	Test #	Matched Ciphertext	Mismatched Ciphertext	Test #	Matched Ciphertext	Mismatched Ciphertext
1	0.147815	0.514124	8	0.162606	0.570792	15	0.243466	0.291249
2	0.173770	0.325206	9	0.174728	0.647530	16	0.888892	0.356948
3	0.528111	0.605916	10	0.323668	0.153763	17	0.105377	0.148760
4	0.340858	0.689019	11	0.867692	0.574903	18	0.504219	0.948298
5	0.196920	0.626709	12	0.649612	0.794391	19	0.156373	0.081013
6	0.875539	0.500279	13	0.522100	0.344048	20	0.257004	0.705466
7	0.727851	0.039073	14	0.437274	0.974555			

5.3 Simulations

5.3.1 Simulation Parameters

The simulations used to examine the effect of Xbits, Xslices, Single File Data Segmentation, and File Directory Tree Segmentation on increasing IDACS security used the same simulation suite discussed in "4.5.2 Attack Detection, Traceback, and Remediation Simulations". In order to simulate the use of Xbits, Xslices, and Data Segmentation, the simulation was expanded to make the access-DB-attacks directed towards accessing the contents of a Segmented File Directory Tree and the data files it stores. The Segmented File Directory Tree used in these simulations is illustrated in Fig. 49. Each Traitor $Cust_\psi$ in the simulation had legal access (permissions) to access 3 random data files in the File Directory Tree; however, no Traitor $Cust_\psi$ had access to the full set of authentication tokens ($Client_\rho$, $Badge_\zeta$, Pwd_θ, and PIN_λ). Because of this, all access-DB-attacks were illegal. Each Traitor $Cust_\psi$ began with the contents of the File Directory Tree (minus Xslices) encrypted on his $Client_\rho$, with the Xbits and Xslices necessary for decryption residing in the IDACS datacenter. Each Traitor $Cust_\psi$ began with a pointer to the "root" of the File Directory Tree; he would need to retrieve the Xbits and Xslices necessary to decrypt the "root" from the IDACS datacenter in order to access the "root's" contents (the pointers for the Level 1 Folders). He would then need to retrieve the Xbits and Xslices to decrypt a given Level 1 Folder from the IDACS datacenter in order to access that Level 1 Folder's contents (the pointers to the children Level 2 Folders). A Traitor $Cust_\psi$ would need to repeat this process until he was able to retrieve all four Segments of a target Data File.

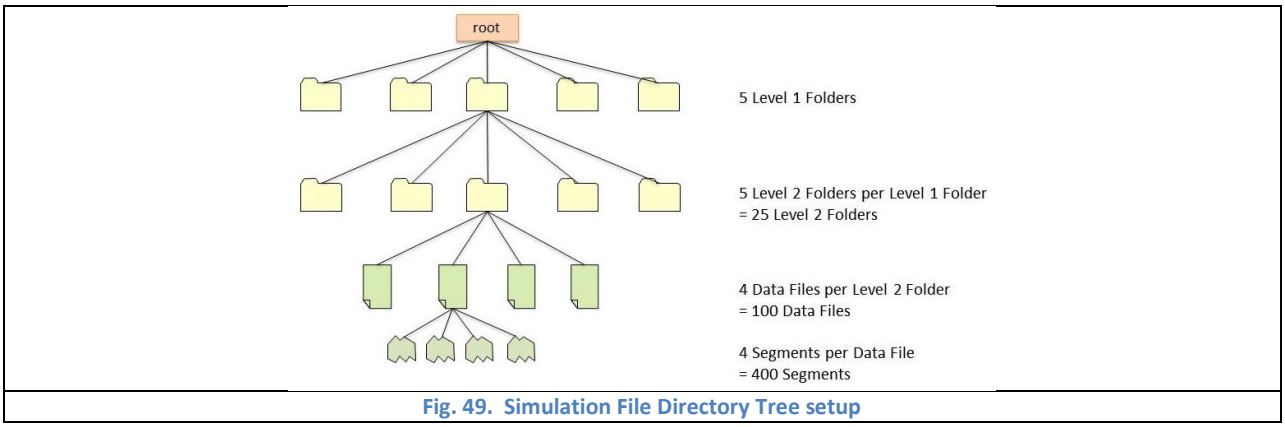


Fig. 49. Simulation File Directory Tree setup

Each Traitor $Cust_{\psi}$ in IDACS possessed the same encrypted File Directory Tree; however, each Traitor $Cust_{\psi}$'s File Directory Tree was encrypted using different encryption keys, different Xslices, and different *Content*($\boldsymbol{PID_{\varepsilon}}$) associated with the store Xbits and Xslices. Therefore, Traitor $Cust_{\psi}$ were not able to collaborate with each other by sharing *Content* ($\boldsymbol{PID_{\varepsilon}}$) and thus "skipping over" the retrieval of a given folder; all Traitor $Cust_{\psi}$ were forced to completely decrypt their own File Directory Trees. Additionally, each Traitor $Cust_{\psi}$ contained only the portions of the File Directory Tree that were "ancestors" of the Data Files that particular $Cust_{\psi}$ had permissions to access; in this simulation, Traitor $Cust_{\psi}$ were unable to access files for which they did not have permissions. However, the group of Traitor $Cust_{\psi}$ was able to collaborate with each other in a limited way; if one Traitor $Cust_{\psi}$ had retrieved a particular Data File Segment, all other Traitor $Cust_{\psi}$ would seek to retrieve other Data File Segments, rather than pursue data that had already been successfully recovered. Additionally, the group of Traitor $Cust_{\psi}$ would give priority for retrieval attempts to the Traitor $Cust_{\psi}$ with the deepest exploration into the File Directory Tree, thus focusing the resources of Traitor SAs and SSAs towards the Traitor $Cust_{\psi}$ with the highest likelihood of successfully accessing Data File Segments.

During this simulation, it was necessary to complete two separate IDACS datacenter accesses in order to decrypt a single folder/data file pointer/data file segment; one access to retrieve the Xbits, and one access to retrieve the Xslices. The length of the approach and return authentication chains was 2 ($N = 2$); this parameter was shortened from the previous value of 4 in order to allow more access-DB-attacks to succeed during this simulation. This simulation used the turn-traitor-attack vectors and metamorphic variations defined for Scenario 3 in "4.5.2.2 Simulation Parameters", with 40 attack vectors and 100 variations per attack vector (these parameters were increased in order to mask the limiting effects of these parameters from the results of this simulation). Phase 2 of the simulation began after a threshold of 90% of the SAs and SSAs had been turned into traitors, with a single Traitor $Cust_{\psi}$ for each Traitor SA or SSA. This 90% threshold is much higher than what we would expect to see in a real-world situation; however, it was set at that level for two purposes. First, the 90% threshold represents a catastrophic

scenario; if IDACS is capable of defending against this type of scenario, then the real-world performance is expected to be much higher. Second, it was necessary to raise the threshold to 90% in order for an appreciable number of access-DB-attacks to succeed so that the effect on the Segmented File Directory Tree could be observed. Access-DB-attacks would cease if the percentage of active SAs or SSAs that were traitors fell below 15%. Additionally, no SAs or SSAs in this simulation were spoofed; all traitor SAs or SSAs were completely functional traitors.

During Phase 2 of this simulation, if a Traitor $Cust_{\psi}$ was identified, it would be quarantined and the entire File Directory Tree residing on that $Cust_{\psi}$ would be "re-encrypted". Therefore, if that $Cust_{\psi}$ was later turned into a Traitor, he would have none of the File Directory Tree decrypted, and would have to start again from the "root". However, once a Data File segment was retrieved, it was considered to be owned by the attacker regardless of whether that $Cust_{\psi}$ was detected in the future or not, and that data was added to the pool of data that had been successfully retrieved by the collaborating Traitor $Cust_{\psi}$.

5.3.2 Controlled vs. Runaway simulations

Because the threshold of 90% of SAs and SSAs turned traitor before Phase 2 of the simulation began, a unique situation presented itself. In most cases, the percentage of Traitor SAs and SSAs in IDACS would drop quickly after the start of Phase 2 of the simulation (Fig. 50), similar to the results in "4.5.2.3 Simulation Results" with a 60% threshold. However, with a 90% threshold, in about 1 in 10 simulation runs, new Traitor SAs and SSAs would be turned more quickly than they were detected and quarantined at the beginning of Phase 2. If 100% of the SAs and SSAs in IDACS were turned Traitor, there were no loyal SAs or SSAs remaining to detect, identify, and quarantine the Traitors. In that case, IDACS was completely controlled by the attacker, and the IDACS defenses were completely nullified. In this discussion, a simulation that results in 100% Traitor SAs and SSAs will be termed a "Runaway Botnet", while a simulation that reduces the percentage of Traitor SAs and SSAs over time will be referred to as a "Contained Botnet". Fig. 50 compares the results of IDACS Network loyalty for a given Contained Network simulation run and a given Runaway Network simulation run.

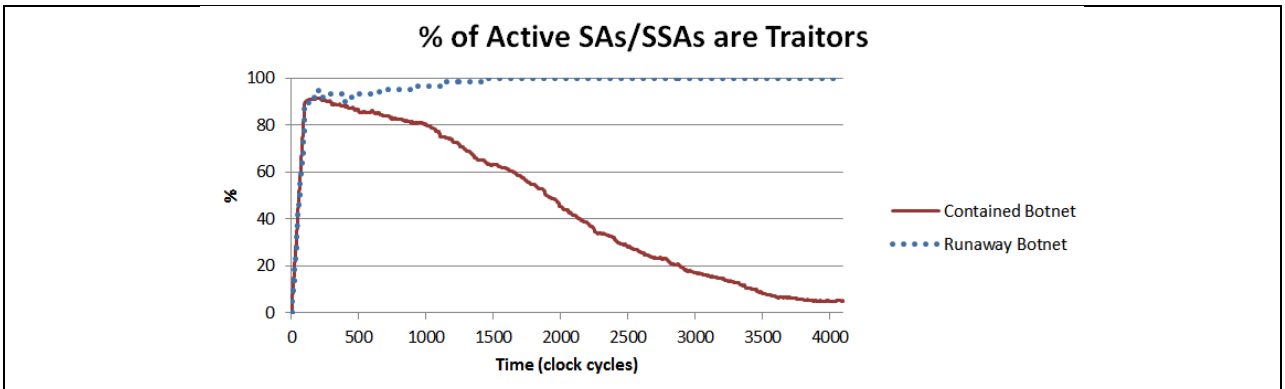


Fig. 50. Percentage of Active SAs/SSAs that are Traitors in IDACS, Contained vs. Runaway Botnet

It should be noted that Runaway Botnet simulations occurred in only 1 out of 10 simulation runs with an SA/SSA Traitor threshold of 90%; Runaway Botnets were not observed in simulations with a threshold of 80% or less. Therefore, a Runaway Botnet represents a highly unlikely, but very catastrophic situation. For the sake of completeness, the results for Runaway Botnet simulations will be included in the following discussions.

5.3.3 Data Protection Results

The results of this simulation were analyzed to discover the effectiveness of the protection provided by the Segmented File Tree Directory against illegal access of the Client-side encrypted data. Fig. 51 shows the percentage of the File Directory Tree stolen by the attacker averaged across 9 Contained Botnet simulation runs, compared to the same data for a single Runaway Botnet simulation. For the Contained Botnet, a small percentage of the File Directory Tree is initially retrieved, riding on the initial high percentage of Traitor SAs and SSAs (Fig. 50). However, as Traitor SAs and SSAs are identified and quarantined, more Traitor $Cust_{\psi}$ are also identified and quarantined; as their individually encrypted File Directory Trees are re-encrypted, those previously retrieved Folders are lost, and the percentage drops. Eventually, when fewer access-DB-attacks are being made and even fewer are successful, due to a low percentage of Traitor SAs and SSAs in IDACS (compare to Fig. 32 and Fig. 34), the percentage of the File Directory Tree that is retrieved will drop to zero. This simulation demonstrates that even under extremely adverse situations, IDACS will contain and eliminate the results of an initial burst of successful illegal IDACS database accesses. However, Fig. 51 also demonstrates that in the rare, catastrophic situation of a Runaway Botnet, the Traitor $Cust_{\psi}$ become functionally equal to Loyal $Cust_{\psi}$, and the entire File Directory Tree can be retrieved with sufficient time.

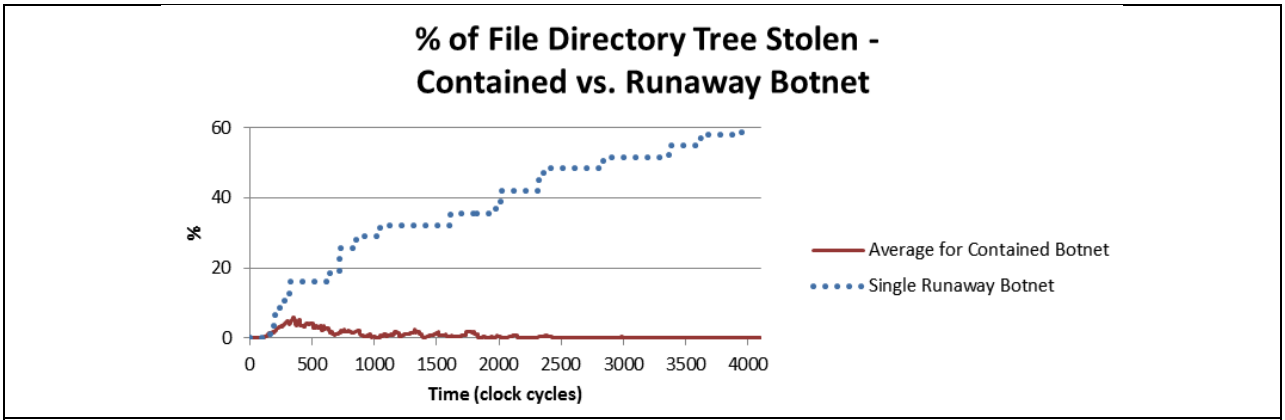


Fig. 51. Percentage of File Directory Tree stolen over simulation period

Fig. 52 and Fig. 53 show the percentage of the total Data File Segments in IDACS that were successfully stolen by Contained Botnets and Runaway Botnets. For a Contained Botnet (which is the more realistic situation), the Traitor $Cust_{\psi}$ meet with some initial success in retrieving Data File Segments (compare to the percentage of the retrieved Data File Tree in Fig. 51). However, after IDACS becomes more successful at blocking access-DB-attacks, no more Data File Fragments are retrieved. Because a Data File Fragment, once stolen, is permanently stolen (re-encryption of a Data File Tree does not nullify the theft of a given Data File Fragment), quarantine of Traitor $Cust_{\psi}$ does not reduce the percentage of Data File Segments stolen; however, IDACS is able to limit the damage to an average of about 0.22%, or slightly less than one complete Data File Segment. Once again, however, if an attacker manages to accomplish a Runaway Botnet (the rare, less realistic situation), there is a virtual open door for the Traitor $Cust_{\psi}$ to retrieve all of the Data File Segments, limited only by time.

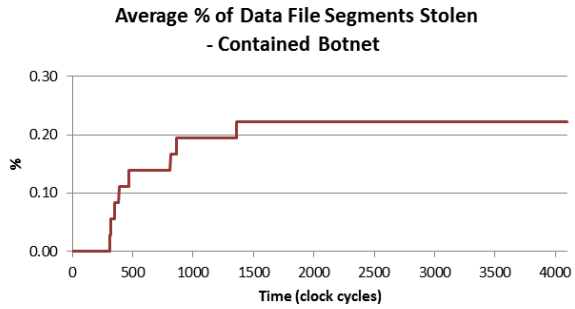


Fig. 52. Average percentage of Data File Segments stolen for a Contained Botnet

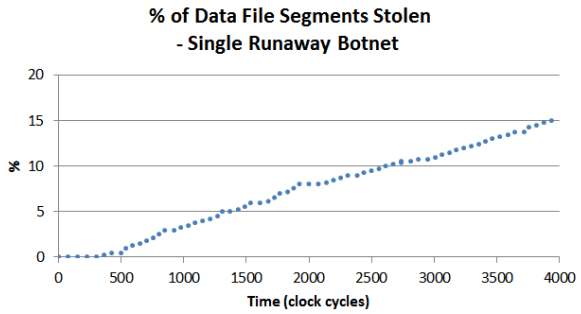


Fig. 53. Percentage of Data File Segments stolen for a Single Runaway Botnet

5.3.4 Leakage Detection Results

One of the advantages of the IDACS Segmented File Directory Tree approach is that it allows previous illegal IDACS datacenter accesses to be detected. When a Traitor $Cust_{\psi}$ is detected, the IDACS forensics engine reports to the System Administrator that all Data File Segments previously retrieved by that $Cust_{\psi}$ have been stolen. It is very useful, in the aftermath of a network breach, to know where data leakages occurred, and what data was leaked.

Fig. 54 and Fig. 55 demonstrate when File Data Segments were retrieved during the simulation, and when they were detected as having been stolen. Fig. 54 shows that File Data Segments stolen during the Contained Botnet simulation were detected as stolen soon after the theft. This demonstrates the strength of data leakage capabilities of IDACS. However, when the Runaway Botnet gains control of all of the SAs and SSAs in IDACS (Fig. 55), none of the stolen data is detected as such.

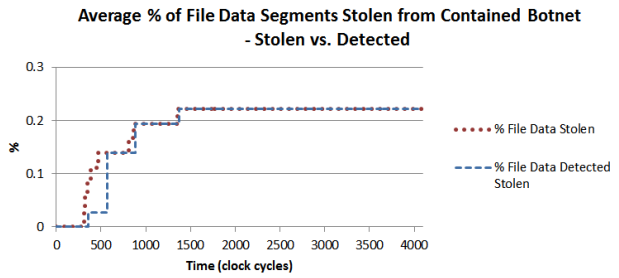


Fig. 54. Histogram for when Data File Segments were stolen vs. when they were detected Stolen; Contained Botnet

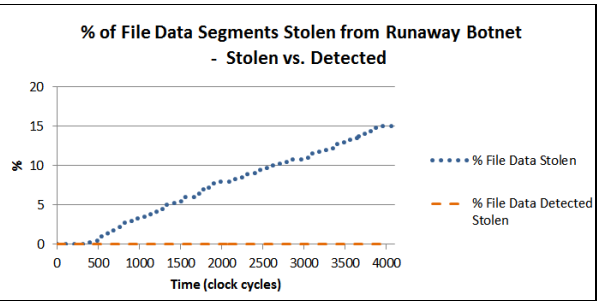


Fig. 55. Histogram for when Data File Segments were stolen vs. when they were detected stolen; Runaway Botnet

When a Traitor $Cust_{\psi}$ is detected to be a Traitor, this allows IDACS to hold that $Cust_{\psi}$ accountable for stealing files. Fig. 56 demonstrates the success rate for identifying every $Cust_{\psi}$ that was ever turned Traitor for the Contained and Runaway Botnet simulations. In the common real-world case (Contained Botnet), IDACS is able to identify and quarantine a large percentage of the Traitor $Cust_{\psi}$, only failing to identify them once the percentage of Traitor SAs and SSAs drops below the threshold below which access-DB-attacks s are no longer launched (compare to Fig. 50). Thus, IDACS provides strong capabilities for identifying and holding accountable Byzantine agents in the system. For the rare case (Runaway Botnet), several Traitor $Cust_{\psi}$ are identified, but once 100% of the SAs and SSAs in IDACS are turned Traitor, there are no new detections of Traitor $Cust_{\psi}$.

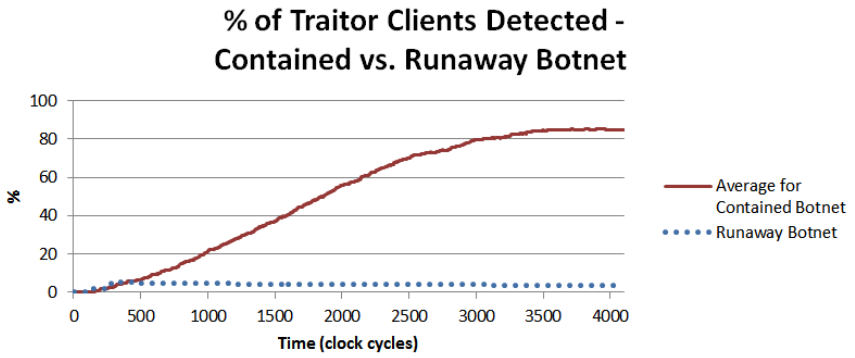


Fig. 56. Percentage of all Clients ever turned Traitor that are detected, Contained vs. Runaway Botnet

6 Implementation

At this time, the IDACS Network has been implemented by the researchers as a proof-of-concept to demonstrate the feasibility of IDACS. The current implementation focuses on the IDACS Network Protocol (which governs network messages as well as the exchange of $\overline{\boldsymbol{OTP}_{\psi}}$, $\overline{\boldsymbol{PID}_{\psi}}$, and XV) and distributed storage. Only limited forensics capabilities have been implemented; currently, only Algorithm 9 (identifying stolen/cloned Client-side $Client_{\rho}$, $Badge_{\zeta}$, Pwd_{θ}, and PIN_{λ}) has been implemented. Algorithm 7 (identify root attacker),

Algorithm 8 (identify root attacker's botnet), and Algorithm 10 (identify controlled/spoofed SAs and SSAs) will be implemented at a later time.

All IDACS Network elements (SAs, SSAs, and Databases) and the User Badge ($Badge_{\zeta}$) have been implemented in Java as Command Line Interface (CLI) programs (Fig. 58 (a) through (d)). The Client Device ($Client_{\rho}$) has been implemented in two different ways. First, it has been implemented as a CLI program (Fig. 58 (e)). This Client Device implementation performs simple Read and Write operations to store and retrieve blocks of data on the IDACS Database; the purpose of this implementation is to test the IDACS reaction to incorrectly formed PIDs (PID_{ε}) and also compromise of the Client Device ($Client_{\rho}$), User Badge ($Badge_{\zeta}$), User Password (Pwd_{θ}), or Badge PIN (PIN_{λ}). The second Client Device implementation is an app that runs on a BlackBerry 9800 simulator available from RIM (Fig. 58 (f)). This app encrypts a file and stores Xslices and Xbits on the DB in a distributed manner. The current implementation has been tested in the configuration shown in Fig. 57; however, the IDACS network can be scaled to any size desired (1 SSA, 2 SAs, and 1 DB minimum are required). The space separation of the Client Device ($Client_{\rho}$) and User Badge ($Badge_{\zeta}$) is simulated by storing their respective cryptographic seeds on different Java Virtual Machines.

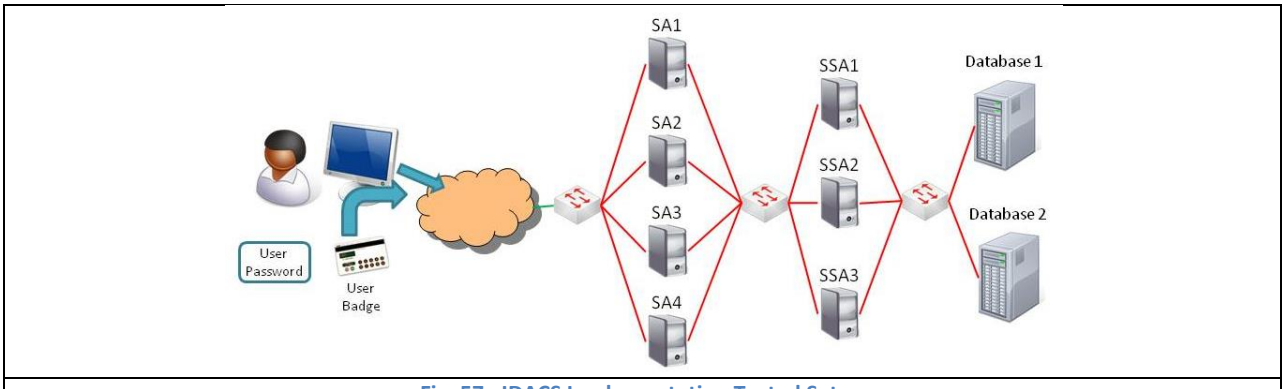


Fig. 57. IDACS Implementation Tested Setup

In addition to being scalable, this implementation of IDACS uses a network communications protocol that achieves reliable delivery over UDP. Given the particulars of the IDACS algorithm and the per-message overhead $\overline{OTP_\psi}$, $\overline{PID_\psi}$, and XV, the researchers believe that combining IDACS with TCP mightbe inefficient. The researchers believe that it would be more efficient to build a proprietary IDACS protocol on top of UDP. Therefore, the current implementation of IDACS is built on top of UDP; in order to compensate for the reliable delivery problem, IDACS contains built-in reliable delivery capabilities based on the concept of TCP SACKs [34]. Using this method, IDACS is able to reliably transfer large numbers of packets.

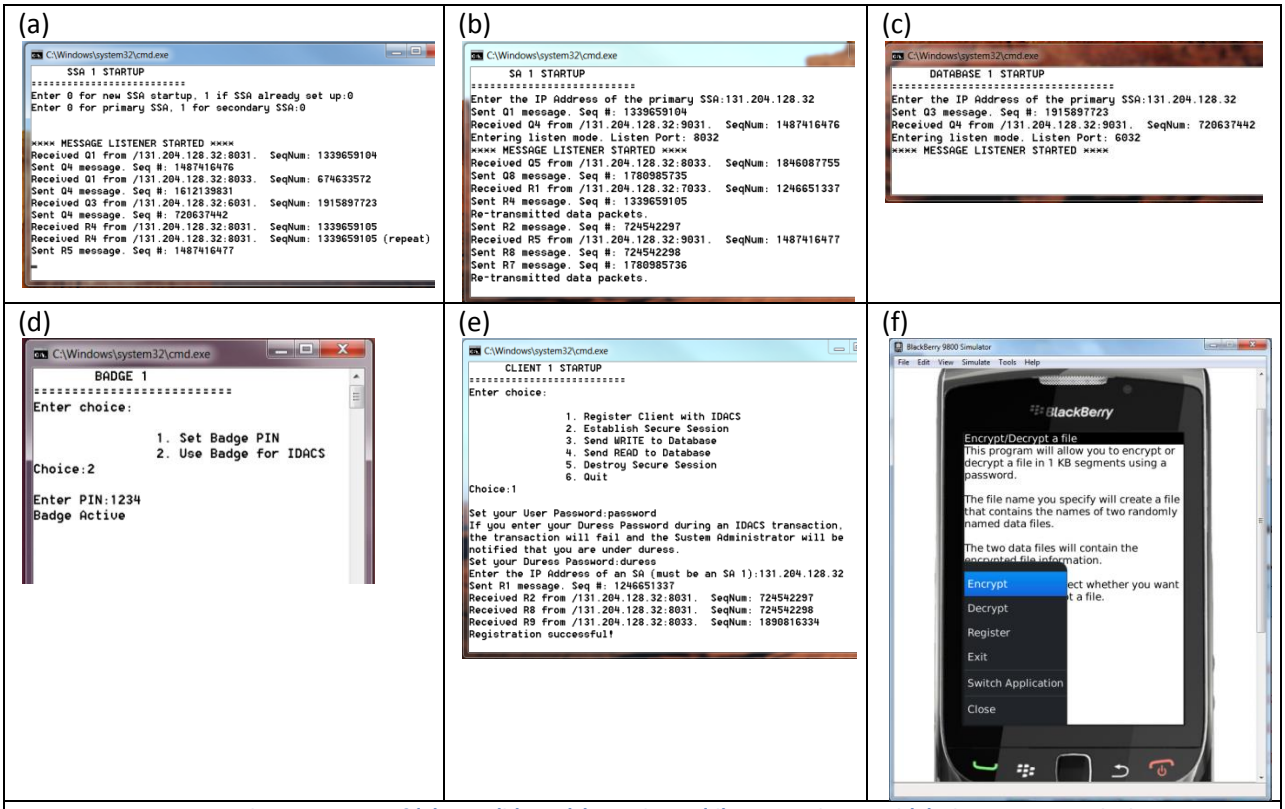
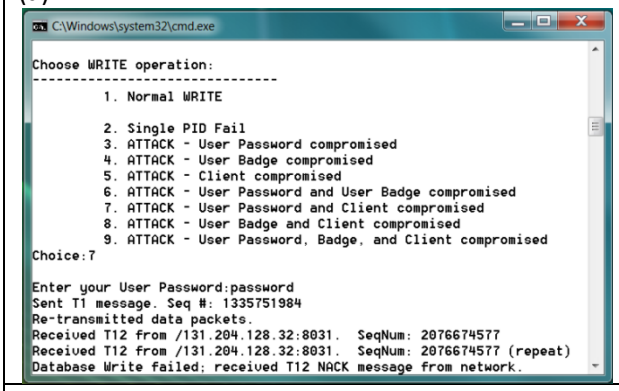


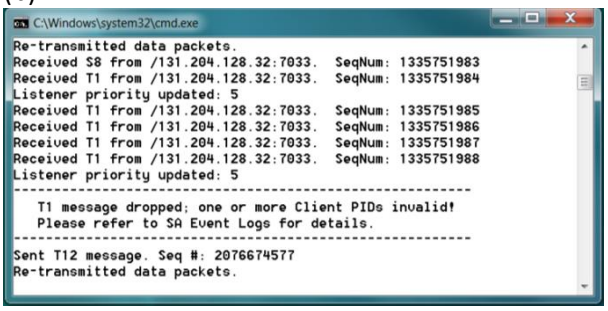
Fig. 58. Java CLI implementations of (a) SSA, (b) SA, (c) Database, (d) User Badge , and (e) Client Device. BlackBerry app implementation of (f) Client.

A demonstration of the real-time digital forensics capabilities of this implementation is shown in Fig. 59. A simulated attacker attempts a Data Write operation, having stolen the User Password and the Client Device, but not the User Badge (the Badge PIN is bundled with the User Badge in this particular implementation) (Fig. 59 (a)). Due to the missing cryptographic seeds residing on the User Badge, several of the PIDs (PID_ε) cannot be formed correctly; these incorrect PIDs are detected by an SA, and the attack is flagged (Fig. 59 (b)). Based on the cryptographic seed space separation and PID formation in this particular IDACS implementation, the digital forensics suite is able to determine correctly that the User Password and Client device were stolen or cloned (Fig. 59 (c)).

(a)



(b)



(c)

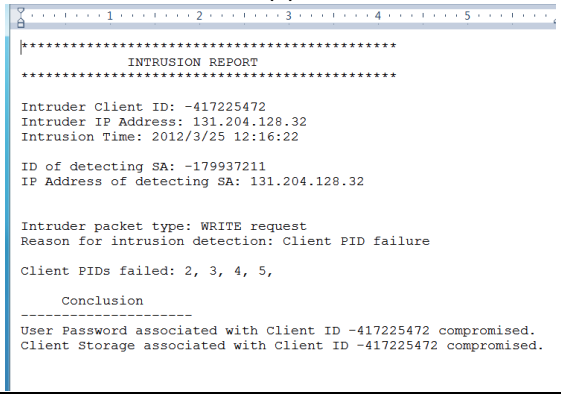


Fig. 59. A demonstration of the implementation: (a) Attacker with access to User Password and Client Device; (b) Attacker attemps a *Write* Transaction and is detected by an SA due to Client PID violations; (c)digital forensics report identifies which User Authentication elements have been compromised.

The BlackBerry application implements the concepts of space/time-separation and also Xbits and Xslices to protect encrypted data. When the BlackBerry application is run, it is given a file to encrypt. The application begins with a few randomly-generated cryptographic seeds that are the basis for all following actions. These cryptographic seeds are used to seed a pseudo-random process which divides the file data into pseudo-random-sized blocks and encrypts each block with a unique pseudo-random AES key. Next, the resulting ciphertext is divided into pseudo-random-sized blocks, and a certain percentage of those blocks are removed as Xslices. Xbits are also pseudo-randomly removed from these cryptographic seeds (using the User Password as a seed for the pseudo-randomness). The post-Xslice ciphertext is then divided into 1 KB blocks, which are stored in alternating data files (Fig. 59 (a)) that have random file names and random file extensions (Fig. 59 (b)). A "pointer" file is formed which contains the names of the data files as well as the cryptographic seeds (minus the Xbits). All of this information is mixed randomly with garbage data, encrypted with the User Password, and stored in the "pointer" file (Fig. 59 (c)). The Xslices and Xbits are sent to IDACS to be stored on a random Database (Fig. 59 (d)). Only the SSAs are able to link the User with the stored Xslices and Xbits; the Databases store no information regarding the type of the data or the owner of this data. The

Database stores all data simply as data; there is no indication as to whether the data is Xslices, Xbits, or another type of data. In this way, an Attacker would be forced to compromise both an SSA and the correct Database to recover the Xbits or Xslices and relate them to the correct Client Device and User. In order to decrypt the file, the User must possess the Client Device and provide the correct User Password to extract and retrieve all the relevant data to complete reassembly and decryption (Fig. 59 (e)). In this implementation, the space separation of the storage of authentication items is simulated by storing Client and User Badge data in different text files (Fig. 59 (f)); however, due to the difficulty of integrating a stand-alone User Badge program with the BlackBerry simulator, the User Badge interface and the User Badge PIN are only used in conjunction with the CLI Client, not the BlackBerry application.

Fig. 60. BlackBerry implementation of IDACS encryption and distributed storage. (a) file ciphertext with Xslices removed is (b) divided and stored in multiple data files. (c) Names of data files are encrypted and stored in a pointer file. (d) Xslices and Xbits are stored on IDACS Database as pure data; no information saved on Database indicating the identity of this data. (e) Correctly reassembling all distributed and mutated pieces results in correct file decryption. (f) Distributed storage between Client and User Badge is simulated using separate text files.

7 Conclusion

The work presented in this paper describes an integrated security system IDACS that utilizes the space-time separated and jointly evolving relationship to provide multiple layers of constantly-changing barriers that will be mathematically infeasible for attackers to predict. The implementation of these ideas in the IDACS system can successfully detect and defeat different types of network attacks, including zero-day attacks. Table 10 details several common network attacks that IDACS addresses. Mathematical analysis demonstrates that it is infeasible to recreate the IDACS authentication protocol, and simulations also reinforce the strength of these space-time relationships.

Table 10. Types of Network Attacks Defeated by IDACS

Attack defeated	Reason
Zero-day attack	Network access control is mathematically defined by a space-time separated and jointly evolving relationship; zero-day attacks which can compromise hosts cannot forge such a relationship when accessing protected data; furthermore, the zero-day attack method can be captured and analyzed
Denial-of Service (DoS) attack	Quick stateless OTP checking allows attack packets to be quickly discarded
Replay attack	Time-evolution means OTPs and PIDs are true one-use items tied to packet sequence number
Client-side device loss	Cryptographic seeds for calculating authentication parameters are space-separated; loss of one or more devices does not allow attacker to reconstruct the space-time separated and evolving relationship
SA and/or SSA hijacking	Mutual support in authentication chain detects a hijacked SA or SSA
SA and/or SSA memory leakage	Space-time separation and evolution of cryptographic seeds means memory leakage at one or more SAs does not leak all OTP/PID seeds
System downtime while waiting for network healing	Space-time evolving determination of network-side authentication chain path allows real-time network healing with no network downtime by using available system migration

In addition to detecting and preventing attacks, the design of IDACS also provides real-time forensics capabilities, allowing traitorous network actors to be identified quickly and accurately. Simulations demonstrate that IDACS forensics are efficient and effective. Also, IDACS uses the space-time separated and jointly-evolving relationship to protect at-rest encrypted data. Time-changing Xbits and Xslices stored across multiple locations and data segmentation provide greater security for encrypted data. Once again, mathematical analysis demonstrates the theoretical strength of this system, and simulation provides a more concrete expression of this security.

IDACS implements the space-time separated and jointly-evolving relationship across multiple aspects of the system to provide a complete end-to-end network and data protection system that has strong mathematical properties.

8 References

[1] J. Wylie, M. Bigrigg, J. Strunk, G. Ganger, H. Kiliccote and P. Khosla, "Survivable information storage systems," *Computer,* vol. 23, no. 8, pp. 61-68, August 2000.

[2] H. Sengar, X. Wang, H. Wang, D. Wijesekera and S. Jajodia, "Online detection of network traffic anomalies using behavioral distance," *Quality of Service, 2009. IWQoS. 17th International Workshop on,* pp. 1-9, July 2009.

[3] A. Mishra, K. Nadkarni and A. Patcha, "Intrusion detection in wireless ad hoc networks," *IEEE Wireless Communications,* vol. 11, no. 1, pp. 48-60, February 2004.

[4] A. Bouchahda, N. L. Thanh, A. Bouhoula and F. Labbene, "Enforcing Access Control to Web Databases," in *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, 2010.

[5] M. Kiani, A. Clark and G. Mohay, "Evaluation of Anomaly Based Character Distribution Models in the Detection of SQL Injection Attacks," in *Third International Conference on Availability, Reliability and Security, 2008. ARES 08*, 2008.

[6] Y. Amir, Y. Kim, C. Nita-Rotaru and G. Tsudik, "On the performance of group key agreement protocols," *ACM Trans. Inf. Syst. Secur.,* vol. 7, no. 3, pp. 457-488, August 2004.

[7] U. Tupakula and V. Varadharajan, "On the design of Virtual machine Intrusion detection system," in *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on* , 2011.

[8] X. Zhao, K. Borders and A. Prakash, "Towards protecting sensitive files in a compromised system," in *Security in Storage Workshop, 2005. SISW '05. Third IEEE International*, 2005.

[9] R. Sandhu and V. Bhamidipati, "The ASCAA Principles for Next-Generation Role-Based Access Control," in *ARES 2008 - International Conference on Availabilityy, Reliability and Security*, 2008.

[10] T. Lodderstedt, D. Basin and J. Doser, SecureUML: A UML-Based Modeling Language for Model-Driven Security, Springer Berlin /Heidelberg, 2002.

[11] C. Cachin and Chandran, N., "A Secure Cryptographic Token Interface," in *Computer Security Foundations Symposium, 2009. CSF '09. 22nd IEEE*, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.161.1048, 2009.

[12] J. B. a. F. S. J. Anderson, "Inglorious Installers: Security in the Application Marketplace," in *Proc. 9th Workshop Economics of Information Security*, 2010.

[13] D. Z. K. S. Haining Wang, "Change-point monitoring for the detection of DoS attacks," *IEEE Transactions of Dependable and Secure Computing,* vol. 1, no. 4, pp. 193-208, Oct-Dec 2004.

[14] K. L. W. Z. Yang Xiang, "Low-Rate DDoS Attacks Detection and Traceback by Using New Information Metrics," *IEEE Transactions on Information Forensics and Security,* vol. 6, no. 2, pp. 426-437, June 2011.

[15] Y. Kim, W. C. Lau, M. C. Chuah and H. Chao, "PacketScore: a statistics-based packet filtering scheme against distributed denial-of-service attacks," *IEEE Transactions on Dependable and Secure Computing,* vol. 3, no. 2, pp. 141-155, April-June 2006.

[16] X. Wang and M. Reiter, "Using Web-Referral Architectures to Mitigate Denial-of-Service Threats," *IEEE Transactions on Dependable and Secure Computing,* vol. 7, no. 2, pp. 203-216, April-June 2010.

[17] P. Peng, P. Ning and D. S. Reeves, "On the secrecy of timing-based active watermarking trace-back techniques," in *Security and Privacy, 2006 IEEE Symposium on*, 2006.

[18] N. Paxton, G.-J. Ahn and B. Chu, "Towards Practical Framework for Collecting and Analyzing Network-Centric Attacks," in *IEEE International Conference on Information Reuse and Integration, 2007. IRI 2007.*, 2007.

[19] D. Whyte, P. C. v. Oorschot and E. Kranakis, "Tracking Darkports for Network Defense," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 2007.

[20] A. G. Y. C. V. P. Zhichun Li, "Towards Situational Awareness of Large-Scale Botnet Probing Events," *IEEE Transactions on Information Forensics and Security,* vol. 6, no. 1, pp. 175-188, March 2011.

[21] M. C. Y. C. M. Q. Kai Hwang, "Hybrid Intrusion Detection with Weighted Signature Generation over Anomalous Internet Episodes," *IEEE Transactions on Dependable and Secure Computing,* vol. 4, no. 1, pp. 41-55, Jan-March 2007.

[22] K. Brasee, S. Makki and S. Zeadally, "A Novel Distributed Authentication Framework for Single Sign-On Services," in *2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, Taichung, Taiwan, 2008.

[23] T. Li and R. Deng, "Vulnerability Analysis of EMAP-An Efficient RFID Mutual Authentication Protocol," in *Availability, Reliability and Security, 2007. ARES 2007. The Second International Conference on*, 2007.

[24] S. Peisert, M. Bishop, S. Karin and K. Marzullo, "Toward Models for Forensic Analysis," in *Systematic Approaches to Digital Forensic Engineering, 2007. SADFE 2007. Second International Workshop on*, 2007.

[25] S. Kirbiz, M. Celik, A. Lemma and S. Katzenbeisser, "Watermarking of AAC Bit-streams for Forensic Tracking," in *Signal Processing and Communications Applications, 2007. SIU 2007. IEEE 15th*, 2007.

[26] A. Sharma, Z. Kalbarczyk, R. Iyer and J. Barlow, "Analysis of Credential Stealing Attacks in an Open Networked Environment," in *Network and System Security (NSS), 2010 4th International Conference on* , 2010.

[27] F. Valeur, G. Vigna, C. Kruegel and R. Kemmerer, "A Comprehensive approach to intrusion detection alert correlation," *IEEE Transactions on Dependable and Secure Computing,* vol. 1, no. 3, pp. 146-169, July-September 2004.

[28] A. Hofmann and B. Sick, "Online Intrusion Alert Aggregation with Generative Data Stream Modeling," *Dependable and Secure Computing, IEEE Transactions on,* vol. 8, no. 2, pp. 282-294, March-April 2011.

[29] F. e. a. Chang, "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems (TOCS),* vol. 26, no. 2, June 2008.

[30] H. H. Huang and A. S. Grimshaw, "Automated performance control in a virtual distributed storage system," in *GRID '08 Proceedings of the 2008 9th IEEE/ACM International Conference on Grid Computing*, Washington, DC, 2008.

[31] H. Weatherspoon, P. Eaton, C. Byung-Gon and J. Kubiatowicz, "Antiquity: exploiting a secure log for wide-area distributed storage," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 200*, New York, NY, 2007.

[32] K. S. a. N. Memon, "Automatic Reassembly of Document Fragments via Context Based Statistical Models," in *Proceedings of the 19th annual Computer Security Applications Conference (ASAC '03)*, Washington, D.C., 2003.

[33] M. R. Garey and D. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, San Fracisco, CA: Freeman, 1979.

[34] July 2011. [Online]. Available: http://csrc.nist.gov/groups/ST/toolkit/mg/documentation_software.html.

[35] A. R. et.al., "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications," Gaithersburg, MD, 2010.

[36] A. Maurer and S. Tixeuil, "Limiting Byzantine Influence in Multihop Asynchronous Networks," *2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS),* pp. 183-192, 2012.

[37] V. Pandit, J. H. Jun and D. Agrawal, "Inherent Security Benefits of Analog Network Coding for the Detection of Byzantine Attacks in Multi-Hop Wireless Networks," *2011 IEEE 8th International Conference on Mobile Adhoc and Sensor Systems (MASS),* pp. 697-702, 2011.

[38] F. Tao, Z. Bingtao and M. Jianfeng, "Security Random Network Coding Model against Byzantine Attack Based on CBC," *2011 International Conference on Intelligent Computation Technology and Automation (ICICTA),* vol. 2, pp. 1178-1181, 2011.

[39] K. Shanmugasundaram and N. Memon, "Automatic Reassembly of Document Fragments via Context Based Statistical Models," in *Proceedings of the 19th annual Computer Security Applications Conference (ASAC '03)*, Washington, D.C., 2003.

[40] M. Mathis, J. Mahdavi, S. Floyd and A. Romanow, October 1996. [Online]. Available: http://tools.ietf.org/html/rfc2018.

[41] C.-H. '. Wu, T. Liu, C.-C. '. Huang and J. D. Irwin, "Modelling and Simulations for Identity-Based Privacy-Protected Access Control Filter (IPACF) Capability to Resist Massive Denial of Service Attacks," *International Journal of Information and Computer Security,* vol. 3, no. 2, pp. 195-223, 2009.

9 Appendix A

The purpose of this section is to prove that the Maximum Weight Path of Specified Length (MWPSL) and Maximum Weight Directed Path of Specified Length (MWDPSL) decision problems are NP-complete.

The process of proving a given decision problem **C** to be NP-complete has two steps:

1. Show that **C** is in NP
2. Show that every problem in NP is reducible to **C** in polynomial time

The first step can be shown by demonstrating that a candidate solution to **C** can be checked for correctness in polynomial (or better) time. The second step can be shown by demonstrating that any *one* known NP-complete problem **B** is reducible to **C**. If one NP-complete problem **B** can be reduced to **C**, then all other NP-complete problems can be reduced to **C**. A problem **B** is reducible to problem **C** if there is a polynomial-time, many-one reduction from **B** to **C**; that is, there is a reduction that can transform any instance of **B** into an instance of **C**. Any algorithm that can be used to solve all instances of problem **C** can be used to solve all instances of problem **B** [30].

The process of proving that the MWPSL and MWDPSL problems (**C**) are NP-complete begins with a proven NP-complete problem, the Hamiltonian Path problem (**B**) [30]. The reduction path between the Hamiltonian Path problem (**B**) and the MWPSL and MWDPSL (**C**) problems is shown in Fig. 61.

Fig. 61. NP-complete reduction path

In the following sections, each of the reductions will be shown in the indicated steps. At the end, the MWPSL and MWDPSL (**C**) problems will be proven to be NP-complete.

Starting Point: The Hamiltonian Path is NP-Complete

Hamiltonian Path: Given an undirected graph G = (**V**, **E**) where **V** is a set of vertices $\{v_1, v_2, \ldots\}$ and every $e \in \mathbf{E}$ is an unordered set of vertices $\{v_1, v_2\}$ called edges. Does G contain a Hamiltonian path, which is a sequence $\langle v_1, v_2, \ldots, v_\eta \rangle$ of distinct vertices from **V** such that $\{v_i, v_{i+1}\} \in \mathbf{E}$ for $1 \leq i < \eta$ and every member of **V** appears once and only once in the sequence?

The Hamiltonian Path problem has been proven NP-complete [30].

Step 1: Show that the Maximum Weight Hamiltonian Path problem is NP-complete.

Maximum Weight Hamiltonian Path: Given an undirected graph G = (**V**, **E**) where every $e \in E$ is an unordered set of vertices $\{v_1, v_2\}$ called edges and has a weight **W**(e) $\in \mathbf{Q}^+$, and there is a number R $\in \mathbf{Q}^+$. Is there a Hamiltonian path $\langle v_1, v_2, \ldots, v_i, \ldots, v_\eta \rangle$ in

G where η=|**V**| such that $\sum_{i=1}^{\eta-1} \mathbf{W}(v_i, v_{i+1}) \geq R$, where {$v_i$, v_{i+1}} ϵ **E**?

Step 1.1: Show that the Maximum Weight Hamiltonian Path is in NP.

A candidate solution to this problem can be checked by tracing the path, verifying that each vertex is touched once and only once, and summing the weights of the edges in the path and checking the final sum. The candidate solution is checked in linear time.

Step 1.2: Show that the Hamiltonian Path problem is reducible to the Maximum Weight Hamiltonian Path problem in polynomial time.

The Hamiltonian Path problem is a special case of the Maximum Weight Hamiltonian Path problem, so the first can be reduced to the second. Create an instance of the Maximum Weight Hamiltonian Path problem. Set all **W***(e)* = 1 for all *e* ϵ **E** and set R = (|**V**| - 1). This is now an instance of the Hamiltonian Path problem, and the reduction is accomplished in linear time.

Result: The Maximum Weight Hamiltonian Path problem is NP-complete.

Step 2: Show that the Maximum Weight Path of Specified Length (MWPSL) problem is NP-complete.

Maximum Weight Path of Specified Length (MWPSL) : Given an undirected graph G = (**V**, **E**) where every e ϵ **E** is an unordered set of vertices {v_1, v_2} called edges and has a weight **W**(e) ϵ $\mathbf{Q}^+$, there is a number R ϵ $\mathbf{Q}^+$ and an integer N ≤ |**V**|. Is there a path **P** = <v_1, v_2, …, v_i, …, v_N> in G such that any v ϵ **V** appears *at most* once in **P** and $\sum_{i=1}^{N-1} \mathbf{W}(v_i, v_{i+1}) \geq R$, where {$v_i$, v_{i+1}} ϵ **E**?

Step 2.1: Show that the MWPSL problem is in NP.

A candidate solution that connects some or all of the vertices can be checked by tracing the path, verifying each vertex in the path is touched *at most once*, verifying that there are N vertices in the path, and summing the path edge weights and comparing the sum to R. This candidate solution is checked in linear time.

Step 2.2: Show that the Maximum Weight Hamiltonian Path problem is reducible to the MWPSL problem.

The Maximum Weight Hamiltonian Path problem is a special case of the MWPSL problem, so the first can be reduced to the second. Create an instance of the MWPSL problem and set N = |**V**|. This is now an instance of the Maximum Weight Hamiltonian Path problem; this reduction is accomplished in linear time.

Result: The Maximum Weight Path of Specified Length (MWPSL) problem is NP-complete.

Step 3: Show that the Maximum Weight Directed Path of Specified Length (MWDPSL) problem is NP-complete.

Maximum Weight Directed Path of Specified Length (MWDPSL): Given a directed graph G = (**V**, **E**) where every e ϵ **E** is an *ordered* set of vertices {v_1, v_2} called *arcs* and has a weight **W**(e) ϵ $\mathbf{Q}^+$, there is a number R ϵ $\mathbf{Q}^+$ and an integer N ≤ |**V**|. Is

there a path **P** = <v_1, v_2, …, v_N> in G such that any v ϵ **V** appears *at most* once in **P** and $\sum_{i=1}^{N-1} \mathbf{W}(v_i, v_{i+1}) \geq R$, where {$v_i$, v_{i+1}} ϵ **E**?

Step 3.1: Show that the MWDPSL problem is in NP.

A candidate solution that connects some or all of the vertices can be checked by tracing the path, verifying each vertex in the path is touched at most once, verifying that there are N vertices in the path, and summing the path edge weights and comparing the sum to R. This candidate solution is checked in linear time.

Step 3.2: Show that the MWPSL problem is reducible to the MWDPSL problem.

The MWPSL problem is a special case of the MWDPSL problem, so the first can be reduced to the second. Create an instance of the MWDPSL problem corresponding to an instance of the MWPSL problem where every e ϵ **E** with a given **W**(e) in the MWPSL problem is replaced by a pair of opposite-direction directed e ϵ **E** in the MWDPSL problem, both with the same **W**(e) as in the MWPSL problem. This now equates to an instance of the MWPSL problem; this reduction is accomplished in linear time.

Result: The Maximum Weight Directed Path of Specified Length (MWDPSL) problem is NP-complete.

Implications for IDACS strength: The MWPSL and MWDPSL problems represent a reassembly-due-to-space-separation problem at a given instant in time. Thus, the space separation of IDACS provides the NP-completeness to the systems. However, due to the joint time-evolution of IDACS, the problem evolves into a completely new MWPSL or MWDPSL problem each time the system states change (which can occur every few seconds). Therefore, the time-evolution greatly increases the complexity of the problem.