

A.A. 2021/22

versione 0.0

SOFTWARE SECURITY

PROF. MILA DALLA PREDA

FABS :)

NOTA

Questi appunti/sbobinatura/versione “discorsiva” delle slides sono per mia utilità personale,
quindi pur avendole revisionate potrebbero essere ancora presenti typos, commenti/aggiunte personali (che anzi, lascio
di proposito) e nel caso peggiore qualche inesattezza!

Comunque spero siano utili!  

CHANGELOG

- 0.0: in progress

INDICE

NOTA	2
Changelog	2
Indice.....	3
1 – Introduzione	4
2 – User authentication.....	10
3 – Security policies	14
3B - Policy di controllo degli accessi	16
4 - Software protection	21
5 - Offuscamento.....	26
5a - Layout obfuscation	29
5b - Data obfuscation.....	30
5c - Code/control obfuscation.....	35
6 - L'impossibilità dell'offuscamento	44
7 - Offuscamento avanzato: semantics-based offuscation	48
8 - Tamper proofing	57
9 – Watermarking.....	64

1 – INTRODUZIONE

L1 – 05/10/2021

Quando si parla di sicurezza, si parla di assets che vogliamo difendere da danni intenzionali: ci immaginiamo un attore attivo che intenzionalmente vuole sfruttare delle vulnerabilità per riuscire a violare gli assets per ottenere vantaggi, solitamente economici.

Approcci di protezione	Componenti che concorrono agli attacchi
<ul style="list-style-type: none">PrevenzioneDetectionRisposta all'attacco	<ul style="list-style-type: none">VulnerabilitàTecniche e minacceMotivazione dell'attaccante.



Attenzione: Non possiamo risolvere il problema solo attraverso la tecnologia. La sicurezza è un problema delle persone.
(Dieter Golimann)

Security vs. Software security

Ci sono delle differenze fondamentali fra la sicurezza tradizionale del software:

- Il software può essere rubato MA essere ancora in possesso del proprietario
- L'informazione può essere rubata MA essere ancora in possesso del proprietario, e potenzialmente non ce se ne accorge
- Gli attaccanti possono essere ovunque nel mondo.

Definizione: Security

La protezione dei sistemi dal furto o danneggiamento del loro software, hardware o di informazione, anche solo inteso come disturbo o redirezionamento dei servizi che forniscono. (M. Gasser, 1988).

Carrellata storica del problema della Software security

Anni 40: nascono gli elaboratori.

Il problema della sicurezza nasce poco dopo come **confidenzialità**: i sistemi erano solitamente multi utente, e ogni utente doveva poter accedere solo all'informazione a lui dedicata. Il problema era la classifying dell'utente e la protezione dei dati. Questo viene scritto nero su bianco per la prima volta su un report del 1972, in cui si iniziano a riconoscere i limiti di sicurezza dei sistemi esistenti all'epoca.

Anni '70: avvento dei mainframe, aka computer in grado di processare mole di dati

Segmentazione delle macchine: oltre alle questioni sulla segretezza dell'informazione, si inizia anche a pensare di poter garantire delle zone fisicamente separate nelle macchine.

Privacy: si inizia a pensare che l'informazione, benché non classificata, possa lo stesso essere sensibile. Quindi si iniziano a sviluppare gli algoritmi di crittografia, e si iniziano a studiare i problemi legati al fatto che i dati non devono essere deducibili da query successive (quando non è accessibile direttamente)

Anni '80: PC

I sistemi erano personal, quindi erano sicuri in quanto locali per persona e isolati. I problemi arrivavano solo col **cambio di contesto**. Continuano a svilupparsi modelli per la policy fra utenti, con tutti i modelli di non interferenza e information flow applicati ai sistemi multeutente. In questi anni è pubblicato anche l'orange book, uno dei primi libri con i criteri per valutare la confidenzialità. Iniziano anche i primi software scritti con l'intento esplicito di inserirsi nelle macchine senza contesto.

Anni '90: Internet

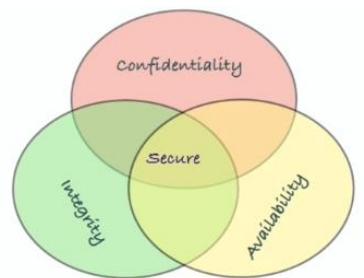
L'isolamento non esiste più; nascono tre problematiche

- In un primo momento il problema della sicurezza era più stato interpretato come **sicurezza delle comunicazioni**.
- Poi si è capito che il problema era molto più ampio: se la macchina è connessa non abbiamo più **controllo sugli input** ricevuti e su chi li manda.
- In questi anni esplode anche tutta la problematica della **protezione della proprietà intellettuale**, e i pc diventano anche strumenti di intrattenimento che devono essere sottoposti a DRM.

Definizione di sicurezza

La sicurezza è l'intersezione di tre proprietà:

Confidenzialità	Integrità	Availability
<p>Solo gli utenti autorizzati possono accedere alle risorse.</p> <p>Si declina in</p> <ul style="list-style-type: none">• Segretezza: informazioni riservate che non devono essere rese pubbliche• Privacy: informazioni sensibili che non devono essere rilasciate	<p>Solo gli utenti autorizzati possono modificare le risorse.</p> <p>Vogliamo prevenire le modifiche non autorizzate.</p>	<p>Un servizio deve essere raggiungibile, utilizzabile e poter funzionare sempre in un tempo ragionevole.</p> <p>Il classico attacco di questo tipo è il Denial of Service, in cui l'attaccante bombardia la vittima di messaggi inutile in modo da non renderlo più disponibile.</p>



Vulnerabilità, minacce/tecniche e motivazioni sono gli ingredienti che l'attaccante sfrutta per attaccare un asset.

Gli attacchi possibili sono tantissimi: per stabilire se un sistema è sicuro quindi è necessario decidere quali siano gli assets e gli attacchi da proteggere.

La sicurezza riguarda la protezione degli assets, e le tecniche di sicurezza suggerite solitamente impongono dei limiti o dei protocolli che gli utenti devono seguire per mantenere la sicurezza del sistema. Il problema di informare gli utenti e renderli consapevoli è molto importante per evitare che il sistema di sicurezza venga bypassato.

La security policy è la formalizzazione degli obiettivi di sicurezza di un'organizzazione, e di come quest'organizzazione decide di raggiungerli.

Attacchi

Attacco

Minaccia che si realizza tipicamente attraverso lo sfruttamento di una vulnerabilità.

È utile rappresentare gli attacchi come un albero dove alla radice ho la minaccia realizzata, mentre i figli sono i sottogool necessari a realizzare quella minaccia.

(i nodi sono tutti in or tranne quello con la doppia ondina).

Scrivere l'albero dell'attacco implica una conoscenza molto profonda dell'attacco, e solitamente è un task non banale ma che può portare a una comprensione profonda dell'attacco.



Oltre all'impatto, viene tipicamente misurata anche la "probabilità"/facilità che l'attacco si verifichi. Anche in questo caso, Microsoft propone una sua classificazione che prende il nome di DREAD:

- **D – Damage potential:** potenziale danno, relativo al valore in denaro del danno che potrebbe essere causato con quell'attacco
- **R – Riproducibilità:** quanto è semplice replicare l'attacco più di una volta
- **E – Exploitability:** impegno, conoscenza e risorse necessarie a completare l'attacco
- **A – Affected users:** numero di utenti che subiscono l'effetto dell'attacco
- **D – Discoverability:** capacità di identificare l'attacco avvenuto.

Proprietà della sicurezza, dal punto di vista degli attacchi

	Integrità		Availability	Confidenzialità
	dati	sistemi		
<i>Unauthorized disclosure</i>				x
<ul style="list-style-type: none"> Esposizione: può essere volontaria –un insider rilascia le informazioni, senza violare alcun sistema – o per errore Intercettazione: attacco “man-in-the-middle” fra due utenti autorizzati Inference: l’informazione viene derivata indirettamente da altri dati Inrusione: un’entità non autorizzata ottiene l’accesso ai dati sensibili aggirando le protezioni. 				
<i>Deception</i> Il sistema viene ingannato presentando dei dati che non sono autentici.	x	x		
<i>Disruption</i>		x	x	
<i>Usurpation</i> Mirano a ottenere controlli non leciti; violano per esempio i privilegi dei vari utenti		x		

Proprietà della sicurezza, dal punto di vista degli assets

	Integrità	Availability	Confidenzialità
<i>Attacchi all’hardware</i>	L’attrezzatura viene distrutta.	L’attrezzatura viene rubata	Furto di drives non criptati.
<i>Attacchi al software</i>	Modifiche non autorizzate, come ad esempio salto del controllo di licenza oppure jailbreak/modifiche custom del software; quindi c’è sia la proprietà intellettuale, che la safety. → funny: azienda ha contattato l’uni perché i clienti hanno voluto il codice sorgente “per sicurezza”, e turns out che lo hanno usato per aggiungersi funzioni custom. Ma che succede se poi il macchinario malfunziona? Come dimostrò che avevano cambiato il codice?	Cancellazione del SW impedendo che gli utenti vi accedano	Utilizzo e distribuzione non legale di codice proprietario.
<i>Attacchi ai dati</i>	Modifica non autorizzata di files esistenti, o creazione di files falsi.	Files cancellati, cambiati o resi inaccessibili	Lettura non autorizzata o estrapolazione dei dati attraverso analisi statistiche
<i>Attacchi sulla comunicazione/rete</i>	I messaggi sono modificati, cambiati di ordine o duplicati; possono anche essere fabbricati dei messaggi falsi.	distruzione dei messaggi, le linee di comunicazioni sono spente.	i messaggi vengono letti, e il pattern di scambio può essere osservato.

Gli attacchi alla comunicazione si distinguono in:

- Passivi:** spiano, violando la confidenzialità, ma non alterano lo scambio di messaggi
- Attivi:** agiscono direttamente sui messaggi che vengono scambiati; sono più semplici da identificare ma è difficile prevenirli

Principio dell’easiest breach

Bisogna considerare che l’attaccante potrebbe usare qualunque tecnica, non necessariamente quella più ovvia o quella che noi abbiamo immaginato. Di conseguenza, bloccare una strada non significa aver reso sicuro l’asset.

La sicurezza informatica è un gioco in cui solo il team in difesa rispetta le regole.

Analisi dei rischi

Misura il grado di rischio di ogni vulnerabilità, ovvero quanto è importante il danno che quella vulnerabilità potrebbe portare a realizzare. Tutta l'analisi dei rischi e della vulnerabilità è importante perché dà una priorità sulle vulnerabilità da risolvere. L'analisi dei rischi è un processo non semplice e costoso se fatto ad hoc sull'azienda, quindi spesso ci si riferisce a degli standard di sicurezza per cercare di garantirsi. Alla fine di tutta questa analisi, il risultato è una **lista con priorità delle minacce**.

Il rischio combina questi tre aspetti:

$$\text{Risk} = \text{Assets}(1) \times \text{Threats}(2) \times \text{Vulnerabilities}(3)$$

Possiamo avere una valutazione:

Quantitativa

Non sempre utile ma molto complessa

Qualitativa

È più tipica, e mette in relazione i rischi a diversi livelli di granularità.

Adam Shostack è un esperto di Cyber Security che ha scritto Threat Modeling, e fa delle talk carine tipo "Threat modeling lessons from Stars Wars".

(1) Assets

Gli assets di un sistema informatico sono:

- **Hardware:** computer, routers, cellulari, carte...
- **Software:** applicazioni, sistemi operativi, DBMS, codice sorgente
- **Dati e informazioni:** dati essenziali per runnare e pianificare il buisness, documenti di design, contenuti digitali e dati sugli utenti
- **Reputazione**

Principio di temporalità

Gli elementi di un sistema informativo devono essere protetti solo finché hanno valore, e relativamente al loro valore.

Quando si proteggono gli assets è bene avere in mente una finestra temporale in cui si vogliono proteggere gli assets. Per esempio, se voglio proteggere la trasmissione di una partita di calcio, mi interesserà proteggere la chiave per quei 90 minuti.

(2) Threats - Classificazione delle minacce

Ci sono diversi modi di classificare le minacce di un sistema. Una possibile classificazione è STRIDE, di Microsoft:

- **S – Spoofing delle identità:** attacchi in cui l'attaccante impersonifica altri
- **T – Tampering:** manomissione dei dati, aka modifiche non autorizzate ai dati compresi i privilegi degli utenti
- **R – Repudiation:** minacce che permettono a un utente di negare di aver compiuto una certa azione; questo è di interesse nelle comunicazioni
- **I – Information disclosure:** diffusione di informazione sensibile, confidenziale o classificata
- **D – Denial of service:** attacchi alla disponibilità di un servizio
- **E – Elevation of privileges:** escalation di privilegi.

A questi si aggiungono

- **Tampering del codice:** violazione e integrità del codice che può portare a comportamenti non desiderati
- **Software piracy:** distribuzione illegale di codice proprietario.

(3) Vulnerabilities

Identificate le minacce, è interessante vedere che c'è una connessione fra vulnerabilità e minaccia:

- **Minaccia :** qualcosa di cattivo che può succedere
- **Vulnerabilità :** crepa che permette all'attaccante di realizzare la minaccia in un attacco.

Gli attacchi possono essere visti come "azioni non autorizzate", quindi molte delle vulnerabilità hanno a che fare con il controllo degli accessi.

Vulnerability scanners

Esistono delle associazioni che tengono un database delle vulnerabilità, che può essere usato per verificare quali vulnerabilità sono presenti nel sistema. Alcuni esempi sono SANS, Certs...

Requisiti della sicurezza

È chiaro il legame con il controllo degli accessi. Ci sono altre proprietà che devono essere assicurate:

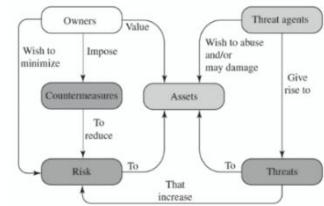
- **Accountability:** il sistema deve tener traccia di cosa accade per poter risalire alla causa del problema in caso di attacchi. È un processo di audit.
- **Responsabilità:** l'utente responsabile deve poter essere rintracciato
- **Autenticità:** devo potermi fidare dei dati e della sorgente dei dati
- **Non repudiation:** non devo poter negare di aver compiuto un'azione; è solitamente legato alle comunicazioni e si risolve con le firme digitali

Sfide della sicurezza

Per quanto possiamo modellare come l'attaccante agirà per portare a termine l'attacco o la minaccia, qualcosa scappa sempre e bisogna continuamente monitorare il sistema per identificare i comportamenti non leciti.

“Questo disegno mi piaceva perché mette assieme tutte le parole che abbiamo visto”

(È un riassunto dell'interazione fra i concetti)



Design principles

Non c'è una ricetta pronta per garantire la sicurezza del sistema; tuttavia esistono dei principi di progettazione che aiutano a garantire la creazione di un sistema sicuro.

Economy of mechanism

I meccanismi di sicurezza dovrebbero essere il più possibile piccoli e semplici.

- L'utente non si scoraggerà e non cercherà di bypassarlo
 - Semplice da mantenere |
 - Semplice da verificare → Meno vulnerabilità
 - Semplice da aggiornare |
- Inoltre, spesso i sistemi troppo complessi si basano su assunzioni che possono essere pericolose: se il sistema viene utilizzato supponendo certe caratteristiche, quando queste vengono a mancare il sistema viene a mancare.

<p><i>Fail-safe defaults</i></p> <p>A meno che a un soggetto non sia dato esplicitamente accesso a un oggetto, l'accesso di default dovrebbe essere negato.</p> <p>Questo è più safe dell'alternativa che in default tutti possano fare tutto e vieto le eccezioni!</p>	<p><i>Complete mediation</i></p> <p>Richiede che se un utente richiede di accedere a una risorsa il sistema verifica che il sistema ha diritto di accedere alla risorsa in quella modalità. Se lo richiede una seconda volta, va comunque riverificato ogni volta.</p>	<p><i>Open design</i></p> <p>La forza del sistema di sicurezza non deve essere nella segretezza del meccanismo.</p> <p>Per esempio, la crittografia è sicura non perché nessuno sa come funziona, ma perché lo è in sé. Questo è importante perché altrimenti la sicurezza sarebbe molto poco stabile: nel momento in cui l'attaccante scopre il meccanismo segreto, vince.</p>
<p><i>Separazione dei doveri</i></p> <p>Un'azione critica deve richiedere la cooperazione di più utenti. Se per fare un'azione critica sono necessari più passi, allora devono essere utenti diversi a eseguire ciascun passo.</p> <p>Questo è utile perché significa che per compiere quell'azione dovrà compromettere più utenti.</p>	<p><i>Separazione dei privilegi.</i></p> <p>Il sistema non dovrebbe dare permessi su una sola condizione.</p> <p>Simile alla separazione dei doveri, ma non solo sugli utenti (allarga a qualunque tipo di condizione)</p>	<p><i>Minimo privilegio</i></p> <p>Gli utenti devono avere il livello minimo sufficiente a compiere le azioni che devono svolgere. Inoltre, i privilegi devono essere dati solo per il tempo necessario a svolgere l'azione.</p>

<i>Mecanismi meno comuni</i> I meccanismi usati per accedere alle risorse non dovrebbero essere condivisibili. Altrimenti potrebbero diventare un canale per lo scambio di informazioni	<i>Accettability psicologica</i> I meccanismi di sicurezza non dovrebbero complicare troppo l'interazione del sistema con l'utente, ondevitare che egli cerchi di aggirare la protezione.	<i>Isolamento</i> Isolano la risorsa da proteggere dagli attacchi. Forma più semplice di protezione. <ul style="list-style-type: none"> • Fisico: non esistono connessioni fisiche fra un accesso pubblico e le risorse • Logico: livelli di sicurezza e altri meccanismi sono responsabili di separare il pubblico e le risorse
<i>Modularità</i> Se ci sono dei livelli di sicurezza da implementare, è bene implementarli bene una volta sola e poi poterli riutilizzare	<i>Layering</i> Aggiungere più livelli di sicurezza, di modo che se un livello è compromesso il sistema non è immediatamente scoperto.	<i>Least astonishment</i> I sistemi di sicurezza sviluppati sono comprensibili a livello di idea dall'utente, così che possa intuirne l'utilità.

Strategie di sicurezza

Una strategia comprensiva di sicurezza include:

- **Specifiche / policy** : cosa deve fare?
- **Implementazione / meccanismi** : come funziona?
- **Correttezza/ assurance** : funziona davvero?

I tradeoff da considerare è che:

1. Semplicità del sistema vs. grado di sicurezza
Vogliamo tenere il sistema semplice da utilizzare ma garantire la sicurezza
2. Costo dell'implementazione, mantenimento, formazione vs. costo dell'attacco da cui ripara

Meccanismi di sicurezza

Sono tutto ciò che possiamo sfruttare per rimanere nello stato sicuro.

- **Prevenzione:** misure che proteggono gli assets preventivamente
- **Detection:** identificare violazioni
- **Risposta:** reazione immediata, bloccando il sistema or w/e
- **Recovery:** come ritornare a uno stato sicuro; non è sempre possibile (es. data disclosure)

Meccanismi di assurance

Chi compra sicurezza vuole essere rassicurato sul fatto che quello che compra garantisce di avere un certo livello di sicurezza; non è semplice misurare la sicurezza.

IT Security management

È qualcosa che va spalmato su tutte le fasi di utilizzo di un sistema software, partendo dall'analisi dei rischi bisogna identificare e prioritizzare le minacce possibili e poi implementare tutte le contromisure che mitigano le minacce.

È un processo ciclico: plan-do-check-act. Sia perché potrebbero nascere nuove minacce, sia perché la detection potrebbe rilevare minacce che non erano state considerate.

2 – USER AUTHENTICATION

L2 – 07/10/2021

Molti dei problemi di sicurezza sono legati a sistemi o processi che compiono azioni non permesse su una risorsa.

Autenticazione

È il processo che **lega un'identità a un soggetto**, dove con soggetto intendiamo le identità attive nel sistema – utenti oppure un processo.

Tipicamente, il soggetto mostra al sistema un'identità e il sistema – per verificare che il soggetto sia detentore di quella identità – chiede uso di alcuni metodi di autenticazione, come per esempio una password.

Metodi di autenticazione

I mezzi di autenticazione usano una di queste quattro strategie, che riguardano qualcosa che il soggetto...

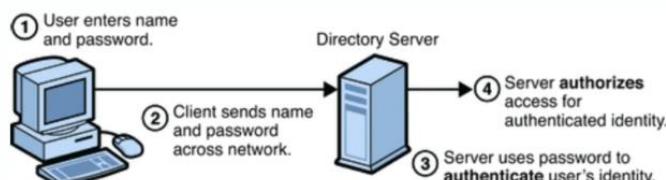
...conosce	...ha	...è (static biometriche)	...fa (dynamic biometrics)
password, pin, domande	dispositivo fisico tipo badge, smart card, physical key	fingerprint, retina, faccia	tono della voce, camminata, pattern vocale.

Autenticazione via password

È il metodo più comune. L'utente fornisce un username e una **password**, dove l'username è l'identità e la password è la verifica. Il sistema deve possedere le informazioni da matchare, e verificare se la PW è corretta.

Password

La password è una **sequenza di caratteri**, associata con un'entità e di cui conferma l'identità, e conosciuta solamente dal sistema e dall'utente.



A volte, oltre alla password, i sistemi possono utilizzare anche delle **informazioni complementari**; sono delle informazioni tipicamente legate al sistema o all'orario di accesso che vengono messe in AND alla correttezza della password. Per esempio, se il sistema è quello dell'azienda, il sistema potrebbe aspettarsi che l'impiegato si colleghi in certi orari specifici.

Comportamenti non ottimali lato user

Se la password fosse ben costituita, sarebbero una metodologia molto efficiente; tuttavia comportamenti non ottimali lo rendono non ottimale.

- **Password overload/reuse:** in media un cittadino deve memorizzare 22 password. Il risultato, quindi è che l'utente usa la stessa password per autenticarsi su più sistemi.
- **Password prevedibili:** tipicamente si usano password molto comuni affinché siano facili da ricordare.

Sembra sicura, ma il 70% degli attacchi prevede di indovinare la password. Un esempio è il botnet mirai, che aveva sfruttato tutti i semplici sistemi IOT in cui veniva lasciata la password di default oppure era molto semplice.

Ciò che rende davvero vulnerabile la password è il fatto che essa è riutilizzabile. Questo può essere affrontato attraverso

- **Password aging:** cambiare password ogni X. Il problema è stabilire ogni quanto; la stima è fatta sul caso pessimo del tempo necessario a indovinarla in media.
 - **Problema:** va garantito che non si possa riusare la stessa password.
- **OTP – one time password:** valida solo una volta, serve un dispositivo che le genera. Il punto è che le password sono slegate, e avendone indovinata una non ha vantaggi sulla successiva.

Protezione lato server

Il server ovviamente deve proteggere inserimento e salvataggio delle password, o ovviamente tutto sarà inutile :)

Inserimento della password

- **Non devono essere spedite in chiaro**, ma crittate; l'idea è che si confronta la versione crittata.
- Inoltre, bisogna fare attenzione a fare sì che i **tempi di risposta siano costanti**: se il tempo di risposta è relativo, ad esempio, al primo carattere sbagliato ecco che erischio di aiutare l'attaccante.

Protezione del sistema

- Devo loggare anche i tentativi falliti, ma **proteggere il log** (perché magari i tentativi falliti sono solo piccoli typos).
- Il file con le password deve essere protetto con crittografia, ed è importante il controllo al file. Possiamo cifrare:
 - **Standard crittografia**: la password è cifrata e poi decifrata per confrontarla col tentativo dell'utente
 - **One way crittografia**: confronta direttamente la password cifrata con la password che arriva; questo ha senso solo se sono certi che non esistano casi di collisione

Ad esempio, nel caso di unix: il sistema prende la password e prende un valore di salt associato all'utente; con questi due verifica se l'hash generato dall'tentativo corrisponde all'hash memorizzato. Il vantaggio è che non sono più in grado di vedere se più utenti hanno la stessa password, e allunga la password.

Craccare una password

È il più semplice, ma ci si può mettere molto; con un alfabeto A e una pwd lunga n , il numero di tentativi necessario è $|A|^n$ nel caso pessimo, ma è facile succeda prima

Attacco a dizionario

Si creano dei **dizionari di possibili password**, con parole pronunciabili o conosciute. Questo restringe il campo, ma non dà garanzia di trovare la password.

- Le **likely password**, che formano questi dizionari, sono parole corte, di uso comune e facili da pronunciare.
- È possibile restringere ancora di più il dizionario usando parole che **hanno a che fare con la sua vita personale**.



Funziona anche per le estensioni: i numeri spesso sono messi a inizio o fine, etc.

Forza di una password

La forza di una password è misurata rispetto a una **stima del numero di tentativi**. Le password possono essere in base alla predicitività da Weak a Strong in base ai pattern comuni.

Zxcvbn è uno stimatore di bontà di password di Dropbox, che considera anche i **pattern della tastiera** e vari altri pattern. Fornta la potenziale password, cerca tutti i potenziali pattern

Pattern	Exemple	
Token	Logitech, parliamentarian	
Reversed	Drowssap	
Sequence	123 2488 jklm	
Repeat	zzz ababab	
Keyboard	qwertyui	
Date	7/8/1947 8.7.47 11.7.21	
Bruteforce	X\$JQhMzt	

Match → Estimate → Search
Input: lenovo1111

lenovo	token	11007 guesses
one	backwords	3284 guesses
no	english	11 guesses
no	backwords	18 guesses
1111	date	2100 guesses
	repeat	48 guesses

Contromisure

Vecchie contromisure

- Lunghezza minima
- minuscole/maiuscole/etc
- Escludere le password più comuni
- Cambio password ogni X giorni
- Password generate dalla macchina

In realtà, il MIST annuncia anche che queste contromisure non sono efficienti.

Quello che funziona è →

Nuove contromisure

- Blocco dopo X tentativi
- Throttling: ritardo fra i tentativi falliti per dilatare il tempo di successo del brute force
- Protective monitoring: avvisare delle nuove autenticazioni
- Blacklist delle password comuni
- 2FA

Keylogger

Tengono traccia di tutti i caratteri, incluse le password.

Intercettazione

È un attacco man in the middle.

Social engineering: leaking users

Inducono l'utente a rilasciare la password ingannandolo o comunque agendo sull'essere umano.

Es. ci si fingeva ministero salute/economia a tema covid e si chiedevano passwords.

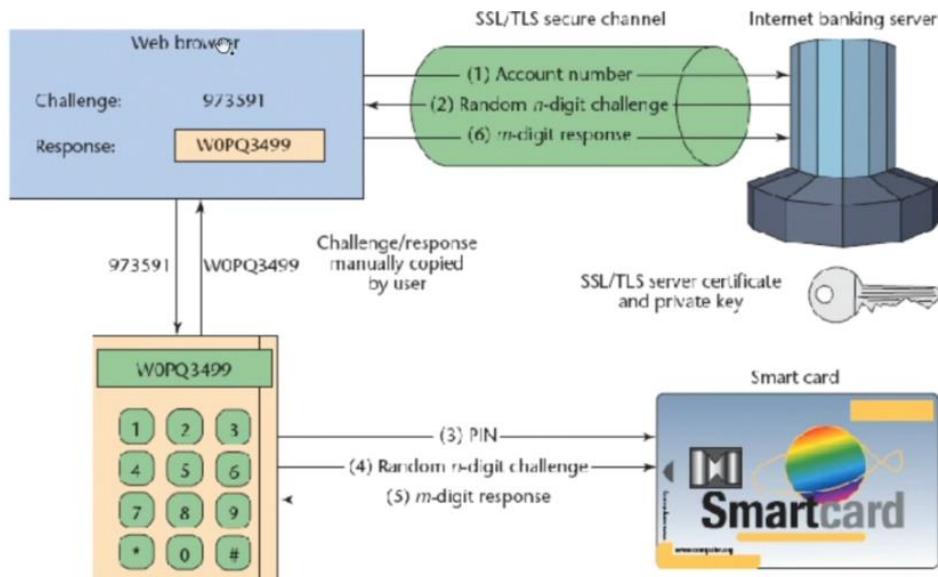
- Foto con i post it attaccati alle machine
- Phishing (most efficace!)
- Shoulder-surfing
- Dumpster-diving
- Countermeasure

L2b - 07/10/2021

Autenticazione token-based (l'utente HA)

Può avere memory cards, smart cards o OTP.

- Esiste anche il **codice a barre**, sia come strisce sia come quadrati; più spesso sono usati per digitalizzare codici velocemente. **Non produce tanta sicurezza** perché può essere replicato, ma è un modo conveniente
- **Memory card**: posso inserire informazioni in modo semplice e anche modificarle, ma non sono molto sicure. Sono solitamente usate in combinazione con altri metodi.
 - Problemi: facilmente modificabili!!
- **Smart tokens**: oggetti con un microprocessore che può generare chiavi, tipicamente OTP oppure può esserci un dialogo col sistema di tipo challenge response: il token viene stimolato dal sistema e il token deve rispondere correttamente; soprattutto viene usata la crittografia.
- **Smart card**: oltre ad avere la memorizzazione di informazione possono anche fare calcoli; hanno dei piccoli processori che possono fare calcoli crittografici in modo veloce, e spesso espongono facilmente il contatto.



- **Tags RFID**: sono dei tags che hanno il vantaggio che il lettore non deve vedere l'RFID, ma basta il segnale radio. È sempre molto utile in logistica. Esistono sia attivi che passivi. Tipicamente funziona con challenge response.

Tutti questi oggetti fisici hanno il problema che **potrei perdere l'oggetto e con esso l'accesso**.

Autenticazione biometrica (l'utente È)

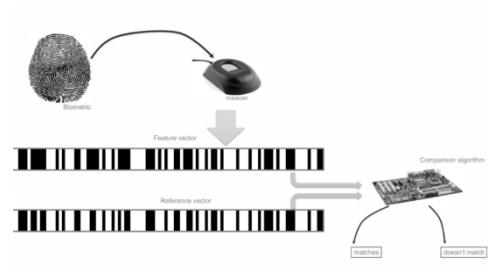
Si basa su qualche caratteristica fisica. In questo caso è richiesto che il sensore possa rilevare quella caratteristica biometrica. Alcuni esempi sono lettori di voce e di impronte.

Per decidere quale caratteristica fisica essa deve essere:

- **Universale**: tutti devono averla
- **Distintiva**: per ognuno deve essere unico
- **Permanente**: non deve cambiare nel tempo
- **Collezionabile**: deve essere determinata e quantificata facilmente.

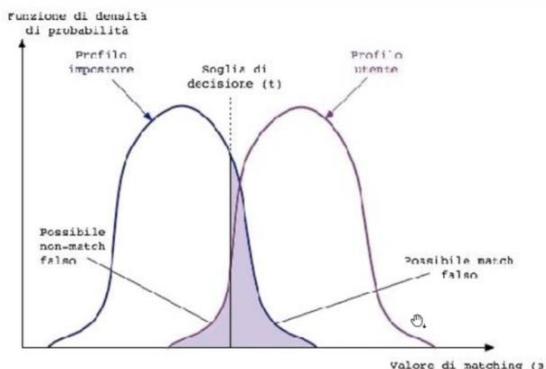
Le operazioni sono:

- **Enrollement**: acquisizione della caratteristica biometrica, che viene tradotta in un formato digitale. Di solito si associa anche un PIN.
- **Verifica**: confronta il template della rilevazione con quello memorizzato. Se c'è un match, viene consentito l'accesso.



Alcune caratteristiche tipiche sono impronta, retina, una parte dell'orecchio, DNA, firma...

Nella realtà succede che nel processo di digitalizzazione si perde informazione, e potrebbero esserci intersezioni fra due utenti. La threshold sul match deve essere quindi piuttosto alta.



Limit

Sono

- Accuracy: voglio evitare la sovrapposizione
- Forging: per esempio, le impronte digitali vengono lasciate un po' ovunque.
- Secondo modo: è sempre necessario un secondo metodo, come per esempio se mi taglio un dito...
- Tolleranza: potrebbe essere scomodo :')

3 – SECURITY POLICIES

Policy e modelli

Possiamo distinguere security policy e security model:

Security model

Procura una **rappresentazione formale e astratta di un insieme di sistemi informatici**, sottolineando le **feature di sicurezza** di cui avrebbe bisogno.

È una descrizione astratta di comportamenti del sistema ed è alla base del design delle policy.

Security policy

Definisce ciò che è permesso e non è permesso.

È analogo a un insieme di leggi, ed è definita in termini di regole ad alto livello e requisiti che dovrebbero essere ben definiti, coerenti ed applicabili.



Soundness

Il sistema soddisfa le proprietà di sicurezza (= tutti gli stati raggiungibili sono autorizzati).

Se un modello è abbastanza formale, allora la soundness è ben definita: $System \models Security\ property$.

Sicurezza di un sistema

Le policy e i modelli sono formalizzate in termini di **stati del sistema**. Lo stato di un sistema colleziona tutti i valori correntemente salvati in memoria, li registra e dà uno snapshot del sistema. Il computer è rappresentato come sistema di transizioni.

Sistema sicuro

È un sistema che, **partendo da uno stato autorizzato, può solo entrare in uno stato autorizzato**.

Il sottoinsieme di informazioni contenute nel sistema che riguarda la protezione è detto **protection state**:

- **File system**: porzione dello stato di sistema che riguarda chi sta leggendo/scrivendo un file, l'access control...
- **Program**: porzione dello stato di sistema che contiene un'immagine run-time dell'esecuzione di un programma e contiene informazioni come il valore del PC, call stack...

Tipi di security policies

Talvolta, essere autorizzati potrebbe dipendere dalla cronologia di computazione, e quindi non dipende solo dallo stato corrente: per esempio il non poter accedere a un oggetto che è stato eliminato. Di conseguenza, è necessario usare una **policy definita sulle tracce di sistema (anziché solo sugli stati)**.

Ne esistono diversi tipi:

- **Security policy militare/governativa**: è sviluppata per dare confidenzialità, e le informazioni sono classificate come *non-classified*, *limited*, *classified*, *secret*, *top-secret*
- **Security policy commerciale**: è sviluppata per dare *integrity*
- **Confidentiality/information-flow policy**: si occupa esclusivamente di confidenzialità
- **Integrity policy**: si occupa esclusivamente dell'integrità.

Access control

Sono dei meccanismi che stabiliscono chi può compiere delle azioni su certe risorse. Un sistema di controllo degli accessi è fatto principalmente da tre elementi:



- **Policies**: cosa ammettiamo che succeda, anche in forma verbosa
- **Modello**: la policy scritta in forma di modello
- **Meccanismi**: implementano la policy di quel sistema.

Le fasi sono:

1. *Autenticazione*

Verifica che l'identità dell'utente sia quella dichiarata

2. Accesso alle risorse

a. *Autorizzazione:*

Concessione di un permesso su un soggetto verso una risorsa

Passa per il controllo degli accessi, che interroga il security manager

b. *Auditing*

Monitoraggio e processing degli accessi di un utente al sistema.

Passa per l'audit, che colleziona le azioni che accadono nel sistema.

Uno degli aspetti delicati è capire di quali dati io voglio tenere traccia.

Il riconoscimento automatico delle intrusioni può avvenire **offline** o anche **in real time**, mettendo subito in atto meccanismi di risposta. Deve quindi decidere **cosa monitorare** e **come/quando reagire**.

Ci sono due approcci possibili:

- **Detection di uso non ammesso:** riconosco qualcosa che non può accadere (es. sequenza di azioni) come attacco
 - Più efficace
 - Potrebbe non riconoscere nuove violazioni
- **Anomaly detection:** allenano il sistema a riconoscere i comportamenti normali e rilevano un comportamento inusuale.
 - Potrebbe riconoscere anche attacchi nuovi
 - Potrebbe avere falsi positivi

Proprietà da soddisfare

Reliable input

Assumiamo che l'autenticazione abbia funzionato, e che gli **input** siano **sinceri**.

Supporto per la granularità

Un meccanismo è tanto migliore quanto più permette di **specificare a basso livello** le azioni che l'utente può compiere.

Need-to-know / minimo privilegio

Ciascuna entità di sistema deve avere il **minimo di autorizzazioni** e risorse per fare il proprio lavoro

Policy open-close

Una policy è **open** se specifica quello che non è permesso (tutto è permesso tranne), mentre è **close** se è al contrario (non puoi fare niente tranne)

Separation of duty

Dividere i passi delle funzionalità di un sistema fra **individui diversi**

Policy amministrative

Bisogna specificare **chi può estendere privilegi** sugli oggetti.

Risoluzione dei conflitti

Se ho più policy, potrei avere **conflitti** e devo decidere come gestirle. Generalmente la cosa più sicura è privilegiare la policy più stretta.

Attori

Soggetti

Sono tutte le **entità in grado di accedere alle risorse**; sono solitamente utenti e processi.

Ci sono tre tipi di soggetti:

- **Owner:** sono i proprietari; hanno i permessi più ati e solitamente possono estendere i privilegi
- **Group:** è possibile raggruppare un insieme di utenti decidendo i loro diritti in quanto di un gruppo
- **World:** gruppo che ha il minor numero di diritti grantiti sull'oggetto che si sta considerando.

Oggetti

Entità/risorse di cui il meccanismo di controllo degli accessi vuole controllare l'accesso.

Privilegi

Sono i modi in cui un soggetto può accedere a un oggetto.

- **Lettura:** può visionare, copiare, stampare la risorsa
- **Scrittura:** modifica dell'oggetto; la cancellazione può non essere separata
- **Cancellazione**
- **Esecuzione**
- **Creazione**
- **Concatenazione**
- **Ricerca**

3B - POLICY DI CONTROLLO DEGLI ACCESSI

L3b_13/10/21

Ci sono diversi tipi:

Discretionary access control (DAC)

La scelta è fatta **in base all'identità del soggetto**; sono i soggetti a decidere chi può accedere; modello flessibile.

Mandatory access control (MAC)

L'accesso è concesso **su livelli di sicurezza** dei soggetti ed oggetti.

Role based access control (RBAC)

Evoluzione del modello DAC che cerca di **limitare gli svantaggi del modello DAC** e si basa sul concetto di **ruolo**. Il ruolo è tipicamente legato al tipo di funzioni che l'utente deve compiere nel sistema in cui si autentica.

Attribute based access control (ABAC)

Ci sono aspetti non gestibili col RBAC, quindi viene sviluppato in cui l'accesso basato sugli **attributi di identità del soggetto, oggetto e ambiente**.

DAC - Discretionary Access Control

Gli utenti possiedono le risorse e controllano l'accesso alle risorse.

- L'accesso è concesso basandosi **sull'identità dell'utente**.
- **Discrezionale** significa che l'utente che ha diritto di accesso può cedere questo diritto a un altro utente, così come lo può revocare. Sono gli utenti **proprietari** delle risorse a decidere chi e come può accedere alle risorse.
- Tutte queste informazioni sono salate nella **matrice di controllo degli accessi**, che a sua volta è un oggetto su cui alcuni utenti sono autorizzati o meno ad accedere.

E' un meccanismo **molto flessibile**, quindi se non è usata in modo coscienzioso **può portare a situazioni non desiderate**.

Caratteristiche di questo modello:

- **Basato sull'identità**: è il soggetto che mi dice cosa può o non può fare
- **Molto flessibile**, quindi per usarlo correttamente deve **essere compreso dagli utenti** ondevitare di avere un sistema con più privilegi del necessario
- Non ha **alcun controllo sul flusso di informazione**.

Matrice di controllo degli accessi

I meccanismi DAC si basano sulla **matrice di controllo degli accessi**, aka una matrice dove ho **una riga per utente/soggetto** e **una colonna per ogni risorsa/oggetto**.

subjects	objects		
	news.doc	photo.png	fun.com
alice	read	view	view
bob	read write	view edit	view modify
charlie			
dave		view	

Anche la **matrice di controllo degli accessi è un oggetto**, e i cui accessi sono regolamentati. Tutti questi accessi interrogano l'access matrix monitor per sapere se l'azione può essere effettuata, e il controllore verifica sulla matrice se l'utente è o meno autorizzato.

Il problema di questo metodo è che **non scala**: diventa difficile mantenere tutto tramite questa singola matrice.

Per cercare di renderla più gestibile ci sono alcune tecniche:

Access control list

Proiezione sugli oggetti; per ogni oggetto viene specificato chi può accedere e come.

- + L'Access Control List può essere aggregata all'oggetto come metadata.
- Se volessi eliminare un utente totalmente dovrei andarlo a cercare in tutte le liste...

	news.doc	photo.png	fun.com
alice	read write	view edit	view
bob		view edit	view modify
charlie			
dave		view	

	news.doc	photo.png	fun.com
Alice's capability	read	view, edit	view
Bob's capability	read write	view, edit	view, edit

Proiezione sulle righe, quindi **per ogni utente ho tutto quello che può fare**. Anche qui ovviamente ignoriamo le celle vuote, e questo rende molto conveniente eliminare un utente.

Limiti

L'idea era di avere un modello formale che permetesse di fare dimostrazioni sulle proprietà. È un meccanismo flessibile, ma è **prone agli errori**: è necessario che gli utenti che lo utilizzano lo comprendano appieno, dato che **non c'è nessun controllo sul flusso dell'informazione**.

Esempi di DAC-policies

I primi modelli sviluppati sono degli anni 70; l'idea era provare a **definire un modello formale** del meccanismo di controllo per capire se era possibile garantire la policy che si desiderava ottenere.

Definiamo: Soggetti S , Oggetti O , access rights R , matrice M .

Graham Dennins Model (1972)

Si basa sul concetto delle **protection rules**, ovvero definisce una serie di regole.

E' il modello da cui hanno preso ispirazione i modelli successivi. Gli oggetti hanno:

- **Proprietari** (soggetto che possiede l'oggetto)
- **Controllori** (soggetto che controlla un soggetto).
- **Diritti di accesso:** diversi modelli identificano diversi diritti. I diritti sono:
 - a. Creare/cancellare oggetti
 - b. Creare/cancellare sogetti
 - c. Avere accesso in lettura
 - d. Dare diritti di accesso
 - e. Cancellare diritti di accesso
 - f. Trasferire diritti di accesso

I soggetti sono visti come oggetti, aka **possono essere creati** e sono presenti come colonne. Tuttavia, cancellare un soggetto è azione esclusiva del controllore (e si cancella la riga del soggetto).

Rule	Command (from s_0)	Condition	Operation
R1	create o	--	add column for o in M; insert 'owner' in $M[s_0,o]$
R2	delete o	'owner' in $M[s_0,o]$	deletes column for o in M
R3	create s	--	add a row for s in M; insert 'controller' in $M[s_0,s]$
R4	delete s	'control' in $M[s_0,s]$	delete row fro s in M
R5	read the access right of s on o	'control' in $M[s_0,s]$ or 'owner' in $M[s_0,o]$	copy A[s,o] in s_0
R6	grant r [r^*] for s on o	'owner' in $M[s_0,o]$	add r [r^*] in $M[s,o]$
R7	delete r from s on o	'control' in $M[s_0,s]$ or 'owner' in $M[s_0,o]$	remove r in $M[s,o]$
R8	transfer r [r^*] for s on o	'r*' in $M[s_0,o]$	add r [r^*] a $M[s,o]$

L'asterisco sulla regola (es. r^*) significa che anche il soggetto a cui ho passato i diritti **può darli a qualcun altro** (aka può copiarli un'altra volta, può concederli ad altri). Insomma, a furia di r^* si rischia di **perdere il controllo** di chi può scrivere.

Harrison-Ruzzo-Ullman Model (HRU)

Cerca di rispondere alla domanda "**S will access O**"? Ovvero, tutte le regole che creano, potranno arrivare a un certo punto a creare la condizione che S accede a O?

Questo ha senso, perché alla fine quello che vogliamo è essere certi che i soggetti *non* accedano a cose dove non vogliamo che accedano.

Definiamo:

- $X_0 = (S_0, O_0, M_0)$ il nostro **stato iniziale**, con M matrice degli accessi
- **Un insieme di comandi primitivi che alterano l'accesso** alla matrice degli accessi M :
 - Create (subject) s , Create(object) o
 - Distruggi (soggetto) s , Distruggi (oggetto) o
 - Inserisci (diritto) r in $M[s,o]$, cancella (diritto) r dalla matrice $M[s,o]$
- **Un insieme di transizioni di stato** rappresentato come t_1, t_2, t_3 e gli stati successivi rappresentati come X_1, X_2 dove la notazione $X_i \vdash_{t(i+1)} X_{i+1}$ significa che la transizione di stato t_{i+1} muove il sistema da uno stato all'altro.
Le transizioni di stato sono **modellate dall'esecuzione dei comandi**, dove S , O e M sono prima dell'esecuzione e S' , O' , M' sono dopo l'esecuzione.

Operation	Conditions	New State
enter r into $M[s,o]$	$s \in S, o \in O$	$S' = S$ $O' = O$ $M'[s,o] = M[s,o] \cup \{r\}$ $M'[s_0,o] = M[s_0,o] \setminus (s_0,o)$
delete r from $M[s,o]$	$s \in S, o \in O$	$S' = S$ $O' = O$ $M'[s,o] = M[s,o] \setminus \{r\}$ $M'[s_0,o] = M[s_0,o] \setminus (s_0,o)$
if $s \notin S$ or $o \notin O$		then $(S', O', P') = (S, O, P)$ and the request fails
create (subject) s'	$s' \notin O$	$S' = S \cup \{s'\}$ $O' = O \cup \{s'\}$ $M'[s,o] = M[s,o] \forall s \in S, o \in O$ $M'[s',o] = \emptyset$ per $s \in S$ $M'[s,s'] = \emptyset$ per $s \in S$
create (object) o'	$o' \notin O$	$S' = S$ $O' = O \cup \{o'\}$ $M'[s,o] = M[s,o] \forall s \in S, o \in O$ $M'[s,o'] = \emptyset$ per $s \in S$
destroy (subject) s'	$s' \in S$	$S' = S \setminus \{s'\}$ $O' = O \setminus \{s'\}$ $M'[s,o] = M[s,o] \forall s \in S, o \in O$
destroy (object) o'	$o' \in O$ $o' \notin S$	$S' = S$ $O' = O \setminus \{o'\}$ $M'[s,o] = M[s,o] \forall s \in S, o \in O$

Dunque, i comandi sono composti da una serie di condizioni da soddisfare e, se sono soddisfatte, una serie di operazioni da fare. Questo rende più flessibile implementare una policy di controllo degli **accessi**.

Questi comandi vanno a impattare la matrice di controllo degli accessi; si capisce che il modello è **più flessibile** poiché **siamo noi a definire quali siano le condizioni: non è solo il proprietario come prima**, dipende da quello che mettiamo nelle condizioni.

Questi modelli nascono con l'obiettivo di definire se è possibile arrivare in un o stato in cui soggetti che non dovrebbero accedere riescano a accedere.

(NO DETTAGLIO TEOREMI) → Creando modelli formali di controllo degli accessi posso dimostrare formalmente se una certa condizione si verifica in un futuro. !! Il problema in generale non è decidibile; lo diventa se limitiamo il modello a comandi monooperazionali o limitando il numero di soggetti.

Take-Grant systems

E' un modello che lavora sui grafi e rappresenta graficamente gli accessi agli oggetti.

Abbiamo solo 4 operazioni fatte sul grafo:

- Create
- Remove
- Take(t) (tolgo diritti)
- Grant(g)

Rappresento le risorse come pallini, e gli archi sono i permessi che la sorgente ha sulla destinazione. Per esempio, nella creazione create(y), parto solo da un pallino x e arrivo a una condizione x->y, perché x ha i diritti su y.

command name (o₁, o₂, ..., o_k)

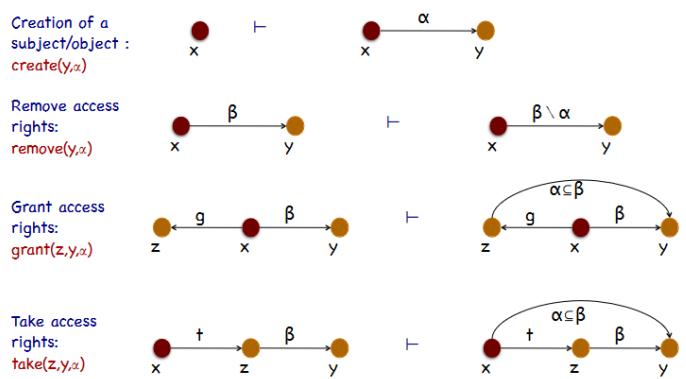
if r₁ in A[s₁, o₁]
r₂ in A[s₂, o₂]
...
r_m in A[s_m, o_m]

then
op₁
op₂
...
op_n

end

✓ When **s creates** a file **f**, **s** is the owner of **f** with read and write access on the file

✓ The owner **s** of the file can **grant** read access to another subject **p**



Anche in questo caso è possibile stabilire la verità del predicato can-share;

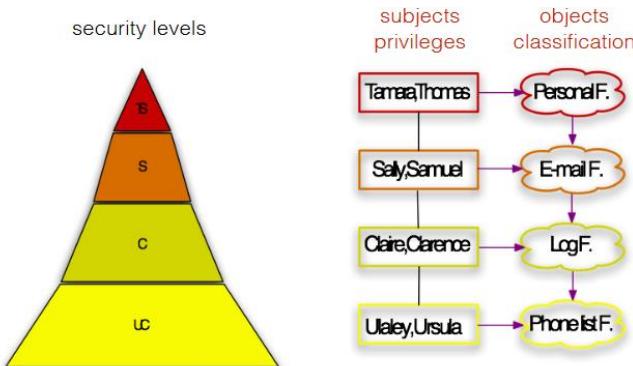
can-share(α, x, y, G_0) = true se esiste una sequenza di grafi di protezione tali per cui $G_0 \vdash G_n$ usando solo le 4 regole di base, e in G_n c'è un arco fra x e y.

Mandatory access control MAC

Le decisioni vengono prese in base al grado di sicurezza del soggetto e dell'oggetto; è tipico di sistemi multilivello tipo quello militare.

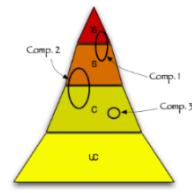
Non è più il proprietario degli oggetti ad avere la possibilità di decidere chi può leggere oggetti e soggetti, ma è deciso **centralmente** in base ai **livelli di sicurezza** assegnati.

E' **meno flessibile**, poiché **non sempre si può modificare il livello di sicurezza**; è più statico.



Per raffinare il modello e applicare il concetto del need-to-know, oltre ai livelli vengono introdotti i **compartimenti** – che sono restrizioni dei livelli. Possono essere anche sovrapposti fra più livelli!

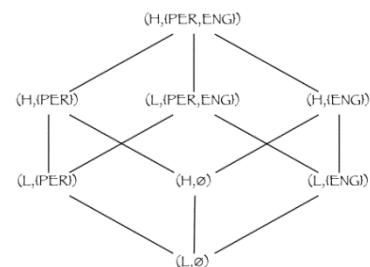
Quindi, la possibilità di accedere è data da **compartimenti** e **livelli**; puoi essere alto in grado e leggere solo parte delle cose top secret (tipo solo alla missione corrente). La classe di sicurezza diventa una coppia, ovvero **classe = (livello, compartimento)**.



- Un oggetto **domina** un soggetto se il livello del soggetto è minoreguale di quello dell'oggetto, e il compartimento del soggetto è minoreguale del livello dell'oggetto.
- Al contrario, un soggetto **può leggere** un oggetto se il livello di sicurezza del soggetto è almeno quello di o, e il soggetto è in tutti i compartimenti a cui appartiene l'oggetto.

Il fatto che sia una coppia fa sì che **l'ordinamento non è più lineare, ma un reticolo di coppie**.

→ No reads up, no write down per evitare che qualcuno possa copiare oggetti su livelli minori!



Modelli

I modelli di controllo degli accessi possono essere classificati anche se:

- Si focalizzano su **integrità** (Bell-Lapadula), **confidenzialità** (Biba, Clark-Wilson) o **entrambi** (Chinise Wall).
- Possono essere applicati solo a ambienti con **policies statiche** (BellLapadula) o permettono **cambiamenti dinamici** nei diritti di accesso (Chinise Wall)
- Si applicano a **ambienti militari** (Bell-Lapadula) o a **ambienti commerciali** (Clark-Wilson, Chinise Wall)
- Sono **informali** (Clark-Wilson) o **formali** (Bell-Lapadula, Harrison-Ruzzo-Ullman)

Modello Bell-Lapadula

È il primo modello MAC ed è quello “tipico”. Si basa sulla security clearance e sulla classificazione degli oggetti, e usa la matrice degli accessi. **Obiettivo: garantire la confidenzialità**, ovvero l'importante è che le cose segrete non devono essere lette da chi è più in basso.

Proprietà:

- **BLP Simple security property: no reads up:** un soggetto può leggere un oggetto solamente se il grado del soggetto è maggiore uguale a quello dell'oggetto. (chi sta sotto non potrà mai leggere cosa sta sopra). Il problema è che solo con questa regola un soggetto a livello alto potrebbe copiare i contenuti dei files di grado basso!
- **BLP Star Property: no write down:** posso scrivere in un oggetto solo se quell'oggetto ha un livello di segretezza maggioreguale. (chi sta sopra non può scrivere sotto, dato che poi gli altri potrebbero leggerlo)
- **BLP DS-propriety:** un soggetto può dare accesso a un oggetto secondo le regole MAC (non leggere in alto e non scrivere in basso) e se questo è concesso dalla matrice di controllo degli accessi. Quindi un soggetto può usare solo gli accessi per i quali ha le autorizzazioni MAC.

Un sistema che ha tutte queste tre proprietà è detto **sicuro**.

Vantaggi/svantaggi:

Vantaggi	Svantaggi
<ul style="list-style-type: none"> • Lo stato BLP describe tutti i permessi correnti • Le policy sono basate su security levels che possono essere cambiati al bisogno • Se cambiamo le proprietà possiamo garantire anche l'integrità 	<ul style="list-style-type: none"> • Fa solo confidentiality ma non integrity • Non si occupa della gestione del controllo degli accessi • Ha dei buchi: un soggetto basso crea un oggetto, un complice a livello alto innalza il livello dell'oggetto oppure non fa niente. Se il tizio di livello prova a leggere il suo oggetto, scopre che esso ha cambiato di livello perché fallisce la scrittura; questa informazione è passata a livello inferiore.

Modello Biba (1977)

È il duale di BLP. **L'obiettivo è garantire l'integrità: prevenire le scritture non desiderate.** Qui abbiamo l'integrità dei soggetti/oggetti, che misura quanto io mi posso fidare di essi.

Proprietà:

- **Simple Integrity Property:** no write up -> un soggetto con un basso livello di integrità non può andare a scrivere sopra, perché comprometterebbe l'integrità.
- **Integrity Star Property** -> **No letture verso il basso:** se il soggetto leggesse contenuti non integri "sporcherebbe" la sua integrità!
- **Integrity policy:** un soggetto può invocare un altro soggetto solo se il soggetto invocato è a livello minore . Una variante permette di reggere oggetti a qualsiasi livello di integrità, ma nel momento in cui un soggetto legge un oggetto di livello minore, il soggetto viene declassificato.

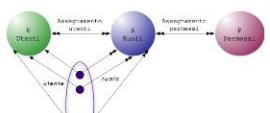
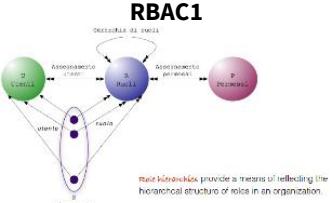
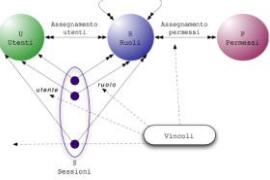
Role Based Access Control - RBAC

E' l'evoluzione di DAC che cerca di **risolvere il problema della numerosità degli utenti definendo dei ruoli.**

Più utenti possono essere assegnati a un ruolo, e un utente può avere più ruoli. **Ha senso quando ci sono molti meno ruoli che utenti.**

Il ruolo rispecchia il ruolo degli utenti all'interno del sistema; quindi sono meccanismi che si prestano bene ad implementare i principi del **least privilege** e del **need to know**.

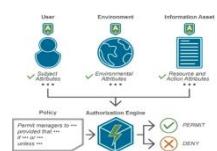
E' stato definito per la prima volta nel 1996 e abbiamo tre modelli:

 <p>RBAC0</p> <p>Diagram illustrating RBAC0: A central oval labeled 'Ruoli' (Roles) is connected to two sets of nodes. On the left, a green oval labeled 'Utenti' (Users) has arrows pointing to 'Ruoli' labeled 'Assegnamento utenti' (User assignment). On the right, a pink oval labeled 'Permessi' (Permissions) has arrows pointing to 'Ruoli' labeled 'Assegnamento permessi' (Permission assignment). Below these connections is a dashed oval labeled 'Sessioni' (Sessions).</p>	<p>Abbiamo utenti, ruoli, premessi e sessioni.</p> <p>Gli utenti sono assegnati in una relazione molti a molti nei ruoli; i permessi sono specificati sui ruoli, sempre in molti a molti.</p> <p>Le sessioni indicano un legame tra utenti e ruoli in un momento; è il concetto temporale.</p> <p>Ogni utente ha uno o più ruoli in una certa sessione; ad esempio, un medico potrebbe essere medico in una sessione e paziente in un'altra.</p>
 <p>RBAC1</p> <p>Diagram illustrating RBAC1: Similar to RBAC0, it shows 'Utenti' (Users) and 'Permessi' (Permissions) connected to 'Ruoli' (Roles). A note at the bottom states: 'Nota: la gerarchia di ruoli provvede a mettere in riflessione l'ineredità dei diritti in base alla gerarchia.' (Note: role hierarchy reflects the inheritance of rights based on the hierarchy.)</p>	<p>Aggiungiamo una gerarchia di ruoli nel modello; la gerarchia ci dice come i diritti engono ereditati in accordo con la gerarchia. (quindi specifichiamo per ogni ruolo solo quelli "in più"!).</p>
 <p>RBAC2</p> <p>Diagram illustrating RBAC2: Similar to previous models, it shows 'Utenti' (Users) and 'Permessi' (Permissions) connected to 'Ruoli' (Roles). A note at the bottom states: 'Nota: si possono impostare vincoli di numerosità, sia generali che per sessione, o vincoli in generale: es. non posso attivare più di N ruoli.'</p>	<p>Possiamo porre dei vincoli di numerosità, in generale o per sessione, o vincoli in generale: es. non posso attivare più di N ruoli.</p> <p>Il vantaggio è diminuire il numero di utenti, quindi posso tornare a matrice di controlle degli accessi ma scala meglio.</p>

Attribute based access control (ABAC)

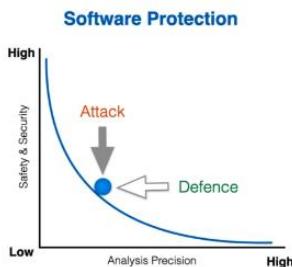


Ci sono delle situazioni nelle quali **RBAC non è sufficiente**; vorrei ad esempio che diverse persone posano accedere alla cisa in base a situazioni particolari (es. orario, che dispositivo...). Questo non è fattibile con RBAC, ma si fa con gli **ABAC**: il meccanismo di autenticazione guarda **attributi del soggetto, dell'ambiente, e sulla risorsa/azione da svolgere**. Ci saranno delle policy in funzione di questi attributi che permettono di definire se concordare l'accesso o meno.



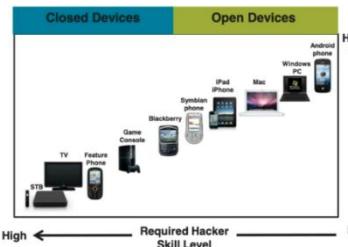
4 - SOFTWARE PROTECTION

L4 19/20/21



Parliamo di protezione del software quando il prodotto che si vuole proteggere è proprio il software. (ok sis,,,))

Diventa un problema reale e riconosciuto perché **il software è inevitabilmente eseguito in un ambiente non sicuro**, e questo può dare problemi di mancati guadagni. Col passare del tempo, i dispositivi stanno diventando sempre più facili da craccare.



Software protection

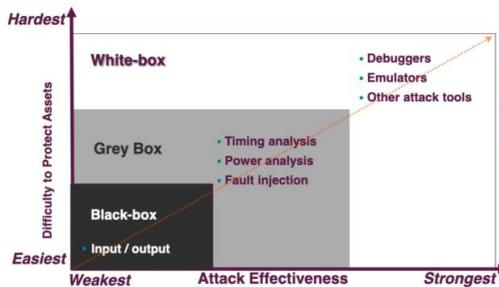
È il desiderio di **nascondere e proteggere qualcosa in un'applicazione** che viene eseguita su un malicious host.

Una prima idea potrebbe essere usare la crittografia. **Non funziona**, perché **prima o poi quel codice sarà decodificato ed eseguito**. Quello che servirebbe fare è la **white box cryptography**, ovvero crittare le cose ma lasciandole eseguibili. Non è ancora possibile.

Possiamo fare un'analogia con la **steganografia**, che è la scienza che studia **come nascondere l'esistenza stessa del messaggio**. È il cosiddetto **prisoner problem**: se ho due prigionieri che si vogliono scambiare un messaggio, e usare una guardia deve trasportare il messaggio, non è sufficiente che il messaggio sia cifrato: non si deve accorgere di starlo portando.

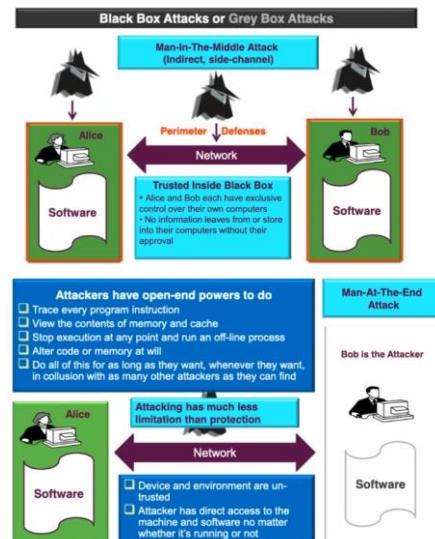
Tipi di ttacchi

Gli attacchi sono classificati in base a **quello che l'attaccante può vedere**, nel seguente modo:



Black-box

In passato, il principale scenario era quello del cosiddetto **man in the middle**: non si attaccano le macchine, ma **solo la comunicazione**.



White-box

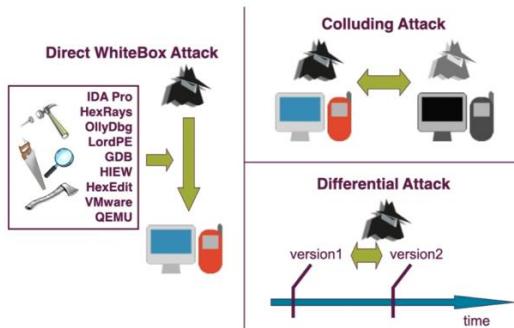
L'attaccante è uno dei due utenti! Ha accesso a tutto ciò che gira sulla macchina, e può usare molti strumenti per capire la trasmissione. Non c'è limite alla sua fantasia :)

Ambienti non fidati

Tradizionalmente, la sicurezza è sempre stata su ambienti fidati; in software security, invece, **l'ambiente non è fidato**.

Minacce

Sono tutte le **tecniche di reverse engineering**: debugger, etc. che supportano l'analista nel comprendere il comportamento dell'applicazione.



Attacco collusivo

Se avessi più di una copia del software potrei **confrontare cosa succede nelle diverse copie** e avere delle **informazioni extra** rispetto a quelle che ho dalla singola

Attacco differenziale

Se avessi a disposizione **versioni successive dello stesso software**, potrei andare a capire delle informazioni in più analizzando le modifiche.

Challenges

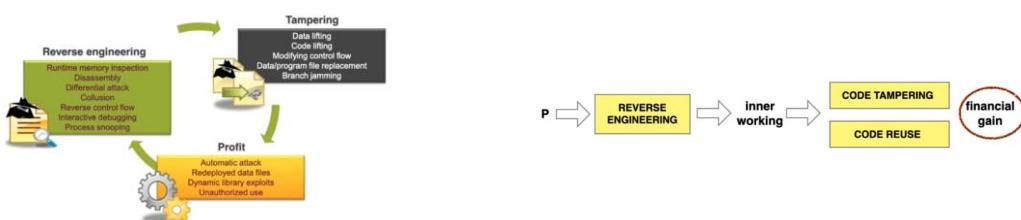
La sfida è che l'utente possa **interfacciarsi a un'altra macchina e fidarsene**, come se la macchina degli altri utenti fossero eseguite in un ambiente fidato.

Vorrei annullare il vantaggio per l'attaccante di essere ingraido di controllare la macchina su cui è in esecuzione il software. Ovviamente posso complicare la vita quanto voglio, ma comunque l'attaccante vede e interagisce con l'esecuzione, e non sarà mai una vera black box. Non esiste una soluzione perfetta.

Attacchi al software

Avvengono su tre fasi che si ripetono:

- **Reverse engineering:** bisogna capire come funziona il software, usando tutti gli strumenti a disposizione: disassemblaggio, attacco differenziale...
- **Tampering:** si mette mano al codice con data lifting, code lifting, slice, modifica del flow...
- **Profit** ☰ ☱ ☲

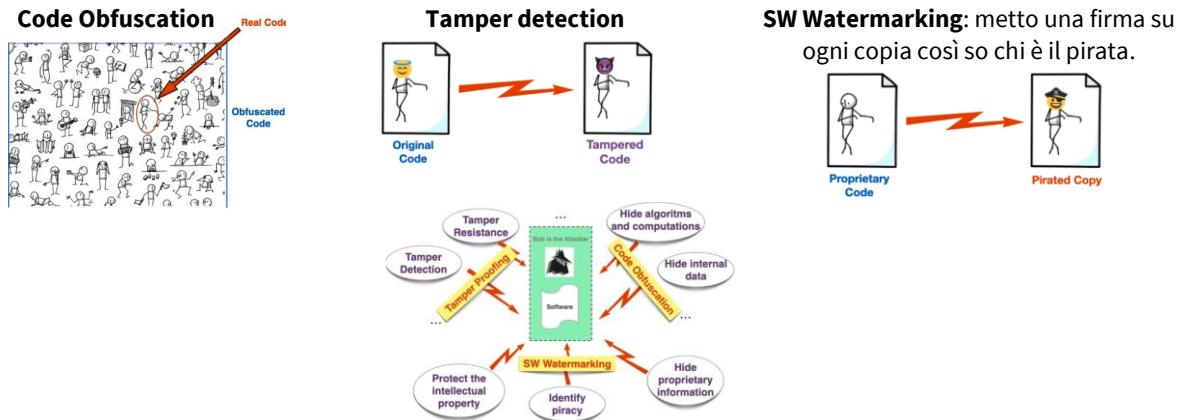


Obiettivi chiave

- **Rendere il codice di difficile comprensione**, modo che per l'attaccante sia il più possibile difficile comprendere il codice. Idealmente vorremmo fosse così complicato che fa prima a svilupparselo lui.
- **Ostacolare e rilevare le manomissioni al codice** (tampering)
- **Resistere al cloning**: praticamente vogliamo resistere alla pirateria, e alla diffusione non autorizzata del codice proprietario.
- **Resistere allo spoofing**: codice eseguito senza autorizzazione per esempio perché ci connettiamo con credenziali non vere

- Se nel codice ci sono dei segreti (es. chiavi), vogliamo proteggerle.
- Stabilire delle risposte all'identificazione di una violazione.

Quindi, riassumendo: manomissione, pirateria e segretezza. per rispondere a questi problemi ci sono tre soluzioni software: offuscamento del codice, tamper proofing, SW watermarking.



Si stima che il costo del software piratato sia sopra i 50 miliardi di dollari. (godo)

Reverse engineering

Il reverse engineering si fa **a tutti i livelli**; contrastare il reversing significa cercare di farlo a tutti i livelli. Poi in base alla realtà effettiva si capisce quale sia il livello più importante.



Per esempio, c'è un filone di ricercatori che sostiene che non abbia senso parlare di offuscamento a livello di codice sorgente, perché interpretano questo solo come il processo che impedisce di estrarre il codice sorgente dall'eseguibile.

A lei era successo che ha collaborato con un'azienda che doveva dare il codice sorgente ai clienti, perché altimenti non si fidavano. Il loro know-how era espresso in alcuni pesi, che loro quindi volevano nascondere dai competitor. Fu un problema, perché la maggior parte degli offuscatori lavorano a basso livello, e quasi nessuno decente ad alto livello.

Tuttavia, dato abbastanza tempo impegno e determinazione, ogni programmatore può fare reversing. E' impossibile rendere impossibile il funzionamento dell'applicazione.

Analisi statica	Analisi dinamica
L'analisi statica analizza e estrae informazioni dal codice senza eseguirlo , ragionando su cosa accadrà in tempo di esecuzione.	L'analisi dinamica esegue: prende come input il programma e un possibile insieme di input, e analizza il programma eseguendolo con quell'insieme di input.
Chiaramente, non mandandolo effettivamente come in esecuzione, vede come possibili più cose che potrebbero accadere ; ad esempio, se c'è un if, vede entrambi i rami (nonostante magari uno dei due rami è sempre vero).	È non conservativa : ritorna solo output e comportamenti che effettivamente accadranno, tuttavia c'è un problema di coverage e il risultato dipenderà dalla completezza dei dati di input.

È definita come **conservativa**: vede tutte le cose possibili.

Static Analysis: Input 'P' leads to a box containing: DISASSEMBLY, DATA FLOW ANALYSIS, CONTROL FLOW ANALYSIS, SLICING, DECOMPILE, and THEOREM PROVING. An arrow points from this box to a circle labeled "CONSERVATIVE INFORMATION".

Dynamic Analysis: Input 'P' leads to a box containing: DISASSEMBLY, DEBUGGING, SLICING, TRACING, EMULATORS, and PROFILING. An arrow points from this box to a circle labeled "NON-CONSERVATIVE INFORMATION".

Tecniche di protezione

Possiamo procedere con le tre tecniche viste prima, come code obfuscation sw watermarking e tamper proofing, oppure possiamo anche pensare alla legge attraverso la tutela intellettuale del codice.

Soggetti interessati

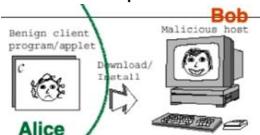
- **Aziende:** tutte le aziende di software, ma in generale anche aziende che producono macchinari che vogliono proteggere i loro segreti!

- **Università:** ricercatori interessati a queste problematiche
- **Militari:** sono interessati alla sefretezza
- **“Bad Guys”**, gli attaccanti.



Malicious host vs Malicious software

Quando si parla di sicurezza bisogna capire chi buono e chi è cattivo. (maia crimew è buona favvanculo)

Malicious host	Malicious software
<p>È quando la macchina (cattiva) cerca di ottenere accesso privilegiato ad un'applicazione (buona), normalmente per violarne la proprietà intellettuale.</p> <p>La protezione è usare SW protection technique per difendere il prodotto.</p> 	<p>È un programma (cattivo) con intento malevolo che si propaga senza consenso dell'utente (buono) e fa dei danni.</p> <p>La protezione è aggiungere livelli di protezione per evitare che qualcuno entri nel suo sistema.</p> 

Sono due facce della stessa medaglia: quel che da un lato è usato in difesa, dall'altro è l'attacco.

Approcci per ostacolare il reverse engineering

Le alternative all'offuscamento sono:

- **HW devices**: per far funzionare il software deve esserci un dispositivo hardware. Non è molto portatile, ma è una strada.
- **Encryption 🔎**: il problema è l'account MATE (man at the end), che può sniffare il codice quando prima o poi, necessariamente, esso comparirà non crittato in memoria.
- **Remote execution 🖥️**: il codice è eseguito su un server, il cliente riceve solo i risultati. È molto molto sicuro, ma richiede connessione al server e potrebbe dover usare una grande quantità di dati.
- **Partial remote execution 🖥️**: È una via di mezzo. Tengo gli aspetti più caratteristici e importanti sul server, e mando al client solo parte dell'esecuzione (col codice che non contiene i segreti). È una buona soluzione, ma bisogna identificare le parti importanti del codice e le altre sono sempre soggette a MATE, oltre al fatto che è richiesta connessione costante.
- **Full code deployed 🖥️**: mando tutto il codice dal server, e ogni volta viene offuscato in maniera diversa. Questo aiuta a rendere più difficile il MATE, ma con una bassa diversificazione rimane molto high risk.

SW Piracy

Per la pirateria, quello che si vuole prevenire è che una copia venduta a un otente possa essere copiata illecitamente.

Quindi, per ostacolare ciò si usa il **SW Watermarking**, che scoraggia il furto e permette di dimostrare che è avvenuto. Quello che si fa è aggiungere una firma al SW, che deve essere

- Difficile da rimuovere
- Stealthy
- High bit-rate
- No extra performance.

Ha a che fare con la steganografia, ovvero il voler nascondere un messaggio in un messaggio.

Anti SW Tampering

Si basa sull'ipotesi che l'attaccante possa ottenere dei vantaggi facendo eseguire un codice diverso da quello originale. Una cosa che si fa, per esempio, è che se uno prova a saltare il controllo licenza il programma crasha.

Per fare questa cosa, quindi, possiamo individuare due fasi:

- **CHECK:** procedura che verifica l'integrità del codice
Per controllare l'integrità del programma ci sono tre modi:

- Controllo il codice, tipicamente attraverso un hash
- Controllo del risultato (challenge-response): non controllo il codice come è scritto, ma controllo che quando viene stimolato risponde correttamente
- Controllo dell'ambiente: è la cosa più complicata; verifica, ad esempio, di non essere eseguita su un emulatore.

A seconda del momento di tempo in cui riesco a rilevare la manomissione, il tool di tampering detection sarà più o meno preciso. Ovvero, il tool sarebbe accorgersene subito; se identifico la manomissione dopo che è avvenuta, va bene; il caso peggiore è quando viene identificata *dopo* che il codice manomesso è stato eseguito.

- **RESPOND:** Risposta alla violazione dell'integrità. Possiamo decidere di:

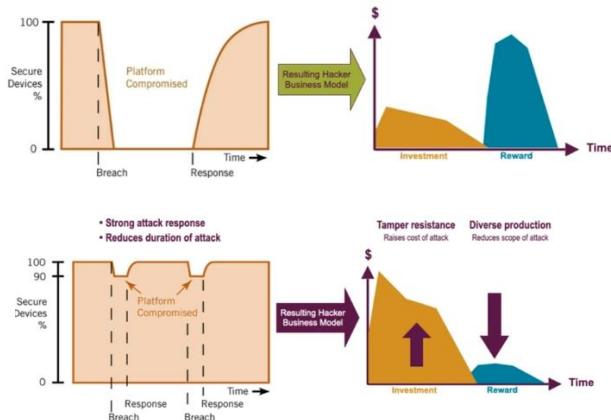
- Terminare l'esecuzione
- Patchare il codice
- Reportare l'attacco

5 - OFFUSCAMENTO

L5 25/10/21

L'offuscamento riesce a far sì che **il tool di reversing ritorna dei risultati non precisi**, e rende quindi l'analisi più difficile. **Siamo tanto più sicuri quanto l'analisi è precisa.**

Quindi mi basta abbassare la precisione dell'analisi per migliorare la sicurezza. L'idea è quella di **mitigare**: se c'è una violazione, vogliamo identificarla e aggiostarla di modo che l'investimento dell'attaccante non sia worth.



Offuscamento del codice

È una trasformazione dei programmi che **preserva il comportamento osservazionale del codice**, possibilmente anche nelle prestazioni.

Il livello di sicurezza garantito da una tecnica di offuscamento dipende da:

- **Sofisticatezza** della trasformazione
- **Potere** del deoffuscatore
- Quantità di **risorse**, ovvero spazio e tempo, disponibili

Misurare questa potenza **non è banale**, e **l'offuscamento non è mai una protezione completa**.

Per valutare l'offuscamento, quindi, si usano 4 parametri:

- **Potenza**: quanta confusione viene aggiunta
- **Resilienza**: misura quanto è difficile per un attaccante deoffuscare
- **Costo**: quanto tempo extra è necessario
- **Stealthiness**: all'attaccante non deve essere evidente che è stato usato l'offuscatore!

$$\text{Qualità} = \text{Potenza} + \text{resilienza} + \text{costo} + \text{stealthness}$$

La prima versione di code obfuscation fu di un paper del 93 e quella di **differenziare il codice** per evitare di permettere di usare più volte On in più punti degli attacchi. → protection through obscurity. Si nota però che la diversificazione **non può risolvere il problema: al più rimanda l'attacco!** Vogliamo **rimandare abbastanza a lungo che l'attacco non risulta più vantaggioso**.

Le diversificazioni proposte elencate nel paper sono:

- **Sostituire istruzioni con sequenze di istruzioni equivalenti**
- **Riordinare le istruzioni** (o quelle commutative oppure inserendo dei jump per poi avere il path corretto in esecuzione)
- **Sostituire le variabili con una diversa rappresentazione** del valore che contengono (codifica) o crittandole
- **Istruzioni che non vengono eseguite o non influiscono** realmente sulla funzionalità.

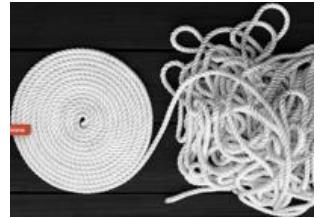
Di fatto sono, quindi, anche le **prime tecniche di offuscamento**. La definizione ufficiale arriva nel 98:

Def: Trasformazione offuscante

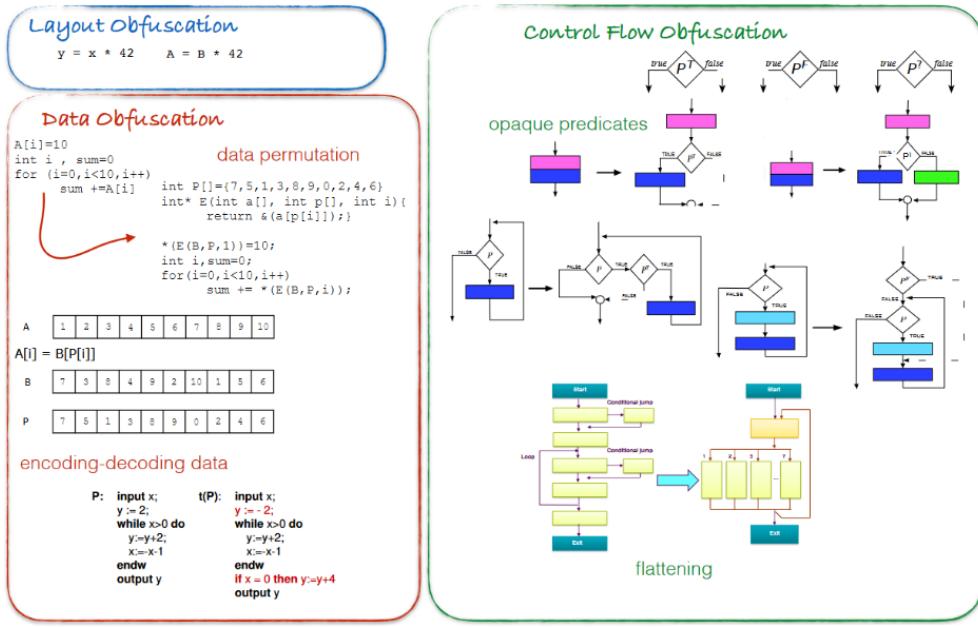
Definiamo $P \xrightarrow{T} P'$ la **trasformazione** di un programma P in un programma target P' .

$P \xrightarrow{T} P'$ è definita trasformazione offuscante se P e P' hanno lo **stesso comportamento osservabile** e soddisfa le condizioni:

- Se P non termina o termina con un errore, allora P' non termina.
- Altrimenti, P' deve terminare e produrre lo stesso comportamento di P .



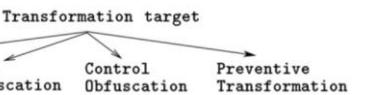
L'offuscation si divide in:



Bontà dell'offuscamento

La bontà di un offuscamento si misura via:

- **Potenza** → *quanta confusione aggiungo*
Grado di offuscamento aggiunto, quanto è più complicato da analizzare il programma trafrormato;
- **Resistenza** → *quanto è facile togliere confusione*
Misura quanto l'offuscamento è resistente, aka quanto è complicato annullarlo (aka ricostruire il programma originale), ad esempio i il renaming non è invertibile.
Esempio: true=false
 - Se metto $x=4$ è poco resistente essendo molto chiara
 - Puntatore==puntatore è molto resistente perché devo capire cosa c'è nelle celle puntate e capire che è sempre true/false
- **Costo** → *penalizzazione delle prestazioni in termini di tempo/spazio*
- **Stealthiness** → *quanto è nascosto il fatto che è avvenuto un offuscamento.*



Potenza

Def paper del 93: Potenza della trasformazione

Sia \mathcal{T} una trasformazione che mantiene il comportamento del programma, tale che $P \xrightarrow{\mathcal{T}} P'$ trasformi un programma sorgente P in un programma target P' . Sia $E(P)$ la complessità di P [come definita da una delle metriche nella Tabella 1].

$T_{pot}(P)$, la potenza di \mathcal{T} rispetto a un programma P , è la misura di **quanto \mathcal{T} cambia la complessità di P** . È definita come

$$T_{pot}(P) \stackrel{\text{def}}{=} E(P')/E(P) - 1.$$

Potenza in offuscamento

\mathcal{T} è una trasformazione potente in offuscamento se $\mathcal{T}_{pot}(P) > 0$.

Il punto debole di questa definizione è che **la funzione E di potenza può essere definita in millemila modi!**

Una possibile interpretazione è scegliere una delle metriche citate dal paper. Qualche esempio:

- **Trasformazioni sintattiche:** # parametri, # righe, # branching # rientri...
Diciamo che per ognuna di questi c'è una facile trasformazione che aumenta la complessità ma non la difficoltà a comprendere il programma. *Quindi non cattura molto bene*
- **Come in crittografia:** Un'altra definizione interessante possibile è presa in prestito dalla crittografia vede come trasformazione offuscante una trasformazione tale per cui dal programma trasformato posso imparare esattamente tanto quanto dal programma originale, un po' tipo "black box". Questa definizione permette di arrivare a poter dire che un vero offuscamento è impossibile. Rilassando questa definizione, in realtà, si arriva ad avere un offuscamento "fattibile"; tuttavia ha una complessità così alta che de facto non è fattibile.
- **Confronto fra risultati di analisi statica** (made in Univr) misuro rispetto all'analizzatore, e la trasformazione è potente se perdo informazioni nell'analisi. Questa funzione è rigorosa e funziona, ma non cattura tutto; *cattura solo i tool*, non il processo dinamico.
- **Test su programmatore**

Insomma, tutto questo non funziona molto perché **non si capisce come poter modellare in modo rigoroso l'attaccante** in caso di analisi empirica/dinamica.

Resilienza/resistenza

La resistenza, quando si è cercato di formalizzare una misura, si è deciso di considerare:

- **Sforzo del programmatore:** tempo richiesto per costruire un disoffuscatore automatico in grado di ridurre la potenza di \mathcal{T}
- **Sforzo del disoffuscatore:** tempo e spazio di esecuzione richiesto da questo deoffuscatore automatico affinché effettivamente riduca la potenza di \mathcal{T} .

Prese queste due misure, posso creare delle categorie che definiscono la resistenza:

$$\text{trivial} < \text{weak} < \text{strong} < \text{full} < \text{one-way}$$

Def: Resilienza

Da una trasformazione \mathcal{T} che mantiene il comportamento del programma, tale che $P \xrightarrow{\mathcal{T}} P'$ trasforma un programma sorgente P in un programma target P' . $\mathcal{T}_{res}(P)$ è detta **resilienza** di \mathcal{T} rispetto al programma P .

$$\mathcal{T}_{res} = \text{Resilienza}(\mathcal{T}_{deoffuscatore}, \mathcal{T}_{sforzoDelProgrammatore})$$

$\mathcal{T}_{res}(P)$ è detta **one-way** se le informazioni rimosse da P non sono ricostruibili da P' .

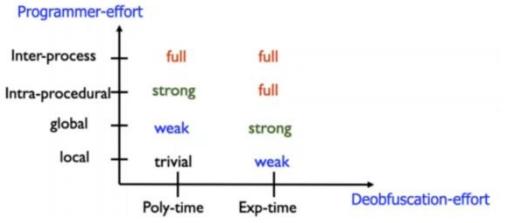
Costo

Il costo a seconda dell'aumento in tempo richiesto è classificabile.

Stealthiness

Possiamo tradurla con "**invisibilità**"; misura quanto il codice aggiunto dall'offuscamento "si nasconde" nel codice originale, ovvero **quanto è simile** – da alcuni degli studi fatti, per l'attaccante umano *localizzare il meccanismo di protezioni è il primo passo per smontarlo*.

METRIC	METRIC NAME	CITATION
μ_1	Program Length	Halstead [8]
	$E(P)$ increases with the number of operators and operands in P .	
μ_2	Cyclomatic Complexity	McCabe [20]
	$E(P)$ increases with the number of predicates in F .	
μ_3	Nesting Complexity	Harrison [9]
	$E(F)$ increases with the nesting level of conditionals in F	Tabella 1
μ_4	Data Flow Complexity	Oviedo [23]
	$E(F)$ increases with the number of inter-basic block varia	
μ_5	Fan-in/out Complexity	Henry [10]
	$E(F)$ increases with the number of formal parameters to F , and with the num data structures read or updated by F .	
μ_6	Data Structure Complexity	Munson [21]
	$E(P)$ increases with the complexity of the static data structures declared in P . ity of a scalar variable is constant. The complexity of an array increases with dimensions and with the complexity of the element type. The complexity increases with the number and complexity of its fields.	
μ_7	OO Metric	Chidamber [3]
	$E(C)$ increases with (μ_7^C) the number of methods in C , (μ_7^b) the depth (disto root) of C in the inheritance tree, (μ_7^d) the number of direct subclasses of C , (μ_7^o) of other classes to which C is coupled*, (μ_7^r) the number of methods that can b response to a message sent to an object of C , (μ_7^t) the degree to which C 's me reference the same set of instance variables. Note: μ_7^t measures cohesion; i.e. related the elements of a module are.	



Es. è più stealth controllare hash == 0 (con sezioni che mettono lo stealth a 0) piuttosto che hash = 34567869879. Per far andare a 0 questo hash, la tecnica tipica è quella di aggiungere un “corrector block” che forza l'hash ad andare a 0.

È **context/programma-sensitive**: una trasformazione che in un programma è stealthy, in altri potrebbe non esserlo.

Classificazione dell'attaccante

- Pattern-matching più veloce e facile
- Analisi statica
- Analisi dinamica
- Human-assisted

Classificazione del goal dell'analisi

- Trovare dove sono dati
- Trovare implementazione di certe funzionalità
- Estrarre frammenti di codice funzionante
- Capire il programma

Obiettivi delle tecniche di offuscamento

Gli obiettivi sono:

- Capire il target della protezione
- Chi è l'attaccante
- Definire la tecnica di offuscamento
- Provare la qualità dell'offuscamento.

5A - LAYOUT OBFUSCATION

È una delle più semplici: **tolgo commenti e rinomino le variabili con nomi che non hanno senso**; o tutti nomi randomici, nomi ripetuti il più possibile per confondere l'attaccante. È usata da numerosi offuscatori commerciali.

Per applicarla, deve essere fatta una “preanalisi” (es. per capire dove posso mettere nomi ripetuti e dove no.)

Sono applicate ovunque ☺

- **One way**
- **Potenza bassa:** *non ingannano i tool di analisi*: rimane tutto invariato..
- **Costo 0**

In realtà, questo tipo di offuscamento è molto potente per l'**authorship**, che è la disciplina che cerca di attribuire il codice a chi lo ha scritto. Per farlo, esistono degli algoritmi che cercano di estrarre dal codice delle caratteristiche stilometriche (es. come metto le parentesi, lunghezza media delle variabili, etc.). Funzionano molto bene su linguaggi ad alto livello, e sembra che vengano mantenuti anche sul compilato. Le trasformazioni di layout ingannano queste features. L'autorship è utile per esempio per capire chi ha scritto codice malevolo, o anche se si vuole nascondere di aver scritto qualcosa.

5B - DATA OBFUSCATION

I dati sono uno degli aspetti fondamentali in un programma; l'analisi di come i dati sono modificati è molto importante. Modifica il modo in cui i dati sono memorizzati, rendendo più complicato:

- Svelare il contenuto del dato reale
- Modificare la rappresentazione del dato impedendone l'analisi reale

Gli algoritmi di data obfuscation si basano su due funzioni, una di encoding e una di decoding.



$$\text{encode } E: T \rightarrow T' \quad \text{decode } D: T' \rightarrow T$$

Dato che i programmi sono manipolati da operazioni, servirà definire la sua controparte che lavora sul dato offuscato.

$$\text{op: } T \times T \rightarrow T \quad \text{op': } T' \times T' \rightarrow T'$$

Valutazione

Non sono semplici come quelle di layout:

- **Analisi preliminare:** devo fare delle analisi su range, potrei avere parametri che non so come nascondere.. insomma, tutte cose in più di cui occuparmi.
- **Parametri:** ho il problema di come nascondere i parametri di offuscamento (costanti opache)
- **Offuscamento omomorfico:** è desiderabile definire queste operazioni per operare in modo omomorfo, ma non è semplice (o possibile) su operazioni non semplice.
- **Analisi dinamica:** in alcuni punti del codice i dati dovranno, presumibilmente, essere comunque messi in chiaro; quindi bisogna prevenire l'analisi dinamica in altri modi.

La cosa buona è che complicano l'analisi statica.

Tecniche

Integers

Problemi delle codifiche:

- **Cambia il range** della variabili → controllo di *non andare in overflow*
- **Non posso combinare variabili codificate diversamente.** → *program slicing* per essere sicuri che in uno slice di un programma siano usate le stesse codifiche.

XOR masking

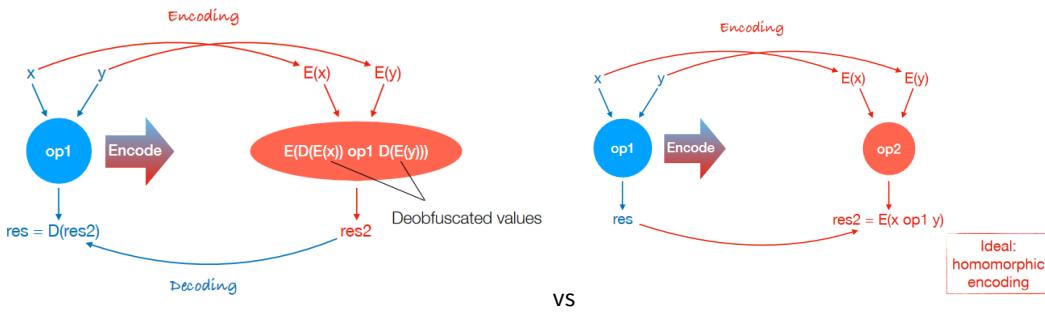
Una trasformazione banale è quella dello **XOR masking**: prende l'encoding di x con un certo parametro p. La funzione di decoding coincide con quella di encoding.

```
int a = 5;
int b = 8;
int x = a+b;           ⇒   int a = 9; // 9 = 5^12
...                   ...   int b = 4; // 4 = 8^12
printf("%d\n",x);    ...   int x = ((a^12)+(b^12))^12;
...                   ...
...                   ...   printf("%d\n",x^12);
```

The parameter needs to be protected!!

Problemi:

- Bisognerebbe poter **nascondere il parametro**
- Si vorrebbe **operare direttamente sui valori codificati** (quindi non come sopra, che decifra-opera-cifra) per non far mai vedere i valori in chiaro.



→ La soluzione sarebbe l'**homomorphic encoding** (cesareee)!

Homomorphic encoding: residual number coding (N e p)

N è primo, p è un parametro; per decodificare basta fare il **modulo N**. Questa è omomorfa per addizione prodotto, ma non per le operazioni di confronto.

Posso generalizzarla usando N come il prodotto di tanti valori coprimi tra loro → **residue number coding TBD**

- ✓ Generalisation based on modular arithmetic
- ✓ m_1, m_2, \dots, m_u , pairwise co-prime
- ✓ $N = m_1 \times m_2 \times \dots \times m_u$
- ✓ x in $[0, N-1]$ is encoded using u components
- ✓ $e(x) = \langle [x]_{m_1}, [x]_{m_2}, \dots, [x]_{m_u} \rangle$
- ✓ This encoding is invertible thanks to the Chinese Remainder Theorem, given $\langle y_1, y_2, \dots, y_u \rangle$ we have an unique (in modulo) solution x to the following equations:
 - ✓ $x \equiv y_1 \pmod{m_1}$
 - ✓ $x \equiv y_2 \pmod{m_2}$
 - ✓ ...
 - ✓ $x \equiv y_u \pmod{m_u}$
- ✓ Homomorphic for addition and product.
- ✓ $E(x) + E(y) = N^*p_1 + x + N^*p_2 + y = N(p_1 + p_2) + (x + y) = E(x+y)$
- ✓ $E(x) * E(y) = (N^*p_1 + x) * (N^*p_2 + y) = N^*(N^*p_1 * p_2 + p_2^*x + p_1^*y) + x^*y = E(x*y)$
- ✓ Not homomorphic for comparison tests $<, >, \leq, \geq$
- ✓ *Analysis of the use of the data to protect*

Altra possibilità da teoria dei numeri

Per esempio, posso rappresentare un intero y come $(y + p * N)$, dove N è il prodotto di due numeri primi successivi e p è un numero random. Per deoffuscare, quindi, devo rimuovere $(p * N)$ e forse fare il modulo? Non ho capito bhp

RSA e crittografia

Potrei usare anche algoritmi di cifratura classica, tipo RSA e DES! Il problema è che ci sono **poche operazioni omomorfe** in crittografia. È un ramo di ricerca.

Booleani

I booleani sono facili da codificare dato che sono pochi! :) Posso **usare tanti valori per ciascun booleano**; ad esempio:

- Tutti gli interi divisibili per 2 sono true, tutti quelli per 3 (e non viceversa) il false.
 - Codifico un valore booleano con una coppia di valori booleani, uguali=true diversi=false. Poi dovrò definire le operazioni su queste. Elenco tutti i casi
- (Nell'immagine: *output codificato // in1cod && in2cod = in1 && in2 = out*)
- Più splitto in T/F in variabili, più mi aumenta il costo.
- La resilienza aumenta se la tabella delle lookup è calcolata in runtime anziché hardcodata.

```

TD and[] = { (TD){0,0}, // {0,0} && {0,0} = T && T = T
              (TD){1,0}, // {0,0} && {0,1} = T && F = F
              (TD){0,1}, // {0,0} && {1,0} = T && F = F
              (TD){1,1}, // {0,0} && {1,1} = T && T = T
              (TD){1,0}, // {0,1} && {0,0} = F && T = F
              (TD){1,1}, // {0,1} && {1,0} = F && T = F
              (TD){0,1}, // {0,1} && {0,1} = F && F = F
              (TD){0,1}, // {0,1} && {1,1} = F && T = F
              (TD){1,0}, // {1,0} && {0,0} = F && T = F
              (TD){1,0}, // {1,0} && {0,1} = F && F = F
              (TD){1,0}, // {1,0} && {1,0} = F && F = F
              (TD){0,1}, // {1,0} && {1,1} = F && T = F
              (TD){0,0}, // {1,1} && {0,0} = T && T = T
              (TD){1,0}, // {1,1} && {0,1} = T && F = F
              (TD){0,1}, // {1,1} && {1,0} = T && F = F
              (TD){1,1}, // {1,1} && {1,1} = T && T = T
}

Lookup table
generated at
obfuscation
time, if
generated at
runtime more
resilient against
static analysis

```

```

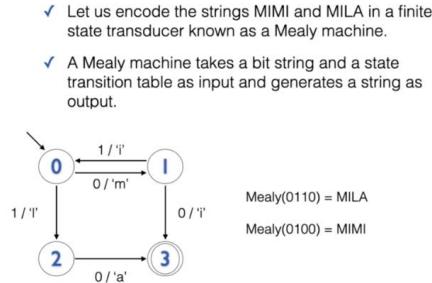
static String G (int n) {
    int i=0;
    int k;
    char S = new char[20];
    while (true) {
        L1: if (n==1) {S[i+1]='A'; k=0; goto L6};
        L2: if (n==2) {S[i+1]='B'; k=2; goto L6};
        L3: if (n==3) {S[i+1]='C'; k=3; goto L6};
        L4: if (n==4) {S[i+1]='D'; goto L9};
        L5: if (n==5) {S[i+1]='E'; goto L11};
        if (n>12) goto L11;
        L6: if ((k++)<<2) {S[i+1]='A'; goto L6};
        else goto L8;
        L8: if ((k++)<<2) valueOf(S);
        L9: if ((k++)<<2) 'C'; goto L10;
        L10: S[i+1]='B'; goto L12;
        L11: S[i+1]='E'; goto L12;
        L12: goto L10;
    }
}

```

Stringhe

Potrei avere delle stringhe semanticamente importanti (es. "inserisci la password", chiavi crittografiche). Quindi, per evitare il problema, potrei voler nascondere quelle stringhe.

Per esempio, posso fare che la stringa generata durante l'esecuzione anziché hardcodata: La stringa diventa una chiamata a funzione di questo tipo:



⇒ ↴

```

static String G (int n) {
    int i=0;
    int k;
    char[] S = new char[20];
    while (true) {
        L1: if (n==1) {S[i++]='A'; k=0; goto L6};
        L2: if (n==2) {S[i++]='B'; k=-2; goto L6};
        L3: if (n==3) {S[i++]='C'; goto L9};
        L4: if (n==4) {S[i++]='X'; goto L9};
        L5: if (n==5) {S[i++]='C'; goto L11};
        if (n>12) goto L1;
        L6: if (k++<2) {S[i++]=A'; goto L6};
        else goto L8;
        L8: return String.valueOf(S);
        L9: S[i++]='C'; goto L10;
        L10: S[i++]='B'; goto L8;
        L11: S[i++]='C'; goto L12;
        L12: goto L10;
    }
}
  
```

Aggregazione

Scalari

Posso unire le cose mettendo due variabili in metà cella a 64 bit. Ovviamente tutte le operazioni vanno sistamate di conseguenza.

Per aumentare la resistenza, ogni tanto posso aggiungere delle opreazione su tutta la variabile (quindi su entrambi i "veri" valori) nascondendo la semantica.

Array

Poco utilizzate, ma le menziona.

#include <stdio.h> int x [] = {1,2,3,4,5,6,7,8,9,10}; int main(int argc, char * argv[]) { int s = 0; int i; for (i=0; i<10; i++) { s += x[i]; } printf("sum=%d\n",s); }	#include <stdio.h> int x1[] = {1,3,5,7,9}; int x2[] = {2,4,6,8,10}; int main(int argc, char * argv[]) { int s = 0; int i; for (i=0; i<10; i++) { s += (i % 2 == 0?x1[i/2]:x2[i/2]); } printf("sum=%d\n",s); } ⇒
#include <stdio.h> int x1[] = {1,3,5,7,9}; int x2[] = {2,4,6,8,10}; int main(int argc, char * argv[]) { int s = 0; int i; for (i=0; i<5; i++) { s += x1[i]*x2[i]; } printf("cprod=%d\n",s); }	#include <stdio.h> int x[] = {1,2,3,4,5,6,7,8,9,10}; int main(int argc, char * argv[]) { int s = 0; int i; for (i=0; i<5; i++) { s += x[2*i]*x[2*i+1]; } printf("cprod=%d\n",s); } ⇒
#include <stdio.h> int x [] = {1,2,3,4,5,6,7,8,9,10}; int main(int argc, char * argv[]) { int s = 0; int i; for (i=0; i<10; i++) { s += x[i]; } printf("sum=%d\n",s); }	#include <stdio.h> int x[] [5] = { {1,2,3,4,5}, {6,7,8,9,10} }; int main(int argc, char * argv[]) { int s = 0; int i; for (i=0; i<10; i++) { s += x[i/5][i%5]; } printf("sum=%d\n",s); } ⇒
#include <stdio.h> int x[] [5] = { {1,2,3,4,5}, {6,7,8,9,10}, {11,12,13,14,15} }; int main(int argc, char * argv[]) { int s = 0; int i,j; for (i=0; i<3; i++) { for (j=0; j<5; j++) { s += x[i][j]; } } printf("sum=%d\n",s); }	#include <stdio.h> int x[] = { 1,2,3,4,5,6,7,8,9,10, 11,12,13,14,15}; int main(int argc, char * argv[]) { int s = 0; int i,j; for (i=0; i<3; i++) { for (j=0; j<5; j++) { s += x[i+5*j]; } } printf("sum=%d\n",s); } ⇒

Array splitting: Trasforma un array in 2 array; dovrò aggiornare l'utilizzo degli indici

Array merging: ne prendo due e li codifico in uno; devo aggiornare la procedura di accesso iordandomi dove ho messo cosa. Confondono l'analista perché rompono la logica dic ome sono usate le strutture dati.

Folding: array in una matrice, mappand relazioni diverse assieme. Again, poi bisognerà aggiustare l'accesso agli indici.

Flattening: prendo una matrice e la srotolo in un array.

```

#include <stdio.h>
char m [] = {
    't', 'r', 'g', 'e', ' ', 'm',
    'c', 's', 'e', 's', 'e', 'a' };
char p [] = {
    11, 3, 7, 1, 4, 0, 5, 6, 9, 10, 8,
    13, 2, 12 };

int main(int argc, char * argv[]) {
    int i;
    for (i=0; i<14; i++) {
        putchar(m[p[i]]);
    }
    putchar('\n');
}

```

Reordering: riordino in base a una permutazione degli indici; dovrò andare ad aggiornare gli indici di accesso agli array. Il risultato è che sembra io scorra l'array "a salti" mentre invece lo sto scorrendo in ordine sequenziale.

La **potenza** di queste trasformazioni viene dal fatto che **rompo l'aggregazione logica** tra dati.

Nascondere i parametri: costanti opache

Con alcuni encoding devo **nascondere dei parametri**, tipo il p per lo XOR o i numeri primi. Una soluzione possibile è quella delle costante opache.

Costante opaca

Una costante è **opaca** quando il suo valore è rimosso dal codice ed è **calcolato a tempo di esecuzione**, come **soluzione di un problema difficile** (NP), in modo tale che sia difficile da indovinare analizzando il programma offuscato.

! Ci stiamo proteggendo solo da un **analizzatore statico**.

Questo funziona perché chi scrive il programma sa che la risoluzione del problema sarà sempre (es.) 12, ma chi invece deve indovinarlo si trova a dover risolvere un problema difficile.

Per implementare correttamente le costanti opache, devono essere soddisfatte queste caratteristiche:

1. **Trasformazione sound:** il programma offuscato comporta come l'originale. Verificare la soundness deve essere facile.
2. Deve essere **difficile per un analizzatore statico indovinare la costante opaca** (=ho una variabile che sembra poter assumere tanti valori ma in realtà è costante)
3. **Costruire il valore nel momento dell'offuscamento non deve essere difficile**
4. **Calcolare il valore a runtime deve essere veloce**

Automatic generation of opaque contents based on the K-clique problem for resilient data obfuscation.
SANER 2017

Accenna all'idea: prendo il problema 3SAT: NPcompleto, calcolabile con la traccia

Voglio un assegnamento tale per cui una formula (clausola di 3 variabili) è soddisfatta. Essendo un and di or, va a farlo se almeno una è falsa. Il problema è poi tradotto nel trovare una K-clique, aka un sottografo con certe caratteristiche.

Es. Ci sono modi per costruire una formula che so non essere soddisfacibile. Quindi genero una forma 3sat che io so che NON è soddisfacibile, e faccio che in base al risultato il bit prende un certo valore. L'analizzatore statico, quindi, per sapere che il bit prende sempre quel valore dovrebbe scoprire se la formula è soddisfacibile o meno; questo è NP e quindi non ce la fa. Il trucco è proprio nel fatto che è difficile capire che NON è soddisfacibile, ma non il viceversa

```

1 int gm [] [51] = {
2     { 0, 0,...,1,1,0,1,1, },
3     ...
4     { ... }
5 };
6
7 int phi = TRUE;
8 int i, j;
9 int n = 17;
10 int s = 3*n;
11 int idx[n*sizeof(int)];
12
13 generate_subset (idx, n, s);
14
15 for (i=0; i<n-1; i++) {
16     for (j=i+1; j<n-1; j++) {
17         if (!gm[idx[i]][idx[j]]) {
18             phi=FALSE;
19             break;
20         }
21     }
22 }
23
24 if (!phi) {
25     opaque_bit = 0;
26 } else {
27     opaque_bit = 1;
28 }

```

- ✓ The function call at Line 13 randomly generates a set of vertexes.
- ✓ The nested loops at Line 15 and Line 16 verify whether the subgraph induced by the set of vertexes is a clique or not. If it is not, which is always the case by construction, the bit is set to the required value, 0 in this example (Line 18).

Efficacia

Protesting software through obfuscation: can it keep pace with progress in code analysis?
ACM Computing Survey 2016

Come valido le tecniche di offuscamento? Ovviamente dipende da quali tecniche e quali analisi/attaccanti.

Dipende tutto da:

Modellazione dell'attaccante	Modellazione dell'obiettivo dell'attaccante:
<ul style="list-style-type: none"> Pattern matching ✓✓: è la più semplice, e consiste nel ritrovare istruzioni in base a espressioni regolari o un classificatore fatto con machine learning. Analisi statica automatizzata ⚙⚙: ragionamenti automatici sulla semantica del codice fatti senza eseguirlo. Ad esempio, possiamo pensare a disassemblatori che interpretano quale branch potrebbe essere preso. Analisi dinamica automatica ⚙⚙: ragionamenti precisi sul comportamento del programma in base alle tracce osservate; ha il problema della code coverage Analisi human-assisted 🕹️💻: è la più potente, ed è il processo di reverse engineering che fa l'umano con l'aiuto di una varietà di strumenti. 	<ul style="list-style-type: none"> Trovare dati: ad esempio chiavi crittografiche, licensing keys, certificati, credenziali Trovare funzionalità: ad esempio trovar el'entry point di una funzione (tipo capire dove inizia un algoritmo crittografico, trovare i DRM...) Estrazione di codice: l'analista vuole estrarre un pezzo di codice da un codice offuscato (tipo, un algoritmo figo per poterlo riutilizzare) Capire il programma: capire interamente cosa fa il programma deoffuscato (deoffuscarlo, trovare vulnerabilità e rubare proprietà intellettuale)

Oltre a ciò, classificano anche i vari tipi di code rewriting in due categorie:

Static code rewriting	Dynamic code rewriting
<p>Un rewriter statico è una sorta di compilatore ottimizzante, che modifica il codice durante l'offuscazione ma l'output è eseguito direttamente senza ulteriori modifiche. Tutte le cose viste fino ad ora sono di questo tipo.</p> <p>Esempi: predici opachi, sostituzione di istruzioni, riordinamento, control flow obfuscation...</p>	<p>Il codice eseguito è diverso da quello che è visibile staticamente nell'eseguibile.</p> <p>Esempi: crittografia, modifiche dinamiche al codice, requirements di ambiente (tipo ch fuori dall'ambiente giusto non rivela i segreti), token hardware, virtualizzazione...</p>

Quello che fanno, quindi, è incrociare queste due assi e per ciascuna capire se vince l'attaccante o l'offuscamento.

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

Name	PM		Autom. Static			Autom. Dynamic			Human Assisted				
	LD	LC	LD	LC	EC	UC	LD	LC	EC	UC	LD	LC	EC
Data obfuscation													
Reordering data	✗			✓					✓				
Changing encodings	✓		✗						✓				
Converting static data to procedures	✓								✓		✓		

Legend	Black	obfuscation breaks analysis fundamentally
	Dark Gray	obfuscation is not unbreakable, but makes analysis more expensive
	Light Gray	obfuscation only results in minor increases of costs for analysis
	✓	A checkmark indicates that the rating is supported by results in the literature
		Scenarios without a checkmark were classified based on theoretical evaluation

5C - CODE/CONTROL OBFUSCATION

È a metà fra riscrittura statica e dinamica.

Riscrittura statica del codice

Appena nato, era l'inverso di un ottimizzatore: dato il codice tutte le trasformazioni erano fatte e il codice offuscato era dato in output ed eseguito in forma offuscata.

5. Rimpiazzamento di istruzioni
6. Predicati opachi
7. Inserimento di codice inutile o irrilevante
8. Riordinamento
9. Trasformazione dei loop
10. Ricombinazione e splitting di funzione
11. Name scrambling
12. Control flow obfuscation

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching, LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

Name	PM		Autom. Static				Autom. Dynamic				Human Assisted					
	LD	LC	LD	LC	EC	UC	LD	LC	EC	UC	LD	LC	EC	UC		
Static code rewriting																
Replacing instructions		✓			✓											
Opaque predicates					✓											
Inserting dead code							✓			✓	✓					
Inserting irrelevant code	✓															
Reordering																
Loop transformations																
Function splitting/recombination						✓										
Aliasing						✓				✓						
Control flow obfuscation	✓								✓	✓		✓		✓		
Parallelized code																
Name scrambling						✓										
Removing standard library calls														✓		
Breaking relations																
Legend							obfuscation breaks analysis fundamentally obfuscation is not unbreakable, but makes analysis more expensive obfuscation only results in minor increases of costs for analysis ✓ A checkmark indicates that the rating is supported by results in the literature Scenarios without a checkmark were classified based on theoretical evaluation									

Predicati opachi

Un predicato è un'espressione booleana che valuta a vero o a falso. Un predicato è opaco quando può assumere una sola configurazione (es. sempre vero o sempre falso), ma **il fatto che assume un solo valore è difficile da dedurre per l'attaccante.**



I **predicati unknown** (=che vanno sia a true che a false) prevedono di il terzo è l'unico che induce confusione anche in analisi dinamica. Consistono nel **mettere nei due branch due codici diversi ma implementano la stessa cosa**. più semplice è rompere il blocco, eventualmente mettendo un bogus block nel ramo che non eseguo.Ci si mettono legami che in realtà non esistono per confondere meglio la comprensione. Si possono anche usare all'interno di un ciclo per estendere la guardia.

Alcuni esempi di predicati opachi numerici, aka guardie "patocche", aka che in realtà valutano sempre allo stesso valore (penso vero), sono a lato.

Questo è fatto è provato dalla teoria dei numeri. Ci affidiamo al fatto che l'analizzatore non pensi a questi predicati, ma ovviamente sono pubblici quindi non sono così resistenti.

$$\begin{aligned}
 & \forall x, y \in \mathbb{Z}: 7y^2 - 1 \neq x^2 \\
 & \forall x \in \mathbb{Z}: 2 \mid (x + x^2) \\
 & \forall x \in \mathbb{Z}: 3 \mid (x^3 - x) \\
 \\
 & \forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: (x - y) \mid (x^n - y^n) \\
 & \forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: 2 \mid n \vee (x + y) \mid (x^n + y^n) \\
 & \forall n \in \mathbb{Z}^+, x, y \in \mathbb{Z}: 2 \nmid n \vee (x + y) \mid (x^n - y^n) \\
 & \forall x \in \mathbb{Z}^+: 9 \mid (10^x + 3 \cdot 4^{(x+1)} + 5) \\
 & \forall x \in \mathbb{Z}: 3 \mid (7x - 5) \Rightarrow 9 \mid (28x^2 - 13x - 5) \\
 & \forall x \in \mathbb{Z}: 5 \mid (2x - 1) \Rightarrow 25 \mid (14x^2 - 19x - 19) \\
 & \forall x, y, z \in \mathbb{Z}: (2 \mid x \wedge 2 \mid y) \Rightarrow x^2 + y^2 \neq z^2 \\
 & \forall x \in \mathbb{Z}^+: 14 \mid (3 \cdot 7^{x+2} + 5 \cdot 4^{2x-1} - 5)
 \end{aligned}$$

<p>Block splitting: inserisco il predicato opaco in mezzo a un blocco già esistente.</p>	
<p>Bogus block: blocco fasullo che uno potrebbe pensare essere eseguito ma in realtà non lo sarà mai. Agiungo molto rumore all'analisi statica.</p>	
<p>Unknown predicate: posso mettere due blocchi diversi ma semanticamente equivalenti. Diversifico un po' il codice, ma in realtà è la stessa cosa. Qui il problema è avere delle tecniche che diversifichino il codice in maniera sensata e automatica.</p>	
<p>Aumentare le condizioni di un loop: Posso mettere in and la guardia di un loop che già esiste con un altro predicato opaco, che magari ha dentro "false dipendenze" e complica l'analisi.</p>	
<p>Trasformare un grado da riducibile a irriducibile: È la cosa che complica più la vita, e questo accade quando ho una guardia (in un path che però non verrebbe mai effettivamente preso) che può saltare in mezzo di un loop. Complica la vita perché gli algoritmi di analisi assumono sempre il grafo di esecuzione sia riducibile, e sono efficienti solo sui riducibili.</p>	
<p>Nello specifico, per farlo tornare riducibile bisogna duplicare il codice ogni volta che c'è un jump in un ciclo.. quindi, dovendo duplicare il codice in giro, posso arrivare a far esplodere la dimensione.</p>	
<p>Per verificare se un grafo è riducibile si applicano due trasformazioni finché non arrivo ad avere un solo nodo: se ci riesco, è riducibile.</p>	
<p>T1: elimino un self loop Si può dimostrare che un grafo è irriducibile se contiene questa struttura:</p>	
<p>→ posso renderla riducibile duplicando →</p>	

Costruire predicati opachi

Tendenzialmente questo viene fatto usando:

- Risultati di **teoria di numeri** (aka proprietà che si sa essere sempre vere o false)
- Difficoltà di fare analisi degli alias
- Difficoltà di fare analisi in concorrenza.

Per proteggere i predicati opachi, comunque è importante sia che siano **difficili da rompere**, ma soprattutto che siano **difficili da beccare** (quindi, se prendo la prima tabella di predicati su internet, posso assumere che la abbia anche l'attaccante).

Calcolo dei puntatori (alias analysis)

Manufacturing cheap, resilient and stealthy opaque predicates, Collberg et al. POPL 1998

Oltre ai prediciati numerici posso introdurre prediciati opachi anche attraverso problemi che so essere difficili da risolvere staticamente. Per esempio, possiamo fare l'**analisi di alias (due puntatori puntano allo stesso posto?)**, che è un problema NP per puntatori consecutivi "profondi" più di due.

Gli algoritmi conosciuti funzionano particolarmente male quando ci sono le cosiddette **destructive updates**, ovvero quando i puntatori possono essere eliminati e modificati.

L'idea è quella di definire delle **liste** dove vengono inseriti **un puntatore per lista**. A questo punto si aggiungono/cancellano/etc dei nodi sulla lista, ma si mantiene l'**invariante** che q_1 è un puntatore di g_1 e q_2 di g_2 .

A questo punto, posso mettere come prediato opaco $q_1 == q_2$. In questo modo, io so che è sempre falso per definizione, ma per l'analisi di alias è molto difficile estrarre questa informazione.

Per complicare ulteriormente, posso farlo con *più* sottografi.

Analisi degli elementi di un array

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
36	58	1	46	23	5	16	65	2	41	2	7	1	37	0	11	16	2

Invariants:

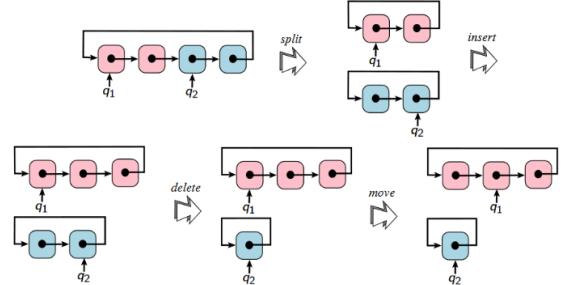
- ① every third cell (in pink), starting will cell 0, is $\equiv 1 \pmod{5}$;
- ② cells 2 and 5 (green) hold the values 1 and 5, respectively;
- ③ every third cell (in blue), starting will cell 1, is $\equiv 2 \pmod{7}$;
- ④ cells 8 and 11 (yellow) hold the values 2 and 7, respectively.

```
int g[] = {36, 58, 1, 46, 23, 5, 16, 65, 2, 41,
           2, 7, 1, 37, 0, 11, 16, 2, 21, 16};

if ((g[3] % g[5]) == g[2])
    printf("true!\n");

g[5] = (g[1]*g[4])%g[11] + g[6]%g[5];
g[14] = rand();
g[4] = rand()%g[11]+g[8];

int six = (g[4] + g[7] + g[10])%g[11];
int seven = six + g[3]%g[5];
int fortytwo = six * seven;
```



CreateOpaquePredicate(P)

1. Add to P code to build a set of dynamically allocated global point-structures $\mathbf{G} = \{G_1, G_2, \dots\}$
2. Add to P a set of pointers $\mathbf{Q} = \{q_1, q_2, \dots\}$ that point to the structures in \mathbf{G}
3. Construct a set of invariants $I = \{I_1, I_2, \dots\}$ over \mathbf{G} and \mathbf{Q} such that
 - $(q_i = !q_j)^T$ if q_i and q_j are known to point to different graphs G_n and G_m
 - $(q_i = !q_j)^?*$ if q_i and q_j are known to both point into a graph G_k
4. Add code to P that occasionally modifies the graphs in \mathbf{G} and the points in \mathbf{Q} while maintaining invariants I
5. Using the invariants I , construct opaque predicates over \mathbf{Q} such that
 - q_i always points to nodes in G_i ,
 - G_i is always strongly connected

Costruisco un array, e stabilisco che l'array ha delle proprietà invarianti, e posso aggiornare l'array mantenendo quelle invarianti.

Sfruttando queste proprietà che conosco, posso fare predicatori opachi. Per esempio:

- **Rosa:** sempre vera per le proprietà che sappiamo!
- **Blu:** modifico sempre i valori ma sto mantenendo le invarianti! Per l'attaccante è difficile capire da questo le proprietà.

Spaccare un prediato opaco

Posso:

1. Trovare la soluzione
2. Trovare gli input
3. Trovare il range di valori
4. Computare il risultato di f per ogni possibile valore e uccidere il branch se è sempre T/F
5. Usare teoria dei numeri o un attacco bruteforce per determinare l'opacità

Insomma, con questi posso creare spazio/rumore ma sto tranquillo che non sto spacciando la semantica. Dall'analisi recente, sembra che i predicatori opachi sono davvero la chiave per spaccare l'analisi statica.

Predicati opachi dinamici

Experience with software watermarking, Palsberg et al, ACSAC 2000

Nascono per cercare di togliere il fatto che i predicati opachi sono **deboli all'analisi dinamica**.

Sono predicati la cui idea è che l'opacità venga **divisa su insiemi di predicati**; quello che voglio sapere non è il valore del singolo valore, ma di un **insieme di predicati opachi correlati** – che singolarmente possono avere valore diverso per ogni run, ma come **gruppo in una singola esecuzione vanno sempre tutti alla stessa cosa**. Diventa molto difficile capire quali predicati siano correlati!

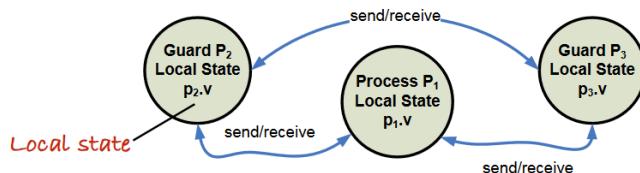
Predicate	Run 1	Run 2	Run 3	Run 4	Run 5
Pred 1	T	T	F	T	F
Pred 2	T	T	F	T	F
Pred 3	T	T	F	T	F
Pred 4	T	T	F	T	F
Pred 5	T	T	F	T	F

Predicati opachi distribuiti

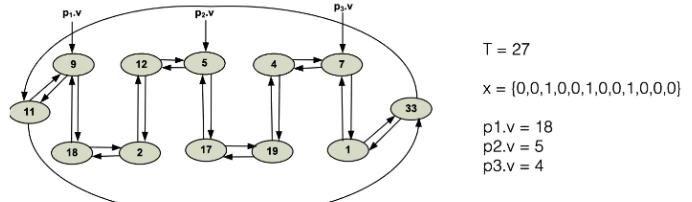
Manufacturing opaque predicates in distributed systems for code protection. Majundar and Thomborson 2006

Si estendono questi concetti a **programmi concorrenti**. Già solo il fatto di aver a che fare con un sistema concorrente – con le race conditions e tutto – ovviamente complica già parecchio la vita. L'idea è sempre quella delle invarianti: creo una **struttura dati globale** alla quale possono accedere un insieme di thread. **Il valore del predicato opaco dipende dagli stati locali di più thread**.

L'opacità di un predicato non è data da un processo, ma tra più processi.



- ✓ **0/1 Knapsack problem is NP-hard**: given a set of non-negative integers $S = \{a_1, a_2, \dots, a_n\}$ and a sum $T = \sum x_i * a_i$ where x_i ranges over {0,1}
- ✓ Consider for example the set $S = \{11, 9, 18, 2, 12, 5, 17, 19, 4, 7, 1, 33\}$
- ✓ P_1, P_2 and P_3 are initialized by passing a dynamic data structure, such as a doubly circular linked-list, initialized with the element of the set S .
- ✓ Every list has also a pointer for each process representing the process local state used in the evaluation of the opaque predicate.



Poi ci si rifa al problema dello zaino (così è difficile da risolvere): ci si chiede se la somma di x numeri fa un certo valore.

Control flow flattening

Protection of Software-based Survivability Mechanisms. Chenxi Wang et al 2001

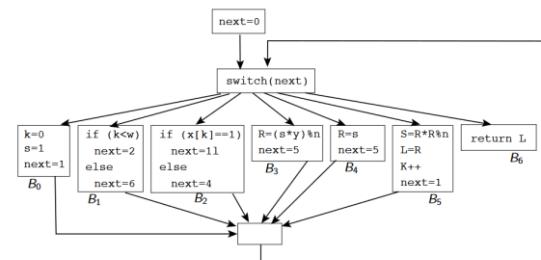


È la massima confusione possibile. È detto anche **chenxification**, dal nome dell'inventrice [Chenxi Wang](#). Ciascun blocco potrebbe essere il successivo di un altro: ho **uno switch che mi dice quale sarà il prossimo**.

```

int modexp(int y,int x[],
          int w,int n) {
    int R, L;
    int k = 0;
    int s = 1;
    while (k < w) {
        if (x[k] == 1)
            R = (s*y) % n;
        else
            R = s;
        s = R*R % n;
        L = R;
        k++;
    }
    return L;
}
  
```

chenxify



- Supermeglio efficace ad offuscare
- Supermeglio dispendioso – 10x del costo! La si applica a zone di codice veramente importanti.

Per fare recovery devo farmi una megaanalisi... Nsomma, megasemplice ma anche megaefficace 😊 Però è costosa;

Efficienza

Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching
LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

Name	PM		Autom. Static				Autom. Dynamic				Human Assisted													
	LD	LC	LD	LC	EC	UC	LD	LC	EC	UC	LD	LC	EC	UC										
Static code rewriting																								
Replacing instructions		✓			✓																			
Opaque predicates					✓																			
Inserting dead code							✓				✓	✓												
Inserting irrelevant code	✓																							
Reordering																								
Loop transformations																								
Function splitting/recombination							✓																	
Aliasing							✓																	
Control flow obfuscation	✓										✓	✓	✓	✓										
Parallelized code																								
Name scrambling							✓																	
Removing standard library calls														✓										
Breaking relations																								
Legend		<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10px; height: 10px; background-color: black;"></td><td>obfuscation breaks analysis fundamentally</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: gray;"></td><td>obfuscation is not unbreakable, but makes analysis more expensive</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: lightgray;"></td><td>obfuscation only results in minor increases of costs for analysis</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: white; vertical-align: middle; text-align: center;">✓</td><td>A checkmark indicates that the rating is supported by results in the literature</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: white;"></td><td>Scenarios without a checkmark were classified based on theoretical evaluation</td></tr> </table>														obfuscation breaks analysis fundamentally		obfuscation is not unbreakable, but makes analysis more expensive		obfuscation only results in minor increases of costs for analysis	✓	A checkmark indicates that the rating is supported by results in the literature		Scenarios without a checkmark were classified based on theoretical evaluation
	obfuscation breaks analysis fundamentally																							
	obfuscation is not unbreakable, but makes analysis more expensive																							
	obfuscation only results in minor increases of costs for analysis																							
✓	A checkmark indicates that the rating is supported by results in the literature																							
	Scenarios without a checkmark were classified based on theoretical evaluation																							

Riscrittura dinamica del codice

Cosa vuol dire “confondere l’analisi dinamica”?

- Quello che l’analisi dinamica impara da un atraccia non è quello che succede su un’altra traccia

Non si capisce se cambiare il codice dinamico mette effettivamente tangere l’analisi dinamica, dato che io guardo la traccia. Insomma forse tange solo la statica... È un problema aperto, soprattutto capirlo in modo rigoroso.

1. Crittazione o packing
2. Modifiche dinamiche al codice
3. Requisiti di environment (chiavi costruite dall’environment: se non c’è l’environment giusto il programma non rivela i suoi segreti)
4. HW-assisted code obfuscation (HW-SW binfing by making the execution of the software dependant on some HW token; without the token, the analysis of the software will fail)
5. Virtualization

Efficienza

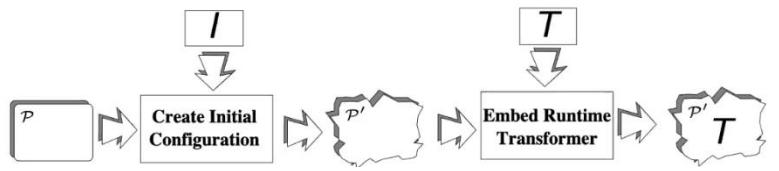
Protecting software through obfuscation: can it keep pace with progress in code analysis? ACM Computing Survey 2016

Table II. Analysis of the strength of code obfuscation classes in different analysis scenarios (PM = Pattern Matching
LD = Locating Data, LC = Locating Code, EC = Extracting Code, UC = Understanding Code).

Name	PM		Autom. Static				Autom. Dynamic				Human Assisted													
	LD	LC	LD	LC	EC	UC	LD	LC	EC	UC	LD	LC	EC	UC										
Dynamic code rewriting																								
Packing/Encryption		✓		✓					✓	✓			✓	✓										
Dynamic code modifications																								
Environmental requirements																								
Hardware-assisted code obfuscation											✓													
Virtualization			✓	✓					✓			✓		✓										
Anti-debugging techniques							✓		✓	✓				✓										
Legend		<table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 10px; height: 10px; background-color: black;"></td><td>obfuscation breaks analysis fundamentally</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: gray;"></td><td>obfuscation is not unbreakable, but makes analysis more expensive</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: lightgray;"></td><td>obfuscation only results in minor increases of costs for analysis</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: white; vertical-align: middle; text-align: center;">✓</td><td>A checkmark indicates that the rating is supported by results in the literature</td></tr> <tr> <td style="width: 10px; height: 10px; background-color: white;"></td><td>Scenarios without a checkmark were classified based on theoretical evaluation</td></tr> </table>														obfuscation breaks analysis fundamentally		obfuscation is not unbreakable, but makes analysis more expensive		obfuscation only results in minor increases of costs for analysis	✓	A checkmark indicates that the rating is supported by results in the literature		Scenarios without a checkmark were classified based on theoretical evaluation
	obfuscation breaks analysis fundamentally																							
	obfuscation is not unbreakable, but makes analysis more expensive																							
	obfuscation only results in minor increases of costs for analysis																							
✓	A checkmark indicates that the rating is supported by results in the literature																							
	Scenarios without a checkmark were classified based on theoretical evaluation																							

Si intendono offuscamenti che vengono fatti in due fasi:

- Prima fase in cui si **modifica il programma** di base
- Seconda fase in cui le modifiche fatte portano il programma in esecuzione a **modificarsi durante l'esecuzione**.



Si rompe, quindi, l'assunzione base degli analizzatori statici.

Idealmente, vorremmo che il programma continui a modificarsi diventando sempre diverso. Di fatto, però, prima o poi si ripeterà.

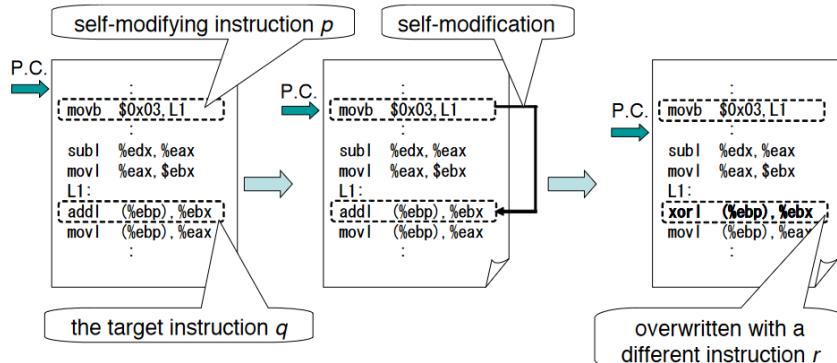


Ci possono essere offuscamenti che permettono al codice di automodificarsi. È nata in quanto l'analisi dinamica ha più successo sull'offuscamento standard. L'aggiungere confusione classico era aggiungere "rumore", cose che poi non succedono, per confondere l'analisi statica. Ma la dinamica non viene toccata, dato che guarda l'esecuzione! Quindi si pensa che per ingannare l'analisi dinamica forse ci vuole qualcosa di dinamico.

Meccanismo di self-modification

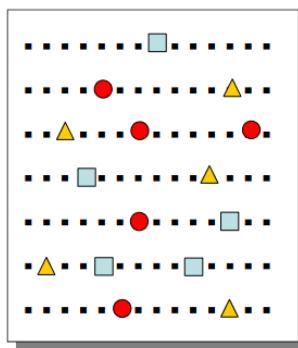
Exploiting self-modification mechanism for program protection COMPAC 2003

Il modo più semplice è selezionare alcune istruzioni e far sì che sovrascriva il codice stesso; ovvero, un'istruzione p del programma rimpiazza un'istruzione q nello stesso programma con un'istruzione r. Tutto a runtime.



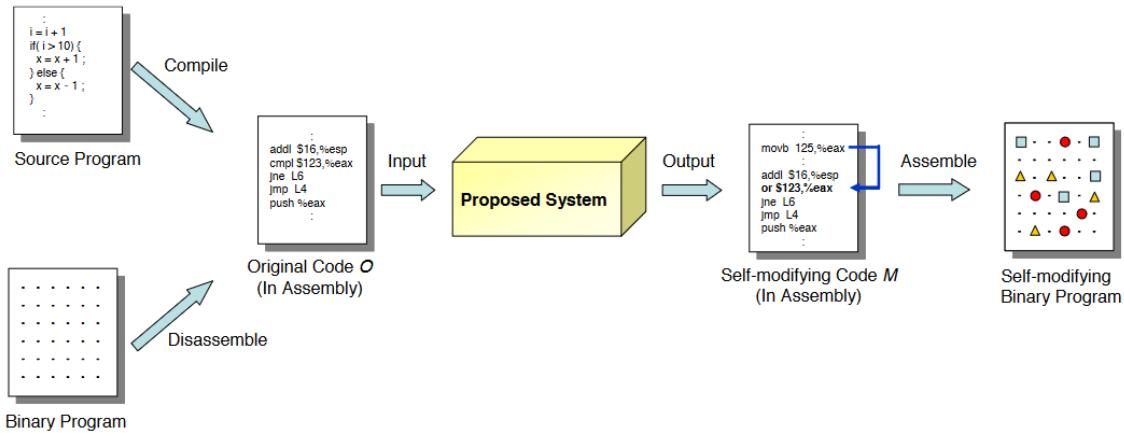
Quindi, per esempio, prendo delle istruzioni e le sostituisco con delle istruzioni dummy (config iniziale). In esecuzione, poco prima di arrivare ad eseguirle, le sostituisco con quelle vere. Poi le sovrascrivo di nuovo. Ho tre routine per ciascuna di queste cose.

- **Target instruction:** istruzione originale che nascondiamo usando il meccanismo di automodificazione
- **Hiding routine (HR):** insieme di istruzioni che nascondono un'istruzione target con un'istruzione dummy
- **Restoring routine (RR):** insieme di istruzioni che ristorano un'istruzione originale, nascosta da una dummy



- an instruction which is the target of camouflage
- a routine which writes the original instruction at run-time
- ▲ a routine which writes the dummy instruction at run-time

È essenziale che nel flowchart ho sempre che la RR sia sempre stata eseguita prima dell'esecuzione del target. Per modificare, agisco sull'opcode o sugli operandi dell'istruzione.



Le prestazioni sono terribili, perché per ogni istruzione ho almeno altre 2 operazioni. F.

Virtualizzazione

È anch'essa un tipo di offuscamento! L'offuscamento-virtualizzazione protegge un programma dall'analisi (sia automatica che manuale) **compilandolo in un bytecode per un'architettura virtuale random**, e ci attacca l'interprete associato. Di conseguenza, posso generare una versione equivalente ma protetta del programma originale usando **l'interprete come codice**, e salvando **il bytecode tradotto come dato** (che poi viene eseguito) nell'interprete stesso.

L'analisi **statica** diventa **totalmente inutile** su questi programmi, perché **solo il codice dell'interprete** (che è un grande switch e basta) è **visibile direttamente**, e il bytecode è un blocco dati non decomprimibile.

È usata molto dai **malware**: dato che l'architettura è randomizzata, interprete e bytecode cambiano molto da istanza a istanza, e questo rende molto **difficile riconoscere il malware**.

È la più potente: devo reversare l'interprete!!! Ma è anche molto costoso, quindi sono in pochi a usarla.

Algoritmo di Aucsmith

D. Aucsmith. Tamper resistant software: an implementation, patent 5892899, 1999



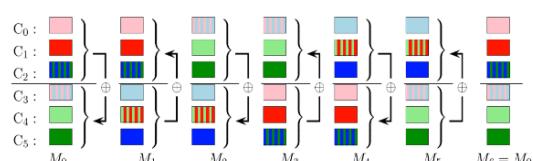
C ₀ :	[Red]
C ₁ :	[Blue]
C ₂ :	[Green]
C ₃ :	[Red]
C ₄ :	[Blue]
C ₅ :	[Green]

È stato ideato nel 1996 ed è il primo tentativo serio. L'idea è che continua a modificare il codice in modo che in ogni momento solo una parte del codice è in chiaro.

Il codice è diviso in 6 sezioni(cell), le quali sono divise fra upper (0-2) e lower (3-5).

Si procede a round. A ogni giro pari le celle sopra sono messe in XOR con quelle sotto (C₀ xor C₃, C₁ xor C₄, C₂ xor C₅), e a ogni giro dispari viene fatto lo stesso ma dal basso verso l'alto.

Siccome quando applico uno XOR due volte quello si annulla, avremo che ad ogni giro ho delle cose visibili e delle cose no. Tutto il giochino sta, per ogni turno, nel capre cosa è in stato "chiaro" e cosa no.



Quindi, bisogna capire cosa mettere inizialmente nelle celle per avere le cose in chiaro al momento giusto. (**ha fatto i calcolini degli xor inversi per capire cosa mettere dovema onestamente mi sparo scst guardateveli voi**)

Impedire il disassembly

Obfuscation of Executable Code to Improve Resistance to Static Disassembly. Linn and Debray. CCS 2003

(^ qui c'è la paper, no login required)

Le tecniche di offuscamento si concentravano sul confondere la fase di **decompilazione**/ricostruire il codice sorgente dall'assembly.

In realtà ci sono stati anche alcuni tentativi di confondere il disassembly.

Per confondere il disassembly, bisogna cercare di confondere gli algoritmi di disassembly; quello che bisogna fare è confondere il disassembler affinché pensi che dati (che non sarebbero istruzioni) siano istruzioni. Questo, quindi, provoca un errore di disassemblaggio.

L'idea è quella di andare ad inserire dei junk bytes, ovvero dei pezzettini di istruzioni che scombinano il disassembly delle istruzioni successive. Ovviamente devono essere inserite in blocchi che non verranno eseguiti!

Chiamiamo "candidate blocks" quei blocchi in cui posso mettere dei junk bytes, poiché non sono reachable durante l'esecuzione. Questo significa che il blocco precedente dovrà finire con un jump o con una chiamata a funzione!

Ci sono 2 algoritmi "classici". Semplificando:

Linear sweep

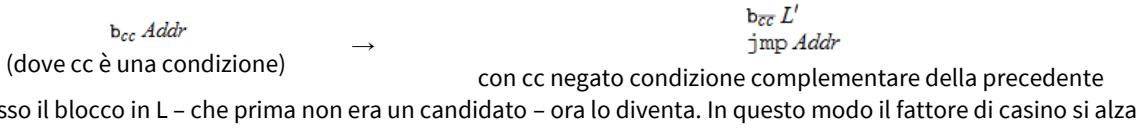
Il disassembler sa l'inidirizzo iniziale e finale della zona dove è contenuto il codice; una alla volta, disasembra istruzione per istruzione tutto ciò che trova. In questo modo, se metto dei dati brutti là in mezzo essi verranno trattati come istruzioni e sminchiano le cose.

Debolezza: inserisco dei jump bytes o istruzioni fasulle o pezzi di istruzione che disallineano il lavoro del disassemblatore.

Di default, riusciamo ad avere un 15% delle istruzioni disassemblate non correttamente, perché possiamo mettere junk solo nei blocchi non eseguiti, e che quindi sono preceduti da un salto "obbligatorio".

Branch flipping

Con compilatori ottimizzati, i blocchi papabili hanno una distanza di circa 30 istruzioni; purtroppo per noi questo non è sufficiente, perché i disassemblatori riescano ad autoaccorgersi dell'errore e ripararsi. Quindi, possiamo **aumentare il numero di candidati** attraverso una trasformazione detta **branch-flipping**. L'idea è di invertire il senso dei salti condizionali, traducendo ad esempio:



Call conversion

Infine, possiamo anche aggiungere la **call conversion**, che fa arrivare questa percentuale al 42%: *one solution to this problem is to reroute call instructions through a specialized branch function that branches to the intended target function via perfect hashing, as in the standard branch function, but then returns to some predetermined offset from the original call instruction (i.e., the offset to the real successor instruction that lies beyond some number of junk bytes). Using this method we are able to obscure control flow information by making function entry points more difficult to decipher while also increasing the potential to mislead the disassembler.*

Recursive traversal

È un po' più furbo, perché prende in considerazione il control flow.

Parte dall'istruzione iniziale, la disasembra; se quell'istruzione era di salto cerca di capire quali siano le possibili istruzioni successive e disassemblerà solo quelle, seguendo i flow possibili.

Debolezza: non è sempre facile trovare il successore staticamente; ci sarà dell'approssimazione.

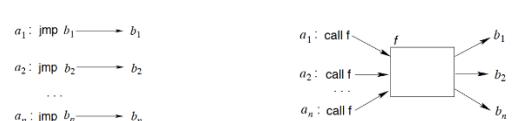
Quindi, per ingannarlo, si cerca di aumentare il numero di cambi del block. Questo perché



Branch functions

Per il secondo devo disturbare il calcolo del successore. Quindi introduco delle branch functions: una branch function è una unzione che, quando è chiamata da una locazione, sposta il controllo su una certa altra locazione corrispondente.

Assumo che dopo una jump ho una ret. Con la branch function trasformo i jump in chiamata a funzione con una lookup table. Questa f non ha la ret! Quindi confondo il calcolo dei successori.



Il diassembly viene spaccato col 75% di successo.

Confondere l'analisi dinamica

Code Obfuscation against Static and Dynamic Reverse Engineering HI 2011

L'idea di base è fare sì che capire una run di codice (via analisi dinamica) sia poi di fatto inutile per capire tutte le altre. Questo si chiama **diversificazione del CFG**.

L'idea è di **dividere il codice in gadget**, e ogni gadget finisce con un jump. Quindi **l'esecuzione è una serie di gadget**. Voglio diversificare i gadget in base all'input; il successore è calcolato in base ai gadget già eseguiti (quindi in base alla *storia + corrente*, non solo ciclo statico!!)

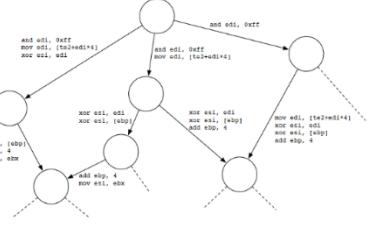
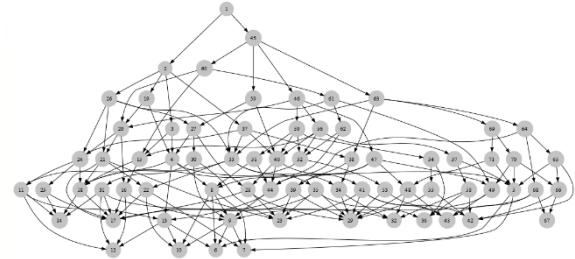
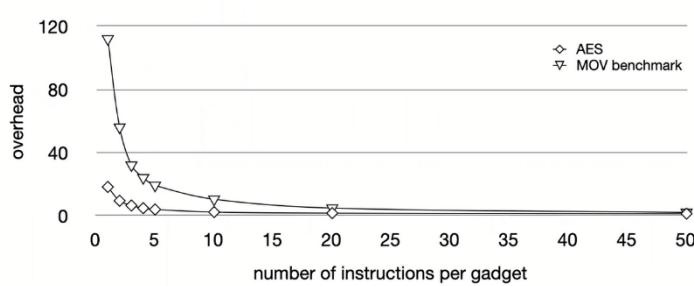


Fig. 2. Gadget graph.

Voglio che diversi input generino una sequenza di gadgets diversa. La situazione ideale sarebbe che per ogni input ho un'esecuzione diversa. Serve un metodo automatico per diversificare i gadget.

Più piccoli sono i gadget, più sono numerosi, più è efficace, più ho overhead. La forza è proporzionale al # di gadget.



Applicare l'offuscamento ai sistemi embedded

2018 IEEE Transactions on Computers

Inizia ad esserci del movimento anche in questo ambito.

Offuscamento in .NET

Ce lo dice perché ci ha fatto un'indagine. Ci sono un sacco di offuscatori per l'assembly di .NET, perché è il livello che ha senso proteggere: da qui si riesce a ricavare il codice sorgente.

	Obfuscator	Confuserex	Smart Assembly	Eazfuscator	.NetGuard	Spices.Net
Cost	free	free - disc	from 716\$	399\$	custom/jobs	399\$
Maintained						
Renaming	✓		✓	✓	✓	✓
Constant Encryption		✓	✓	✓	✓	✓
Resource Encryption	✓		✓	✓	✓	
Control Flow			✓	✓	✓	✓
Data Obf					✓	
Virtualization				✓		
Remove Useless			✓			✓
Anti Debugging/Profilers	✓				✓	
Anti-Decompilation	✓				✓	
Anti Disassembly	✓		✓		✓	✓
Anti memory dumping	✓				✓	
Assembly merging/embedding			✓	✓		
Debugging Support				✓	✓	
Tamper Detection	✓	✓			✓	
Deobfuscator	NoFuserEx	DeSmart-2007	De4dot - 2015		De4dot - 2015	
Deobfuscator			De4dot - 2015			
Summary	no	not maintained	very good	very good	very good obf as a service	very good recognized by microsoft

6 - L'IMPOSSIBILITÀ DELL'OFFUSCAMENTO

Parte da uno studio del 2001, che pone un quesito poi risolto nel 2012. È, alla fine, in linea col teorema di Rice.

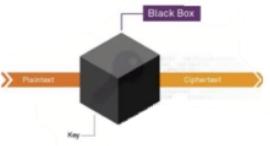
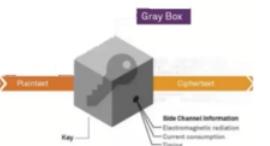
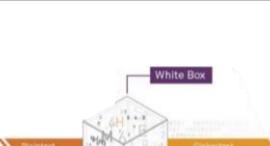
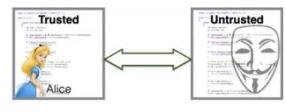
Questo lavoro non nasce dalla comunità di programming languages, ma da quella di **crittografia**.

Il concetto che deve rimanere è il seguente: **offuscare è impossibile**, così come analizzare è impossibile: perché il teorema di Rice ci dice che le uniche proprietà analizzabili dei programmi sono top e bottom (= “è un programma”). Questo ovviamente non ha impedito lo sviluppo di un sacco di sviluppatori, perché **è un risultato solo generale!** Il fatto che sia impossibile *in generale* pone domande più interessanti, tipo: quando è possibile? Che cosa è possibile? Sotto quali condizioni? Per quali classi di programmi?

Quindi, NB: i risultati “negativi” come l’impossibilità generale non sono risultati limitanti ;)

Tipi di sicurezza

Borrowiamo alcuni concetti dalla crittografia.

	<p><i>Black-box security</i> L’attaccante può osservare solo quello che entra e quello che esce.</p>
	<p><i>Gray-box security</i> L’attaccante vede qualcosa; ad esempio, può fare analisi dei consumi, sa il tempo che ci mette a cifrare, etc. In ambito grey box sono già possibili molti più attacchi che in black box.</p>
	<p><i>White-box security</i> L’attaccante vede tutto. È lo scenario in cui siamo noi! I DRM, per esempio, sono tutti programmi che cercano di far sì che il nostro scenario non sia più white box. Nello specifico, noi siamo in uno scenario MATE (man at the end): il programma è trusted, ma l’utente (e l’ambiente in generale) no. Quindi, per proteggere il codice, lo rendiamo black box.</p> <p style="text-align: right;"><i>Man At The End</i></p> 

Diciamo che quando le chiavi crittografiche si trovano all’interno del codice, l’attaccante può esser molto motivato a cercare di attaccarle; il problema si ha quando quelle chiavi vengono usate, perché diventano inevitabilmente **visibili**.

Lo scopo della **white-box crypt** è scrivere algoritmi di crittografia tali per cui **capire il processo di decrypt che permetterebbe l’accesso white-box è complicato**. Quindi, l’algoritmo è molto complicato e c’è **tanto rumore**, perché il calcolo non è nascosto *in quanto non lo vedo*, ma è nascosto *in quanto perso in mezzo ad altri millemila calcoli*.

Quindi, idealmente, vorremmo scrivere un codice che in uno scenario white-box alla fine **simula uno scenario black-box**.

Definizioni

Offuscamento (in questo contesto)

In questo contesto, un offuscatore è un programma che dato un programma P ritorna un programma equivalente O(P), ma O(P) è inanalizzabile.

È stato dimostrato che è possibile offuscare una specifica classe di funzioni, dette **point functions**.

Point functions

È una funzione $I_x(W) = \{1 \text{ if } w = x, 0 \text{ altrimenti}\}$

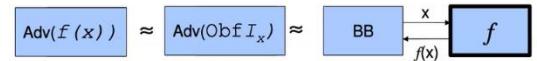
È dimostrato che queste funzioni possono essere offuscate attraverso una **one-way function**.

One-way functions

Sono funzioni semplici da calcolare ma impossibili da invertire.

L'offuscamento di una funzione di questo tipo funzionerebbe nel seguente modo. Prendiamo una funzione $y = f(x)$.

$$Obf\ I_x(w) = \{if\ y = f(w)\ return\ 1\ else\ return\ 0\}$$



f non è invertibile, quindi non mi dà nessuna informazione! I_x è come avere accesso blackbox a f .

Quindi sì, riusciamo ad offuscare, ma **è un caso molto specifico!** È possibile costruire un offuscatore che funzioni su ogni classe di funzioni partendo da questo risultato? La risposta non c'è, è domanda aperta.

Impossibilità di offuscare

On the impossibility of code obfuscation. Barak et al JACM 2012

Nel 2012/2001, hanno dimostrato che l'offuscamento è impossibile.

Definizioni preliminari

Virtual black-box property

Qualunque cosa che può essere scoperta attraverso la forma offuscata poteva essere osservata osservando il comportamento input-output del programma originale.

La dimostrazione, ovviamente, sarà il costruire una classe di funzioni particolari che spaccano tutto.

Analizzatori

Definiamo gli attaccanti come programmati che hanno una certa probabilità di prendere un cammino di esecuzione. Ogni scelta è al 50%. Ci limitiamo alle macchine di turing, quindi avremo delle stringhe che vengono accettate o meno.

Offuscatore di una macchina di Turing (TM)

Definiamo un offuscatore O come un algoritmo che per ogni macchina di turing M che:

- **Funzionalità:** $O(M)$ descrive una macchina di turing che ha la stessa funzionalità di M .
- **Rallentamento polinomiale:** il tempo e lo spazio di $O(M)$ sono al più polinomialmente più grandi di M .
- **Virtual black-box:** avere accesso alla macchina offuscata equivale ad avere accesso oracolo (=solo input e output) alla macchina originale.

2-TM offuscatori

Definiamo un offuscatore di due macchine di turing come **un offuscatore per cui valgono le stesse cose di prima**, e in più la proprietà virtual black-box vale di modo che **l'analizzatore può analizzare due macchine anziché una sola**:

l'attaccante che ha accesso alle due macchine offuscate impara la stessa cosa di un attaccante che ha accesso oracolo alle due macchine offuscate.

Dimostrazione

Dato che la proprietà è vera per ogni M , si costruiscono una classe particolare di macchine di turing che disprovano il perogni.

Definiamo queste due macchine di turing particolari:

$$\alpha, \beta \in \{0, 1\}^k \quad \text{Secret}$$

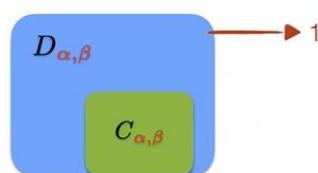
$$C_{\alpha, \beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0 & \text{otherwise} \end{cases}$$

$$D_{\alpha, \beta}(C) = \begin{cases} 1 & \text{if } C(\alpha) = \beta \\ 0 & \text{otherwise} \end{cases}$$

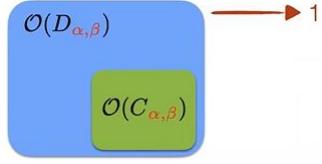
Distinguish if C computes $C_{\alpha, \beta}$
from $C_{\alpha', \beta}$ for any
 $(\alpha, \beta) \neq (\alpha', \beta')$
NON COMPUTABLE!

Simply compute $C(\alpha)$ for
poly(k) steps and check!

- C ritorna beta solo se gli do il segreto alpha, else torna 0.
- D prende in input la macchina di turing C , e se l'input dato a $C(\alpha) = \beta$ allora torna uno, altrimenti torna 0.



Ne genero le versioni offuscate e le chiamo $\mathcal{O}(C)$ e $\mathcal{O}(D)$, che per definizione mantengono la semantica.



L'accesso oracolo fa fatica a imparare il comportamento: **vedrà che è molto simile a una macchina che dà sempre 0**. La chiave è tutta qui: se io ho il codice, allora capisco facilmente che dà 1 quando è α , ma se invece devo andare a tentativi praticamente avrò una macchina sempre 0!

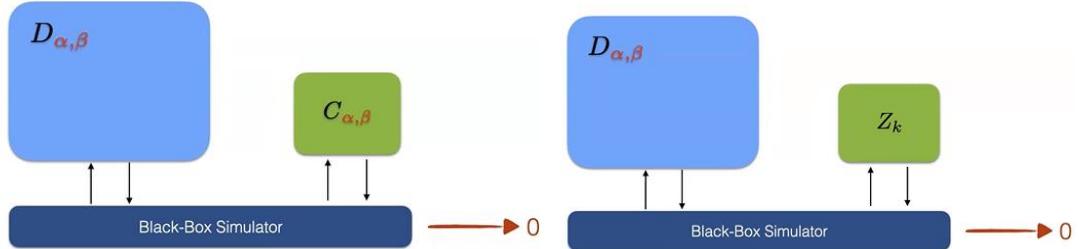
“Cosa vuol dire questo? No perché devo ricordarmi anche io!” cit. :’

Quindi, se ho questa macchina di Turing e il codice è molto semplice capire il comportamento. Ma se devo capirlo a oracolo, la vedo praticamente uguale a una macchina che dà sempre 0!

L'attaccante A andrà a eseguire l'offuscamento della seconda sull'offuscamento della prima, per vedere se fa 1:

$$\Pr[A(\mathcal{O}(C_{\alpha,\beta}), \mathcal{O}(D_{\alpha,\beta})) = 1] = 1$$

- Qual è la probabilità che un attaccante, eseguendo un offuscamento della prima nell'offuscamento della seconda, ottenga 1? 100%, perché l'offuscamento preserva la semantica! Per quanto io possa offuscarla, la semantica è preservata quindi non scappo da questa cosa.
- Qual è la probabilità che avendo accesso oracolo a queste due macchine si ottenga 1? Pochissimo! Presa una macchina Z che dà sempre 0, non è così diversa da C o D .



Quindi la virtual black-box non funziona: avere accesso a $\mathcal{O}(C)$ e $\mathcal{O}(D)$ dovrebbe essere come avere accesso a oracolo a C e D , mentre è praticamente uguale alla macchina Z !

Consideriamo un avversario che esegue D su C . L'attaccante, usando le cose offuscate, riesce a vedere che l'output fa 1:

$$\Pr[A(\mathcal{O}(C_{\alpha,\beta}), \mathcal{O}(D_{\alpha,\beta})) = 1] = 1$$

Allo stesso tempo, l'attaccante che esegue D su Z vede più o meno sempre 0.

$$|\Pr[S^{C_{\alpha,\beta}, D_{\alpha,\beta}}(1^k) = 1] - \Pr[S^{Z_k, D_{\alpha,\beta}}(1^k) = 1]| \leq 2^{-\Omega(k)}$$

Ma questo è molto simile/uguale a quest'altra:

$$\text{By definition of } A: \Pr[A(\mathcal{O}(Z_k), \mathcal{O}(D_{\alpha,\beta})) = 1] = 2^{-k}$$

Questa similarità contraddice la virtual box property. Avere accesso al codice offuscato, quindi, è profondamente diverso che avere accesso input-output. **Per un attaccante, avere un pezzo di codice – per quanto offuscato – è diverso da avere accesso input output!**

Conseguenze del risultato dell'impossibilità

Questo risultato non ci fa arrendersi; è possibile *in generale*, ma posso comunque agire nello specifico.

La cosa interessante del paper è che offuscare è impossibile *con queste condizioni*, ma possiamo riflettere su come **rilassare le condizioni** dell'offuscatore.

Indistinguishability offuscator

Indebolisce la black-box property.

Se ho due circuiti che fanno la stessa cosa e li offusco, **guardando la versione offuscata non so dire se derivo da uno o dall'altro**. Se esistesse una “forma normale”, quindi, offuscherei passando da questa forma normale.

Questo offuscatore **rivela ciò che è inevitabile rivelare e nasconde altre caratteristiche**.

Garg et al. CRYPTO 2013

È utile perché è stato dimostrato che **si può realizzare, benché ingestibile come complessità**. Il concetto di essere indistinguibile è diverso dal concetto di poter imparare “tanto quanto! – c’è insomma l’idea che **qualcosa verrà inevitabilmente imparato, ma tutto il resto – che è specifico dell’implementazione – viene nascosto**.

7 - OFFUSCAMENTO AVANZATO: SEMANTICS-BASED OBFUSCATION

Lavoro svolto in Univr col Giaco ❤ nel dottorato della professoressa.

L8 – Semantic code obfuscation

Sono stati gli unici a dare un **framework formale** per comprendere questo fenomeno. 😊

Parallelismo:

Teorema di Rice: non è possibile decidere nessuna proprietà non banale di un programma.
 → In realtà, da lì in poi la program analysis è riuscita a creare un sacco di analisi (dato che è solo una impossibilità generale).

<vs.>

Offuscare un programma è impossibile

→ Sapere che offuscare un programma è impossibile non significa che non riesco a creare un offuscato che nasconde qualcosa! Non è la soluzione generale ma qualcosa posso fare lo stesso.

Modellazione dell'attaccante

Loro pensano bene di ipotizzare l'attaccante come **analizzatore statico** – aka codice che non viene eseguito. Per sua natura, vede come possibili più cose di quelle che accadranno nella realtà.

Esempio: aggiungere dipendenze false

Sarebbe un programmino che conta caratteri, parole e righe.

Per offuscarlo, **aggiungo delle dipendenze false fra le variabili**. Sto introducendo dei predici opachi:

- Se il numero di parole è <= del numero di caratteri è sempre vero!

In questo modo, **vedo più dipendenze del dovuto** perché posso mettere quello che voglio in un predicato opaco:

```
Original()
int c, nl = 0, nw = 0, nc = 0, in;
in = false;

while ((c = getchar()) != EOF){
    nc++;

    if(c == ' ' || c == '\n' || c == '\t') in = false;

    elseif(in == false) {in = true; nw++}

    if(c == '\n') nl++;

}
out(nl,nw,nc);}
```

c	nc	nw	nl
c	X	X	
nc	X		X
nw	X		X
nl	X		X

```
Obfuscated()
int c, nl = 0, nw = 0, nc = 0, in;
in = false;

while ((c = getchar()) != EOF){
    nc++;

    if(c == ' ' || c == '\n' || c == '\t') in = false;

    elseif(in == false) {in = true; nw++}

    if(c == '\n') {if(nw <= nc) nl++};

    if(nl > nc) nw = nc + nl;
}

out(nl,nw,nc);}
```

Always false

c	nc	nw	nl
c	X	X	
nc	X	X	X
nw	X	X	X
nl	X	X	X

Esempio: confusione sul segno della variabile

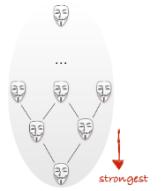
```
P: input x;
y := 2;
while x>0 do
    y:=y+2;
    x:= x-1
endw
output y
```



```
t(P): input x;
y := - 2;
while x>0 do
    y:=y+2;
    x:= x-1
endw
if x = 0 then y:=y+4
output y
```

- $\text{Sign}(\llbracket P \rrbracket) = \{\sigma \mid \exists n \geq 0. \sigma \in \langle 0+, \perp \rangle \rightarrow \langle 0+, + \rangle \rightarrow \langle +, + \rangle^n \rightarrow \langle 0, + \rangle\}$
- $\text{Sign}(\llbracket t(P) \rrbracket) = \left\{ \sigma \mid \begin{array}{l} \exists n \geq 0. \\ \sigma \in \langle 0+, \perp \rangle \rightarrow \langle 0+, - \rangle \rightarrow \langle +, 0+ \rangle^n \rightarrow \langle 0, + \rangle \end{array} \right\}$
- $\Im(\llbracket t(P) \rrbracket) = \Im(\llbracket P \rrbracket)$

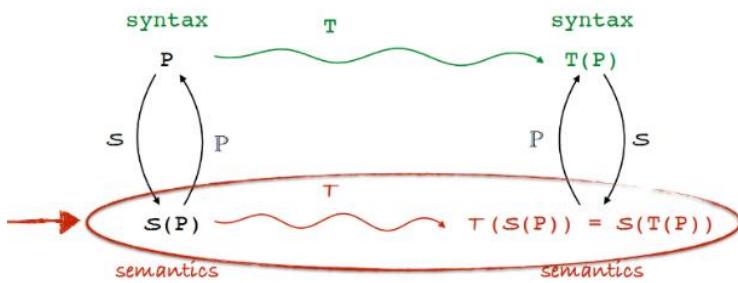
Se modelliamo l'attaccante come **osservatore di una proprietà del codice** (es. il segno delle variabili, o il range delle variabili) possiamo ordinare parzialmente (=il risultato è un reticolo) gli attaccanti in base alla **precisione della loro visione**.



Un offuscamento agisce sulla **sintassi**, ma l'attaccante vuole imparare cose della **semantica**. Ma se applico l'offuscamento, cosa succede alla semantica? Ovvero, **come proietto cambiamenti della sintassi sulla semantica?**

Questa relazione è definita dal programma di Cousot, che permette di passare dall'una all'altra senza troppo dolore; il main issue è che per ogni semantica ci sono infinite sintassi.

L'idea è che ogni trasformazione sintattica corrisponde a una trasformazione semantica. Al massimo posso avere delle trasformazioni semantiche di cui non ho l'equivalente sintattico, ma a noi interessa l'altro lato!



Definizione semantica della potenza

Attraverso la semantica, definiamo la potenza di un offuscamento:

Potenza di una trasformazione sintattica (POV: sei la semantica)

La trasformazione di un programma P è potente rispetto a un attaccante/analisi A e un programma P' quando:

$$A(S(P)) \neq A(S(T(P))) = A(T(S(P)))$$

Per esempio, la trasformazione con le false dipendenze è **moltissimo potente per l'analisi delle dipendenze**, perché l'analisi delle dipendenze sulla versione offuscata **ritorna un risultato diverso** rispetto all'analisi sul programma originale.

Il top sarebbe arrivare a top, cioè a non dare informazioni *at all*. Nel caso delle dipendenze, questo si otterrebbe avendo che nella versione offuscata del programma succede che tutto dipende da tutto.

Definizione semantica dell'offuscamento

È possibile caratterizzare il **comportamento offuscante** di ogni trasformazione come **la più concreta proprietà che viene conservata da quella trasformazione**.

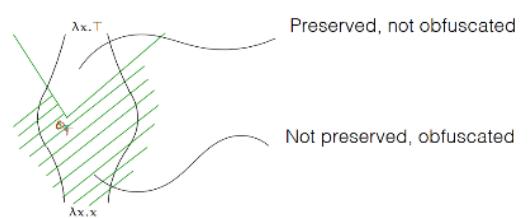
Ogni volta che trasformo un programma nascondo qualcosa: nascondo quello che non c'è più ;) Se faccio un rename, non conosco il nome che c'era prima, ad esempio.

La domanda quindi è: posso **caratterizzare l'azione offuscante** di una trasformazione?

La risposta è **sì**: se data una trasformazione riesco a **costruire la più concreta proprietà che preserva**. La più concreta/forte proprietà è quella che posso **vedere sia sul programma trasformato che su quello originale**.

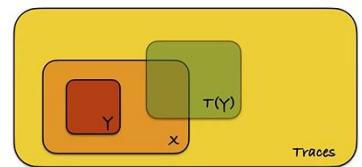
$$\begin{aligned} O_T &= \text{glb}\{O \text{ in uco (Semantics)} | \text{ for every } P: o(s(P)) \\ &= O(T(S(P)))\} \end{aligned}$$

In pratica, il goal è che riesco a definire una proprietà che fa da limite fra quello che viene nascosto e quello che non viene nascosto. È possibile con la constructive characterization.



Constructive characterization

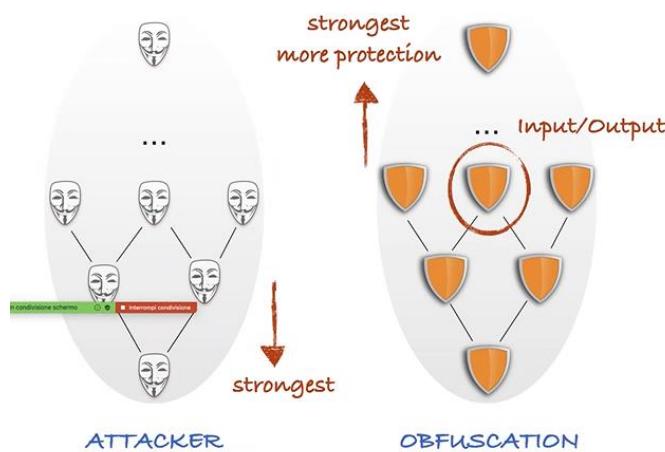
Data una trasformazione, è possibile costruire la più concreta proprietà che preserva. L'intuizione è che sarà una proprietà che ogni volta che io prendo un insieme di tracce e il loro offuscamento, deve vederle uguali.



Una proprietà alla fine è un insieme di tracce: **all'interno di questo insieme deve esserci sia una traccia che la sua offuscata.**

Questo significa che possiamo **ordinare le trasformazioni rispetto alla loro capacità offuscante**. Quindi, se ho che le trasformazioni possono essere ordinate in base alla più concreta proprietà che nasconde. Il gioco, quindi, è fatto:

- L'attaccante può essere **ordinato** in base a **quante tracce vede**
- Gli offuscamenti possono essere **ordinati** in base alle **proprietà preservate**



A questo punto, l'attaccante **vince se è più astratto della più concreta proprietà preservata** (=l'astrazione preserva quello che interessa all'attaccante); l'attaccante perde se sta sotto. Se non sono direttamente confrontabili ho delle operazioni sui domini che mi possono dire chi vince.



Qui sopra vedete anche un esempio di ordinamento parziale MA LASOM PERDER VA LA

Riassumendo: vogliamo interpretare l'offuscamento in maniera oggettiva/quantitativa della potenza, certificando con una proof che quella proprietà è stata protetta o meno. Con questo riesco a farlo, perché mi sono ricondotto per entrambi i casi allo stesso modo.

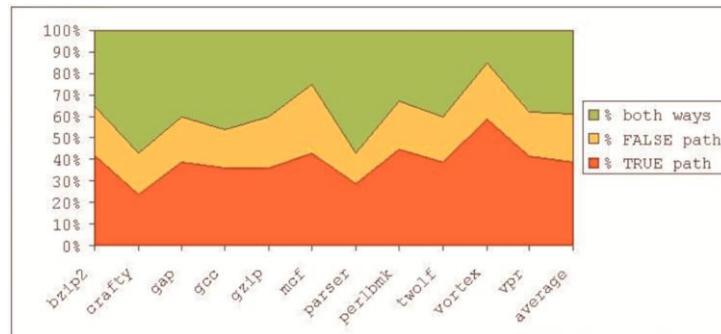
Uso di questa analisi: rompere i predicati opachi!

In generale, per esaminare un predicato e capire se è vero o falso devo:

- Fare analisi delle dipendenze
- Capisco tutte le istruzioni che concorrono al calcolo della guardia
- Provo per tutti i possibili valori coinvolti nel calcolo e poi capire se va sempre a vero.

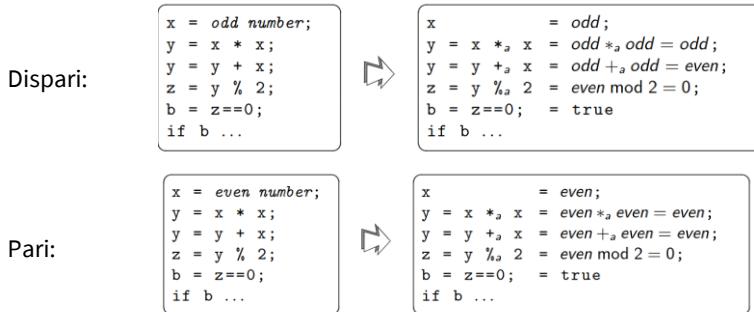
Provare per tutti i possibili valori è, praticamente, un **attacco bruteforce**. Altrimenti, potrei scegliere solo alcuni input di test e usare solo quelli per capire. La issue è che rischio che magari in realtà non era opaco!

Lei con un collega ha provato a fare dei test stimolando dei benchmark con input di riferimento: **molti predicati andavano solo in true o solo in false, pur non essendo predicati opachi.**



Allora, si sono messi nel caso del predicato opaco $\forall x \in \mathbb{Z} : 2 \mid (x^2 + x)$.

1. **Bruteforce:** Facendoci un attacco e facendo tutte le combinazioni su una variabile a 16 bit, ci vogliono **8.83 secondi** a realizzare che è un predicato opaco.
2. **Analisi (astratta) intelligente:** Se invece so, ad esempio, che x ha dominio “numeri dispari”, posso fare considerazioni.



Questo mi fa capire che, **se azzeccassi il dominio giusto, mi basterebbe simulare 2 esecuzioni** anziché chissàquante! In quest'modo becchiamo 66176 domini opachi. In 2 esecuzioni ho finito e in 8 secondi becco millemila casi simili.

Il problema è capire che domini usare! Ci vorrebbe un modo automatico per estrarre l'occhiale giusto, ma non è fattibile.

Generalizzazione dell'attacco astratto: come capisco il dominio?

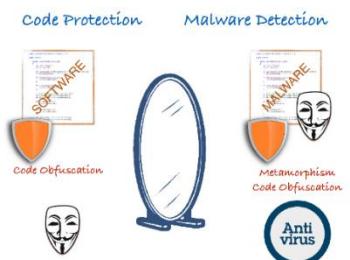
Salta i dettagli, ma sono riusciti a generalizzarlo.

Traslazione in malware detection

Tutta questa formalizzazione sulla code obfuscation può essere **reinterpretata nella malware detection**, dove chi usa la difesa è il malware, e chi attacca è l'antivirus.

I malware usano tecniche di **offuscamento** per **non essere riconosciuti dai tool**; il “metamorfismo” dei malware è l’offuscamento del malware di modo che la signature del malware non sia riconosciuta.

Nei tool di malware detection, tutti funzionano ugale:



1. **Creano un modello del comportamento malevolo**
2. **Metodo per verificare se un programma presenta quel comportamento.**

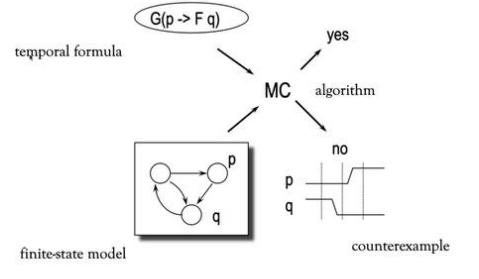
Il modello del comportamento malevolo non è altro che **una proprietà del codice!**

Specificare il comportamento malevolo

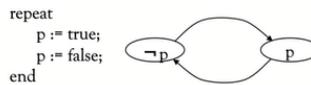
Abbiamo tre modi per specificare il comportamento malevolo:

Model checking MD

- **Modello astratto del malware:** formula in logica temporale (specifica alcune proprietà temporali del comportamento)
- **Modello astratto del programma possibilmente infetto:** struttura di Kripke, che si deriva dal CFG del programma.
- **Detection:** algoritmo di model checking, che decide se la struttura di kripke soddisfa al formula logica che modella il malware. Esistono tanti tool automatici che lo fanno!



Un modello di Kripke è un automa dove gli stati sono etichettati da prediciati.



Alcuni esempi di strutture di kripke e formule temporali:

Modalità universali	Modalità esistenziali
<p>AG p</p> <p>In ogni cammino è sempre valido p; p è sempre vera.</p>	<p>AF p</p> <p>In ogni cammino c'è sempre un punto in cui la proprietà è sempre vera.</p>
<p>EG p</p> <p>Esiste un cammino in cui p è sempre vera.</p>	<p>EF p</p> <p>Esiste un cammino in cui, in un punto, p è vera.</p>

Esistono anche molti altri operatori, che permettono di definire in maniera molto precisa le proprietà.

Kinder et al. "Detecting malicious code by model checking" DIMVA 2005

Nella paper di Kinder et al hanno pensato bene di esprimere l'essenza dell'essere malevolo come una **proprietà di una logica temporale**. Per farlo, hanno deciso di estendere le logiche temporali classiche CTL in **CTPL**.

In CTPL, ogni comando in Assembly diventa il predicato di una logica dove l'op code è il nome del predicato, e gli operandi sono gli argomenti.

`cmp(ebx,[bp-1])`

Dopo di che le si rendono **simboliche**: anziché dire che faccio (ad esempio) una mov con il registro EAX or EBX or..., posso usare un simbolico e lo catturo con “esiste”. Comunque è solo zucchero sintattico, l'espressività della logica non cambia.

EF(mov eax, 937 AND AF(push eax)) OR
EF(mov ebx, 937 AND AF(push ebx)) OR
EF(mov ecx, 937 AND AF(push ecx)) OR
CTL

→

Exits r EF(mov r, 937 AND AF(push r))

CTPL

Quando poi si verifica la formula, bisognerà fare un **binding** per capire a **quale specifica istanza** di registro è vera la formula.

```

1.  $\exists L_m \exists L_c \exists v_{F_{t,0}} ($ 
2.  $\exists r_0 \exists r_1 \exists L_0 \exists L_1 \exists c_0 ($ 
3.  $\quad EF(\text{lea}(r_0, v_{F_{t,0}}) \wedge EX E(\neg \exists t(\text{mov}(r_0, t) \vee \text{lea}(r_0, t))) U \#loc(L_0)) \wedge$ 
4.  $\quad EF(\text{mov}(r_1, 0) \wedge EX E(\neg \exists t(\text{mov}(r_1, t) \vee \text{lea}(r_1, t))) U \#loc(L_1)) \wedge$ 
5.  $\quad EF(\text{push}(c_0) \wedge EX E(\neg \exists t(\text{push}(t) \vee \text{pop}(t))))$ 
6.  $\quad U(\text{push}(r_0) \wedge \#loc(L_0) \wedge EX E(\neg \exists t(\text{push}(t) \vee \text{pop}(t))))$ 
7.  $\quad U(\text{push}(r_1) \wedge \#loc(L_1) \wedge EX E(\neg \exists t(\text{push}(t) \vee \text{pop}(t))))$ 
8.  $\quad U(\text{call}(\text{GetModuleFileNameA}) \wedge \#loc(L_m)))$ 
9.  $)$ 
10.  $\wedge (\exists r_0 \exists L_0 ($ 
11.  $\quad EF(\text{lea}(r_0, v_{F_{t,0}}) \wedge EX E(\neg \exists t(\text{mov}(r_0, t) \vee \text{lea}(r_0, t))) U \#loc(L_0)) \wedge$ 
12.  $\quad EF(\text{push}(r_0) \wedge \#loc(L_0) \wedge EX E(\neg \exists t(\text{push}(t) \vee \text{pop}(t))))$ 
13.  $\quad U(\text{call}(\text{CopyFileA}) \wedge \#loc(L_m)))$ 
14.  $)$ 
15.  $\wedge EF(\#loc(L_m) \wedge EF \#loc(L_c))$ 
16.  $)$ 

```

Il mostro qui a sinistra (**cristodio**) è una **formula CTPL** che modella il comportamento di un worm, ovvero che crea copia del proprio comportamento.

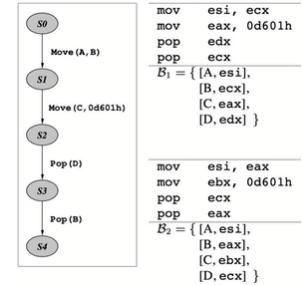
La verifica è comunque efficiente, ma il problema è che la formula va fatta a mano. T_T L'analizzatore di malware dovrà analizzare il malware e estrarne (a mano) questa formula. Richiede una forte conoscenza di CTPL e del malware. **Quindi il trick sarebbe scoprire un modo per estrarla automaticamente** (giovani menti! Tocca a voi! Forse machine learning?)

Static analysis of executables to detect malicious code patterns. USENIX'03

Si cerca di **formalizzare in un automa**. Prendo un automa, lo etichetto con predici o pattern che dicono proprietà dell'esecuzione lasciando "libere" delle variabili; poi trovo un malware che con certi valori "matcha" il mio automa.

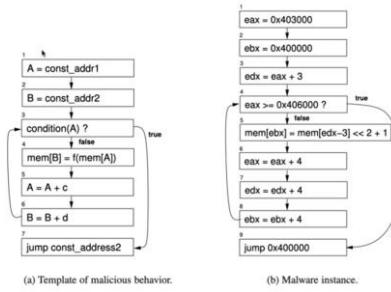
Anche qui, il lavoro è **prendere il malware e costruire un automa con degli aspetti simbolici** (il simbolico è il trick! Altrimenti per sfuggire mi basta fare renaming o scambiare registri).

Semantics-aware MD



Semantics-aware Malware Detection S&P 2005

Creano un **template del malware**. Se il template e l'istanza hanno la **stesso effetto sulla memoria**, DETECTION.



Learning MD

Il problema sostanziale è che l'offuscamento cambia la sintassi mantenendo la semantica, quindi è inutile guardare la sintassi: dovrei verificare la semantica. Questo movimento all'inizio prova a usare gli strumenti con l'analisi dei comportamenti esistenti (automi e model checking). Tutt'oggi vengono applicati, ma ora si usa molto il **learning**: non sapendo descrivere bene formalmente, si preferisce provare un approccio di classificazione partendo da una **batteria di virus**. Ci vuole semantica!!!

Normalizzazione MD

"Normalizing metamorphic malware using term rewriting" SCAM 2006

Si sono chiesti se **esiste una normalizzazione**. Per contrastare l'offuscamento, che diversifica, non possiamo **cercare una forma normale di un programma alla quale ridurre tutti i programmi con una certa semantica**? Esiste tutto un background sullo studio di come capire se sistemi sono confluenti o meno, e come confluirlì (?).

Applicare questo framework ai malware detector: semantic malware detector

Dalla Preda at al. POPL 2007

Quindi, usando la nostra definizione di scudi e attaccanti, abbiamo creato un **framework formale che dimostra soundness e completeness** (no falsi positivi, no falsi negativi). Insomma, il framework che permette di **capire la relazione fra attaccanti e difensori** permette anche di **verificare la capacità degli antivirus** che usano le strutture di kripke o gli automi (che sono astrazioni della semantica).

Possiamo anche **comparare l'efficienza dei diversi AV**, comparando le classi di offuscamento che riescono a gestire.

Limite di queste tecniche di detection: metamorfismo

Lasciando stare quelle signature-based, che pur essendo usate fan cagare... I modelli che si basano sulla semantica hanno un limite comune per cui potranno sempre essere aggirate: **questi modelli sono disegnati avendo in mente gli**

offuscamenti utilizzati. Es. lo scrittore di malware fa il rename? Ok, non guardiamo i nomi. Insomma cerca un offuscamento che rimane invariante attraverso l'offuscamento.

Quindi il limite è che **devo avere conoscenza a priori** delle trasformazioni usate dal malware.

Caso Metaphor

Per esempio, *Metaphor* era un virus *proof of concept* che aveva una *metamorphic engine* ed era *self modifying* usando una serie di regole di compressione ed espansione. Il 70% del codice alla fine risultava metamorfo!

Guardando *Metaphor* ci si chiede: se un *malware* ha al suo interno un meccanismo di modifica, non possiamo **estrarre sistematicamente il meccanismo di modifica**, senza avere conoscenza delle trasformazioni usate? Se capissi il meccanismo delle varianti, beccherei tutte le varianti! In questo modo potrei fare una firma in base a questo reversing, creando una **firma che vale per tutte le varianti**.

Semantica delle fasi

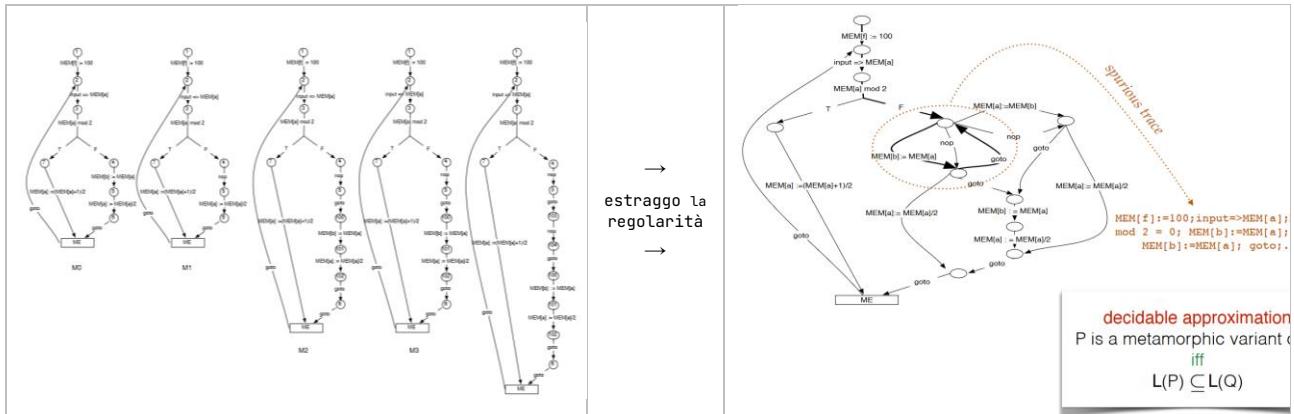
Ci si è chiesto quindi in generale come analizzare i programmi che si automodificano. Per analizzarli serve un **modello semantico in grado di catturare l'evoluzione del codice**. Questo si è concretizzato nella **semantica delle fasi**.

Durante l'esecuzione del programma ci saranno dei calcoli che modificano il codice, e altri che fanno il calcolo per cui è stato disegnato il programma. Vorriamo estrarre le varianti, e quindi riconoscere i punti in cui in realtà viene modificato il programma.

Così catturo la regola di generazione, piuttosto che le singole varianti.

L'idea è, avendo la semantica e uno stato di esecuzione e i dati calcolandi, che se il programma è un programma che si automodifica avrò dei punti in cui cambia la parte di dati del programma; voglio estrarre l'evoluzione del codice. Questo si può caratterizzare con una semantica, detta semantica delle fasi, ma **ovviamente non è calcolabile** perché sennò è troppo facile :)

Quindi bisogna astrarla!



È un'approssimazione: **ho aggiunto anche comportamenti non reali!**

La domanda che nasce da questa questione è: ma se quindi io prendo codice che si automodifica e riesco a costruire un automa che mi fa vedere tutte le varianti dal codice; assumendo di avere tutte le varianti del codice, riesco a estrarre queste regole? **Sì** ma magari sono spurie! Genero **un'approssimazione**. Sarebbe nice automatizzarlo.

Offuscamento e analisi dinamica

Vorremo capire se si può fare o meno, e se c'è un framework formale o meno. Ci racconta come la stanno pensando loro, in termini molto formali quindi con Snoopy.

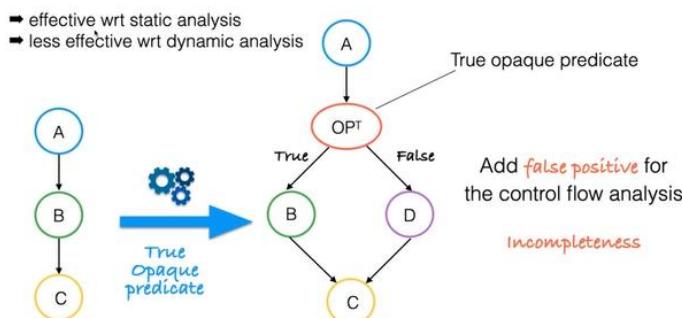
L'**analisi statica** è completa se non ha falsi positivi.

Completeness

L'analisi dinamica non può vedere un falso positivo, dato che vede un sottoinsieme delle tracce da cui cerca di estrarre verità. Per peggiorarla quindi devo aumentare le cose che non vengono viste (=aumento le tracce).

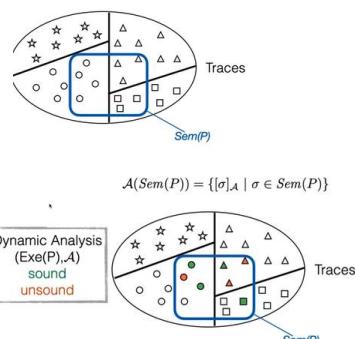
Nell'analisi dinamica **si analizza un sottoinsieme finito di tracce per derivare informazioni su tutto il programma**, come già succede in program testing. Per fatterla, dunque, posso **aumentare i falsi negativi/le tracce da seguire**. Ma posso inficiare l'analisi dinamica attraverso la modifica del codice?

Esempio: predicato opaco



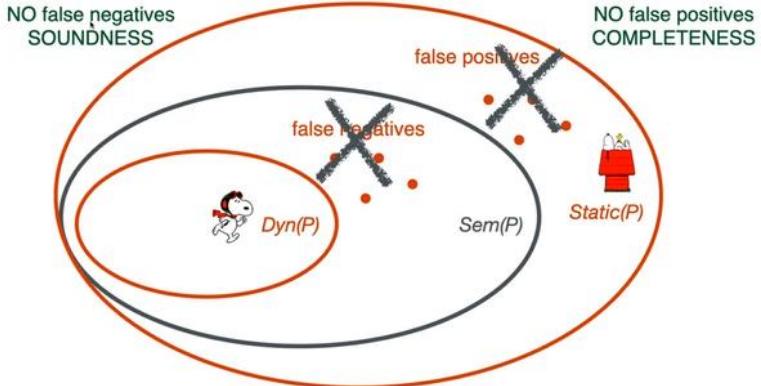
classe di equivalenza. Mettendo un predicato opaco unknown, con quindi due tracce distinte ma equivalenti, il numero di tracce necessarie per coprire tutte le proprietà aumenta.

Ci restringiamo alle **proprietà che valgono su una traccia** (es. Ci sono alcune proprietà – tipo quelle di non interferenza – che no nposso decidere guardando 1 traccia, ma almeno 2.). Queste proprietà prendono tutte le tracce possibili e le partizionano in classi di equivalenza.



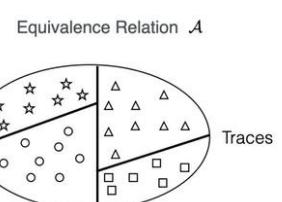
L'**analisi dinamica** è completa se non ha falsi negativi (aka casi che stanno nella realtà ma non vengono visti).

Soundness



Un predicato opaco confonde l'analisi statica aggiungendo falsi positivi (cose impossibili sembrano possibili). Questo non confonde l'analisi dinamica, perché per sua natura non può vedere falsi positivi! Non posso avere esecuzione di qualcosa che in realtà non viene eseguita xD

Quindi, per confondere l'analisi dinamica – che guarda un sottoinsieme di tracce – possiamo **diversificare**, costringendo l'analisi dinamica ad **aumentare il # di tracce da vedere**. Aumentando le tracce, rendo più difficile trovare almeno una traccia per ciascuna



Selezioniamo alcune siano la semantica del programma.

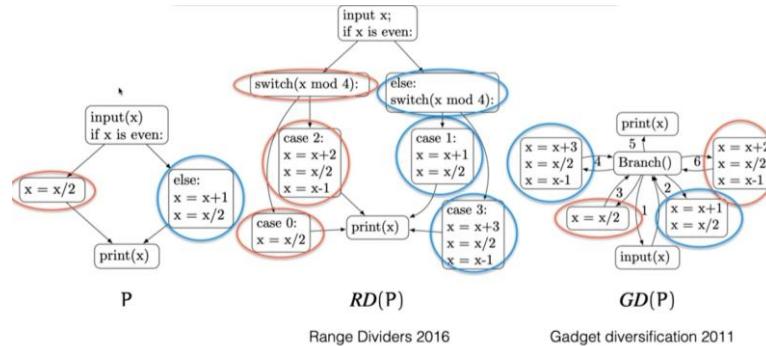
Se dinamicamente considerassi come tracce quelle rosse, avrei dei falsi negativi perché non mi accorgerei del comportamento quadretto.

Esempi

- **Encoding - decoding:** Non ha davvero effetto sull'analisi dinamica.

P input x; sum := 0; while x < 50 • X = [x, 49] sum := sum + x; x := x + 1;	T(P) input x; Encoding x := 2*x; sum := 0; while x < 2*50 • X = [x, 2 * 50 - 1] sum := sum + x/2; x := x + 2; Decoding x := x/2;
---	--

- **Offuscamento dinamico CFG:** Questo effettivamente confonde l'analisi dinamica! Impongo all'attaccante di provare più input.



L'analisi dinamica si basa sul **coverage**: so che **non posso avere la certezza di aver coperto tutto il coverage**. Ci sono criteri di coverage diversi:

- Statement coverage
- Count-statement coverage (ogni statement deve essere eseguito N volte)
- Path coverage
- Count-path coverage

Di fatto questi criteri di copertura non sono altro che delle **relazioni di equivalenza della traccia**; **due tracce sono equivalenti se rispettano la stessa coverage**. Per ogni proprietà che voglio verificare esisterà una relazione che mi dice quanto di ciascuna coverage deve essere verificato. **Per ora non è ancora nulla di formalizzato**.

8 - TAMPER PROOFING

Tamper proofing

Il tamper proofing consiste nell'assicurarsi che il programma viene eseguito come inteso, anche in presenza di avversari che cercano di cambiare o monitorare l'esecuzione.

L'obiettivo è **accorgersi di tampering**.

Con il tampering posso avere numerosi obiettivi; per esempio:

- Far balzare il controllo della licenza o aggiungere modifiche mancanti. Si può agire sia modificando il codice che emodificando l'ambiente di esecuzione.
- Munizioni infinite
- Far funzionare un programma finito il periodo di valutazione
- DRM...

Il tampering può succedere in tanti modi:

- Modificare l'eseguibile
- Aviarlo su un ambiente modificato: OS modificato, dinamic linker modificato, librerie modificate, emulazione, debugger

Tutti gli algoritmi di tampering hanno due fasi:

- **Check**: garantire che il codice è integro e non ci sono state manomissioni
- **Respond**: cosa fare se la verifica rileva del tampering

Check

Lo si fa verificando qualche proprietà invariante.

- **Code checking**: verifico che il codice sia invariato attraverso un hash.
- **Result check**: meccanismo di challenge-response per cui stimolo l'applicazione e mi aspetto un certo tipo di risposta, e come verificare se la risposta è autentica.
- **Environment checking**

La precisione della detection viene misurata in base a quanto presto riconosciamo la manomissione; l'ideale è accorgersene prima che la parte di programma modificata venga eseguita.

Respond

Varie azioni:

- **Terminare** il programma
- **Restorare** la porzione manomessa con la versione originale
- **Modificare il comportamento del programma per concluderlo** (es. in un videogioco sbuca un megamostro brutto e imbattibile)
- **Peggiorare la performance**
- **Punire l'attaccante distruggendone l'ambiente**

Checking by Introspection

Prevede di calcolare la **hash di una zona di codice**. È comodo perché è veloce

Possono essere:

- **Static**: si cercano pattern sospetti sul codice, tipo la zona di inizializzazione
- **Dynamic**: cerca nel codice

```
.....
start = start_address;
end   = end_address;
h = 0;
while (start < end) {
    h = h ⊕ *start;
    start++;
}
if (h != expected_value)
    abort();
goto *h;
.....
```

Rete di check and respond

Protecting software code by guards. Chang and Atallah DRM Workshop 2001

Di norma si usa una **rete di check and respond** (check che controllano i check che controllano il codice), per evitare che qualcuno non abbia agito su expected value o mettendo un || true.

Tipicamente quando il check fallisce ci sono **funzioni di repair**. Li usava Skype con due livelli di check (check che controllano check che controllano il codice). Domanda mia scema: ma allora non è più facile modificare la versione "safe" di B che poi viene repairata, e sminchiare ad cazzum il codice come vogliamo così poi quando fa il repair ci attacca esattamente la versione modificata che vogliamo? TODO

L'idea è che check e respond andranno messi con un certo criterio – ad esempio, è buona cosa controllare che **il respond segua il check**; ogni **respond deve dominare una check region** (aka, grafic blabber per dire che devi passare per forza dal check/respond prima di eseguire)

Generare le funzioni di hash

Per cercare di limitare le funzioni che riescono a **riconoscere** i controlli sull'hash attraverso **pattern matching**, bisogna diversificarle.

```
typedef uint32* addr_t;
uint32 hash1 (addr_t addr, int words) {
    uint32 h = *addr;
    int i;
    for (i = 1; i < words; i++) {
        addr++;
        h ^= *addr;
    }
    return h;
}
```

Simple hash XOR-based

```
typedef uint32* addr_t;
uint32 hash2 (addr_t start, addr_t end) {
    uint32 h = *start;
    while (1) {
        start++;
        if (start >= end) return h;
        h ^= *start;
    }
}
```

Simple hash variant

Si può anche pensare di aggiungere o sottrarre numeri random alla hash, oppure anche di calcolare l'hash solo di alcune righe alternate. Sono semplici, perché il pattern matching è semplice.

Skype aveva generato **centinaia di check diversi** con un modo complicato per generare indirizzo iniziale e finale: gli indirizzi sono generati dinamicamente, ma l'operazione da fare con l'hash era scelta a caso (add, sub, xor,...) ...ma essendo un pattern è stato riconosciuto il pattern e ciaoone :')

Nascondere la protezione: corrector slot

```
h = hash(start,end)
if (h != [0xca7babe5]) abort()
```

not stealthy!

Il controllo della hash non è molto stealthy, perché usa valori probabilmente molto diversi da quelli usati nel resto del codice. È poco stealth.

```
start: 0xab01cd02
0x11001100
slot: 0x?????????
0xca7ca7ca
end: 0xabcddefab
```

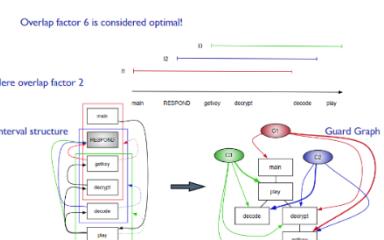
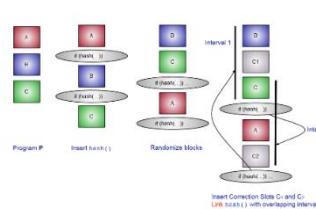


```
h = hash(start,end);
if (h) abort();
```

Quindi, posso nascondere la locazione del controllo della protezione, è stato proposto che nella zona di cui posso calcolare la hash metto un "buco", detto corrector slot, e poi lo vado a riempire in modo tale che hash del codice + quello spazio vuoto farà 0. Il test, così, diventa un confronto su zero che è molto più normal-looking. Dato che la hash è una sommatoria, è molto facile forzare il valore a zero ☺

Posso anche controllare un blocco in combinazione da più di una funzione di hash, con overlapping.

Un overlap factor di 6 ottimale (ovvero i blocchi sono sovrapposti da almeno n checker). In questo modo per l'attaccante è più difficile sgamarli tutti e 6!



Attaccare la introspection (self-hashing)

In generale, per attaccare gli algoritmi di introspection è necessario:

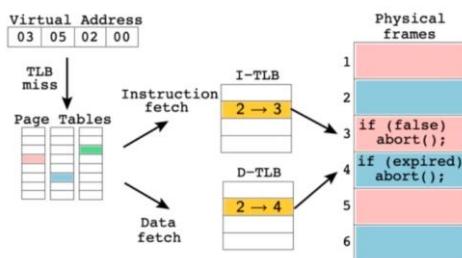
- Analizzare il codice e trovare i checkers (elimina a catena dei check)
- Analizzare il codice e trovare i responders
- Disabilitare check/respond senza distruggere il resto del programma

Questo è possibile perché è possibile riconoscere che è avvenuta una check/respond in quanto ho codice che viene trattato come dato (es. calcolo dell'hash sul codice, restore del codice) – cosa che ovviamente è inusuale, ed è diverso da quello che succede di solito in un codice che non si automanipola.

L'attacco viene eseguito andando ad agire sull'ambiente. Prendo due copie del programma:

- Copio il programma P in P_{orig} .
- Modifico P come voglio, creando la versione hackerata P'
- Modifico il kernel K del sistema operativo, in modo che le letture dei dati siano direzionate su P_{orig} e le letture delle istruzioni invece su P' .

È possibile sapere se la lettura era riferita a dati o a istruzioni perché sull'UltraSparc, l'hardware dà informazione all'OS di un TLB miss dando un'eccezione – e questa è diversa per dati e istruzioni.



Problemi dell'instrospection

Il problema alla base dell'instrospection è che fa operazioni inusuali, ovvero legge il proprio codice. Questo non è per un cazzo stealthy.

State inspection

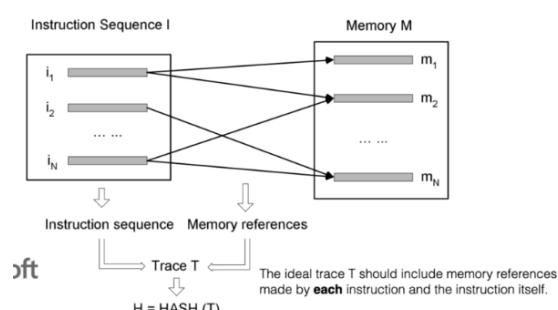
Allora soi è penato se fosse possibile creare un altro principio. Anziché chiederci se il programma è corretto ci chiediamo se **il programma è in uno stato valido**. Se l'hash è puramente sintattico, **l'inspection è più semantico!**

Oblivious hashing

Oblivious hashing: a stealthy software integrity verification primitive. Cheng et al. Information Hiding 2002

Si prova a proiettare il calcolo dell'**hash sulla traccia di esecuzione**.

L'idea è: ho una sequenza di istruzioni seguite, e le istruzioni accedono a celle di memoria. Se calcolo l'hash della traccia, mi sto concentrando su **cosa fa effettivamente** anziché su cosa c'è scritto!



Implementazione: per fare questo devo **modificare il codice e la sua esecuzione**; nel paper identificano nella traccia una serie di cose precise di cui fare l'hash:

- Left-hand dell'assegnamento: ogni volta che valuto un'espressione che viene assegnata, faccio la hash
- Hash dei predici fatti per gli IF

Quindi ogni istruzione di questo tipo viene trasformata in hash, e ottengo una sequenza di hash che vengono aggiunti all'hash "totale" della storia.

L'idea è semplice, ma non ha avuto molto seguito perché ha un limite grosso: **non possiamo verificare ciò che dipende dagli input** (dato che non saprei a prescindere l'hash della traccia che prenderò). Poi ecco, **solo il 13% delle tracce è deterministica** quindi ecco.

Practical Integrity Protection with Oblivious Hashing, ACSAC 2018

Successivamente provano a renderlo utilizzabile. L'idea è innanzitutto di capire quali istruzioni dipendono o meno dall'input, e questo può essere fatto via analisi del codice:

- **Data dependency**: istruzioni dove almeno uno degli operandi dipende dall'input
- **Control flow dependencies**: l'esecuzione o meno dipende dall'input (sono in un branch o loop dove la guardia dipende dall'input).

Queste sono analisi classiche che già esistono da tempo.

Poi si classificano le istruzioni come:

- **III** indipendenti dall'input: *oblivious hashing già funziona*
- **DDI** data-indipendenti
- **DDI** data-dipendenti
- **CFDI** control-flow dipendenti

Inconsistent hash values

DDI e CFDI sono le due categorie che ci fanno hash diversi per input diversi, ovvero presentano hash values inconsistenti.

Se dipendono dall'input abbiamo due problemi possibili:

- Non riusciamo a calcolare la hash perché ci manca il dato
- Non riusciamo a calcolare la hash perché non sappiamo quante volte viene eseguito un loop

Per sbrogliare la situazione, definiscono queste short-range-oblivious-hashing, che sono una tecnica composta da un insieme di pezzettini di codice che sono usati sempre tutti insieme. Per esempio, dato un ciclo, se io calcolo l'hash del ciclo. Per esempio, se calcolo l'hash del corpo di un loop che non so quante volte verrà eseguito, comunque quell'hash è valido per la prima esecuzione.

Questo risolve il problema per i cicli :) L'hash sarà sempre uguale solo sul corpo del loop, perché l'hash viene fatto sull'esecuzione simbolica.

Noice

Remote tamper-proofing

Quello che ci presenta fa parte del progetto ReTrust, dell'università di Trento, che l'ha coinvolta. Questo mega progetto degli anni 2001-2011 propone il seguente problema: **come fa un server da remoto a fidarsi di un client?**

Sicuramente possiamo usare le tecniche di hash come challenge-response; tipo che il server chiede di mandare l'hash di una porzione del codice.

```

1 enum period {Peak, OffPeak, Normal};
2 float computeUsage(float *kwMinute, int size , enum period rate){
3     float usage = 0;                                // III
4     for( int i=0;i<size ; i++){                     //DDI
5         float rating = 1.0;                          //DII|CFDI
6         if( rate == Peak){                           //DDI
7             rating = 2.0;                            //DII|CFDI
8         } else if( rate == OffPeak){                 //DDI
9             rating = 0.5;                           //DII|CFDI
10        }
11        usage += kwMinute[i]*rating;               //DDI
12    }
13    return usage;                                  //DDI
14 }
15 ...
16 meterUsageCycle(float *kwMinute, int size , enum period rate){
17     kwHour = 0;                                    // III
18     if( isHoliday () ){                         //DDI
19         enum period normalRate = Normal;        //DII|CFDI
20         kwHour = computeUsage(kwMinute, size, normalRate); //DDI|CFDI
21     } else {                                     //DDI|CFDI
22         kwHour = computeUsage(kwMinute, size, rate); //DDI|CFDI
23     }
24     ...
25 }
```

Sarebbe possibile risolvere anche con dispositivi hardware eh, ma il progetto voleva provare a trovare delle tecniche software.

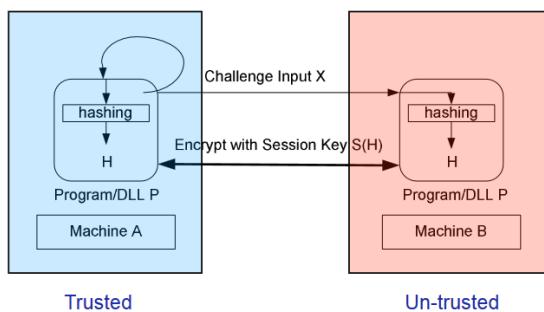
In generale, la soluzione è sempre muoversi sul server (SW as a service), ma significa avere un load importante sul server e latenza per il client. Di conseguenza, ci sono vie di mezzo per bilanciare dove svolgere cosa. Si ragiona anche sulla possibilità di non dare niente di interessante / manomittibile al client 😊

Il grosso problema, in generale, è che il client può mentire.

Oblivious hashing nell'esecuzione remota

Basically usiamo oblivious hashing per farci challenge-response; computiamo l'hashing as usual, ma i valori della challenge ci vengono dati remotamente dal server.

Il server manda come challenge la zona di cui vuole che gli voglia restituita l'hash, e il client deve rispondere. Il problema è che il client modificato potrebbe mentire.



Code slicing

Hiding Program Slices for Software Security. X. Zhang and R. Gupta. CGO 2003

È un algoritmo sviluppato per evitare la pirateria, e nello specifico per evitare che l'attaccante riesca a far funzionare la sua copia.

L'idea è quella di dividere i moduli del software in;

- **Componenti open:** sono installati e eseguiti su macchine untrusted
- **Componenti hidden:** sono installati ed eseguiti su macchine trusted, aka sul server.

Tutto sta nel capire come dividere le cose. I componenti open possono essere rubati, ma non danno tutte le funzionalità. I componenti hidden restano sul server, dove richiede molto più effort fare modifiche.

Resilienza: derivare i componenti hidden è molto difficile	Costo: latenza per far comunicare i componenti open e hidden.
---	--

Quest'idea si basa principalmente sul concetto di **program slicing**, che prende una variabile del programma (presa a criterio per lo slicing) e tutte le istruzioni che vi dipendono da lì in avanti.

Quindi, se una certa variabile è da proteggere, **sposto tutte le istruzioni che ne dipendono sul server :**

I problemi sono dipendenti da quanta protezione sto aggiungendo: questo aggiunge comunicazioni extra che possono aumentare l'overhead dal 3 al 58% - ma questa misurazione è fatta su una LAN, e nella realtà con la rete vera (e quindi con la latenza) è **molto difficile usare questa tecnica**.

CLIENT	SERVER
<pre> int client(int x,int y){ f1(x,y); f2(y,x); c = 2*x+4; f3(c); } int sum = 0; f4(sum); f5(); return x*f6(); } </pre>	<pre> int Ha = 5; int Hc = 0; int Hsum = 0; void f1(int x,int y){ Ha=4*x+y; } boolean f2(int y,int x){ if (y < 5){ Hc = Ha*x + 4; return true; } else return false; } void f3(int c){ Hc = c; } void f4(int sum){ Hsum = sum; } void f5(){ for(int i=Ha;i<10;i++) Hsum += i; } int f6(){ return Hsum+Hc; } </pre>

Barrier slicing

Barrier slicing for remote software protection, Ceccato et al. SCAM 2007

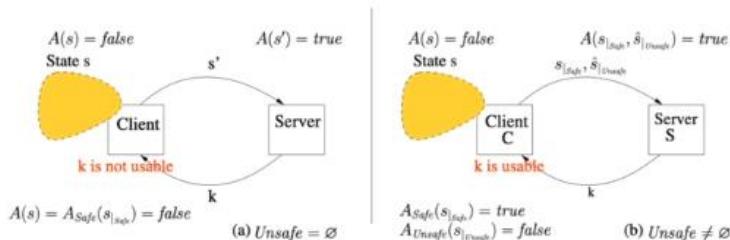
Si è cercato di migliorare le cose proponendo una variante di questa soluzione, ovvero lo slicing con barriera. In questa versione, prendo tutto ciò che dipende da una variabile... ma solo fino a una certa barriera.

Decido io dove mettere la barriera! Cosa è sensato? C'è qualcosa del client che il server può verificare verificando delle asserzioni sui messaggi che il server gli manda? Il messaggio dipende dallo stato del client, quindi voglio definire delle asserzioni sul messaggio che permettano di verificare la validità dello stato del client.

Sarei a posto se attraverso il messaggio potessi verificare lo stato del client. Per essere più generici posso immaginare che lo stato s possa essere suddiviso in due parti:

- Una parte “**safe**” che parla di quelle variabili su cui se il client mentisse non otterebbe dal server una risposta utile (nel senso che se mente ottiene una risposta diversa)
- Parte su cui invece può mentire, aka sembra vada bene ma ha mentito.

$$A(s) = A_{Safe}(s|_{Safe}) \wedge A_{Unsafe}(s|_{Unsafe})$$



L'idea quindi è di spostare le cose e **mettere come barriere tutte quelle istruzioni che manipolano variabili safe** – aka, le cose su cui non può mentire sono inutile siano spostate sul server.

Insomma, le cose che se fossero manomesse poi non farebbero più funzionare il server sono “automaticamente” protette. L’idea è sempre la remote execution, ma questa idea ci permette di minimizzare il calcolo da spostare

Orthogonal replacement

Remote software protection by orthogonal client replacement. Ceccato et al. SAC 2009

Ortogonalità cerca di significare che due parti di codice sono indipendenti l'una dall'altra.

Ok client: ti lascio eseguire il codice ma **continuo a sostituire il tuo codice concodice ortogonale (=indipendente dalla copia precedente) ed equivalente**, ovvero un offuscamento tale per cui **per l'attaccante avere accesso a una versione precedente non dà alcun vantaggio**. Facendolo abbastanza spesso, sto *resetando la conoscenza dell'attaccante* ☺

Il problema è generare in automatico e all'infinito le copie ortogonali!
Anche intuitivamente, avere due copie del codice per forza deve darti qualche tipo di vantaggio.

Lo studio funzionava prendendo come definizione la seguente, che è molto debole:

Ortogonalità = due statement sono ortogonali se non sono matchati dall'algoritmi di cloning.

Questo si riferisce a delle tecniche di analisi che riconoscono i cloni; nate per riconoscere pezzi di codici equivalenti in un programma più grande, con lo scopo di poter fixare eventuali bachi anche in codice clonato non documentato. Sarebbe più sensato se la definizione di ortogonalità fosse fatta in riferimento a un attaccante.

Il caso di Skype!

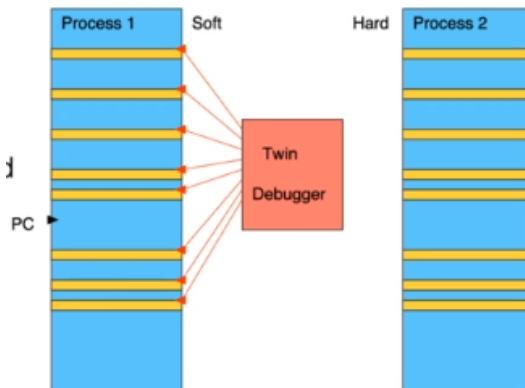
È un po' datato ma all'inizio era così.

<p>Binary packing</p> <p>Il client era crittato, in xor oppure con una chiave TSA. Anche una volta caricato in memoria, rimaneva tutto crittato e veniva decrittato a runtime.</p>	
<p>Anti-dumping</p> <p>L'esecuzione inizializzava l'esecuzione, inizializzava un'area e poi caricava le librerie dinamiche. Poi cancellava il codice.</p> <p>Il codice caricato era cifrato, e quando decifrato andava a cancellare anche parte delle librerie dinamiche caricate.</p>	
<p>Code integrity</p> <p>Usava una rete di check a due livelli.</p>	

+ 300 funzioni di hash per verificare l'integrità del codice e per diversificarle usava vari offuscamenti.

+ analisi di ambiente per capire se era in ambiente controllato, e in caso crash immediato.

Per attaccarlo, in sostanza, si usa il processo gemello.



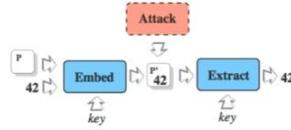
Perché così skippo direttamente il problema di beccare tutte le funzioni di hash! Mando in esecuzione due processi e calcolo le hash sul processo originale. :)

9 – WATERMARKING

Il primo esempio di watermarking è quello della carta Fabriano. Di watermarking digitale si inizia a parlare negli anni 90, soprattutto per le immagini

Ci sono tanti tipi di watermarking, ma in generale sono composti da un programma P e una firma W tale per cui

- W è resiliente agli **attacchi automatizzati**
- W è **stealthy**
- W ha **high bitrate**
- W comporta **poco overhead di risorse**.



Esisterà un programma di **embedding** che firma il programma, e quindi anche un **estrattore** che riesce a estrarre la firma dal programma. Serve una chiave per lo stesso motivo a cui serve in crittografia: la potenza di questi algoritmi non uò essere nel fatto che gli algoritmi stessi sono segreti, ma devono essere basati su un segreto (chiave) coosciuto dal costruttore.

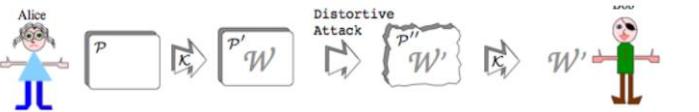
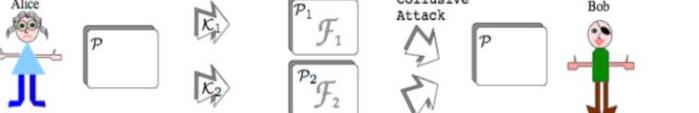
<p>Immagini</p> <p>Un esempio di water marking per le immagini è questo, che altera la luminosità: Si aggiunge un certo delta di luminosità, e poi per capire se era watermarkato o meno si fa la somma delle luminosità e si verifica se era inusualmente alta. Facendolo in questo modo, rimane poco visibile all'occhio umano!</p>	<pre> Embed: Init_RND(K); repeat n ≈ 10000 times i ← RND(); j ← RND(); fix brightness : a_i ← a_i + δ; b_j ← b_j - δ; Extract: Init_RND(K); S' ← 0; repeat n ≈ 10000 times i ← RND(); j ← RND(); sum brightness : S' ← S' + a_i - b_j; • $\sum_i (a_i - b_i) \gg 0 \Rightarrow$ watermark! • Bit-rate: 1 bit per image. </pre>
<p>Audio</p> <p>Viene inserito un watermarking nelle parti “alte” dell’onda audio, aggiungendo dei piccoli eco che noi non possiamo riconoscere</p>	<p>LSB Hiding: Shows an audio waveform with small rectangular pulses representing hidden data.</p> <p>Echo Hiding: Shows an audio waveform with two small rectangular pulses labeled δ₀ and δ₁, representing echo-like hidden data.</p>
<p>Testo</p> <p>Si possono aggiungere interlinee, spazi bianchi, struttura sintattica o addirittura semantica fra i vari sinonimi. Il messaggio è sempre, però, che il messaggio deve rimanere inalterato.</p>	<p>White-Space:</p> <pre>I saw the best minds 12pt { of my generation, 14pt { starving hysterical naked</pre> <p>Syntactic:</p> <pre>It was the best minds of my generation that I saw, starving hysterical naked</pre> <p>Semantic:</p> <pre>I observed the choice intellects of my generation, starving hysterical nude</pre>

Il watermarking serve a disincoraggiare la copia e redistribuzione del software. In realtà esistono diverse versioni di “watermarking” del codice.

- **Watermark**: prevede di prendere il programma originale, inserire una firma ed essa si trova in tutte le versioni
- **Fingerprint**: inserisce firme diverse a seconda dell’utente che lo acquista. (così si capisce da dove sono originate le copie non lecite)
- **Birthmark**: non hanno una fase di inserimento; c’è solo una fase di estrazione, ed estraggono una certa proprietà che è specifica di quel programma.

Proprietà degli algoritmi di watermarking

- **Credibilità:** l'algoritmo di estrazione della firma deve essere credibile, ovvero deve essere altamente improbabile che se viene applicato a un programma non watermarkato si riesca a generare una firma. (estrae firma solo se ce l'abbiamo messa!)
- **Data rate:** misure la qtà di informazione che possiamo inserire all'interno della firma.
- **Stealthiness:** invisibilità; rende più difficile modificare il watermark.
- **Protezione delle parti:** è bene che la firma sia distribuita in tutto il codice e non solo limitata a una firma in una sezione sola.
- **Resilienza:** deve resistere agli attacchi tipici, ovvero...

Sottrattivi: trasforma il programma firmato in un programma senza firma.	
Additivi: vogliamo rendere più difficile inserire una seconda firma. Questo può essere sempre fatto, ma vogliamo legarci a un aspetto temporale per dimostrare quale fosse pressente da prima.	
Distortivi: sono gli offuscamenti: sia di ottimizzazione (es. il compilatore toglie la firma come cosa ridondante) sia fatti in modo che l'algoritmo di estrazione non riesce più ad estrarre la firma.	
Collusivi (solo per il fingerprint): l'avversario compara due fingerprint ed identifica la firma, quindi poi ottiene il programma free.	

Tipicamente si discute della distortività e la sottrattività.

Obiettivi

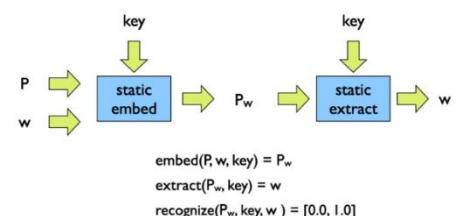
Il watermarking ovviamente **non impedisce copie e distribuzioni**, non è un'azione preventiva; tuttavia:

- **Disincoraggia** le copie illegali
- È utile a **tracciare l'origine della distribuzione** illegale, con la fingerprint
- Posso dimostrare di **possedere il software**.

Watermarking statico

Manipola il programma senza eseguirlo. Dato programma, firma w e chiave key. Abbiamo:

- Un algoritmo di **embed** statico che dato P, w, key ritorna un P_w firmato
- Un algoritmo di **estrazione** che dato P_w , key ritorna la firma
- Un algoritmo di **riconoscimento** che dato P_w , key, w ritorna una probabilità che quel programma contenga quella firma. Questo perché spesso, per contrastare gli attacchi sottrattivi, ho che la firma è in tanti posti diversi e può essere che dopo un attacco riesco a recuperare solo parte della firma.



Limitazioni

Devo assicurarmi che offuscamento e ottimizzazione non vadano a inficiare!

Più nello specifico, sono deboli agli attacchi distorsivi: il watermark può essere distrutto da...

- Ottimizzazioni
- Offuscamenti
- Inlining e outlining
- Decompliamento e ricompilazione.

Quindi? Dinamic momento 😊

Anyway... Dove inserisco l'informazione?

Static data watermarking

Aggiungiamo una **stringa nel codice** con il copyright, tipo

```
> strings /bin/etscape | grep -I copyright  
Copyright © 1998 Netscape Communication Corporation
```

+ Super semplice da inserire
- Banalissimo levarlo lol

Static code watermarking

È sempre bene lavorare per analogia. Quindi l'idea è di sfruttare tecnologia già esistente: il concetto di watermarking rimanda subito al **watermarking di immagini**. Semplicemente, si **inseriscono informazioni/bit in tutte quelle aree delle immagini che non sono percepibili all'occhio umano**. Insomma, scelgo le zone di ridondanza e scelgo una configurazione fra le tante equivalenti possibili.

Ordinamento

Per esempio se posso ordinare cose in un certo ordine, **scegliere sempre lo stesso ordine** può essere una forma di watermarking. Per esempio avendo uno switch case, scelgo un ordine fisso (es. anziché 1-2-3 metto i casi nel mio ordin) per esprimere un certo numero.

È un watermarking **molto poco resistente**:

basta riordinare tutti i case e non solo sto cancellando la firma, ma mi trovo a potenzialmente estrarre altre firme!

Brevetto Moskowitz

Semplicemente: siccome esistono già algoritmi per fare il watermarking dell'immagine, **fracco un'immagine nel software e la watermarko**.

Però, se lo facessi così semplicemente, basterebbe solo buttare via l'immagine per togliere la firma.

Quindi, **dentro l'immagine ci metto anche delle linee di codice fondamentale**, così che se uno riesce a identificare, isolare e togliere l'immagine, automaticamente il programma non andrebbe più.

Brevetto Microsoft

Riordino i blocchi di codice, e metto dei jump per mantenere la semantica.

Lo slowdown è piccolo, ma **non è molto resistente** DATO CHE BASTA RIORDINARE DI NUOVO I BLOCCHI, e in più le tecniche di ottimizzazione del compilatore cancellerebbero tutto il lavoro fatto 😞

Lo slowdown è inincidente, il problema è che non è stealthy

Watermarking in CFG

A graph theoretic approach to software watermarking, Venkatesan et al IH 2001

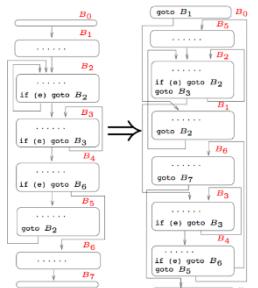
Immaginiamo di codificare un numero in un grafo. Poi costruiamo un programma con un control flow graph con la forma di quel grafo. Lo attacciamo al programma originale; l'extraction deve trovare quel CFG.

I grafici usati per codificare la firma devono avere **forma riconducibile a quella di un programma**; quindi:

- **Riducibili**: non hanno un jump nel mezzo del corpo di un ciclo

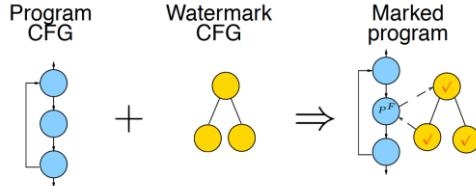


Code Q = Decode (P);
Execute (Q);
}



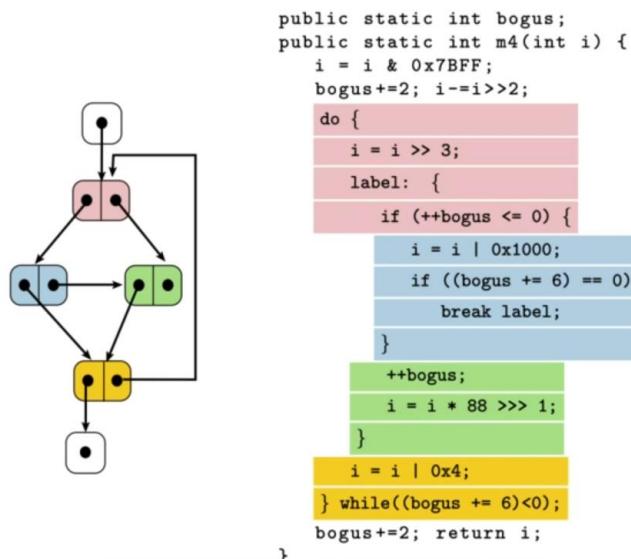
- **Shallow**: il codice (vero) non deve essere nested deeply
- **Small**: quel che ti aspetteresti lol
- **Devono essere resistenti all'edge flips** (negare la guardia e invertire i rami): questo modifica i grafi, quindi il grafo generato deve resistervi.

Quindi prendo i CFG originale, gli aggiungo il CFG di watermark e lo lego attraverso un predicato opaco.



Rimane il problema che l'estrattore deve **capiere quali sono i pallini gialli e azzurri**: basically dovrò mettere dei marking.

Esempio



La cosa sopra funziona perché sappiamo che *m4* è stato progettato per tornare sempre un numero non negativo, e quindi può essere usato come predicato opaco. *Bogus* invece è una variabile globale, e quindi l'attaccante deve fare un'analisi interprocess per sapere che *m4* può effettivamente essere rimosso.

Attenzione: bisogna assicurarsi che il CF non venga ottimizzato dal compilatore.

Estrazione del watermark

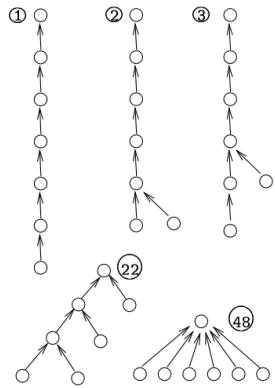
La fase di estrazione è molto delicata, perché devo avere codificato in un qualche modo un mark che mi separi i CF normali da quelli di watermark. Dovrò quindi avere una procedura di riconoscimento che calcoli il mark value di ciascun blocco, e tipicamente:

- O si costruisce un metodo che permette di estrarre il valore anche avendo recuperato solo una certa percentuale del grafo
- Oppure inserisco il grafo più volte

Codifica del numero

Oriented parent-pointer tree

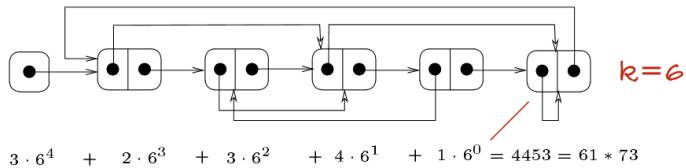
Ho un'enumerazione dei membri della famiglia di grafo con 7 nodi, ordinati in "largest sub tree first". Codifica e decodifica solo polinomiali.



- n is represented by the *index* of the the graph G in some enumeration.
- We must, efficiently, be able to
 1. given n , generate the n :th graph,
 2. given G , find G 's index n .
- Oriented parent-pointer trees ⇒
 1. 655 nodes ⇒ 1024-bit integer,
 2. bit-rate: 1.56 bits per word.

Radix graph

Abbiamo una lista circolare, e ogni elemento punta a un altro, a se stesso o a nulla. La chiave è un valore che funge da base, e in base a dove va il puntatore so quale sia il puntatore che devo moltiplicare per la base.

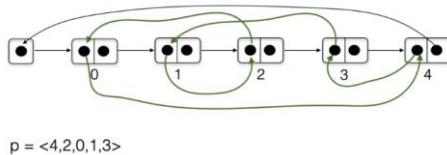


- G is a circular linked list.
- An extra pointer field encodes a base- k digit:

null-pointer	⇒	0
self-pointer	⇒	1
next node pointer	⇒	2 ...
- A 255-node list hides 2040 bits ⇒ bit-rate is 4 bits per word.

Permutation graph

Posso rappresentare il valore come permutazione; in base al valore della permutazione che ho scelto ho codificato qualcosa.



Algoritmo QP

Inserisce il watermark **modificando l'allocazione dei registri**. Il modo in cui lo fa corrisponde all'inserimento di informazione.

Algoritmo per l'analisi (inference graph)

Tipicamente, l'allocazione dei registri avviene basandosi sul cosiddetto **grafo delle interferenze**, che è un grafo con un nodo per ogni variabile, dove è presente un arco fra due nodi se quelle due variabili possono essere vive nello stesso momento – e quindi devono essere salvate su registri diversi.

L'analisi di liveness ci fa mettere gli archi sul grafo, e poi vogliamo colorare il grafo in modo che due nodi connessi da un arco non abbiano mai lo stesso colore – e questo con il numero minimo di colori. Questo graph coloring problem rappresenta l'allocazione dei registri.

1. analisi di liveness

2. traduco in archi e associo i colori

3. traduco in registri

Live variable analysis <pre> V1 := 2 * 2 V2 := 2 * 3 V3 := 2 * v2 V4 := v1 + v2 V5 := 3 * v3 </pre>		<pre> mult R1, 2, 2 mult R2, 2, 3 mult R3, 2, R2 add R1, R1, R2 mult R1, 3, R3 </pre>
--	--	---

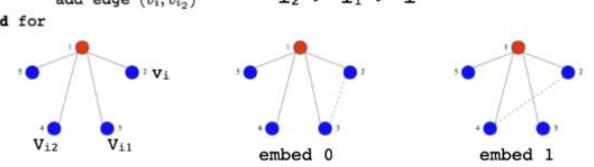
Data questa analisi, l'algoritmo QP **aggiunge concettualmente un arco tra due registri** (benché non è vero che sono legati). A questo punto **sto forzando un cambio colore**, e quindi **aggiungendo un nuovo registro**:



```

for each vertex  $v_i \in V$  which is not already in a triple
  if possible find the nearest two
    vertices  $v_{i_1}$  and  $v_{i_2}$  such that
       $v_{i_1}$  and  $v_{i_2}$  are the same color as  $v_i$ ,
      and  $v_{i_1}$  and  $v_{i_2}$  are not already in triple.
    if  $m_i = 0$ 
      add edge  $(v_i, v_{i_1})$ 
    else
      add edge  $(v_i, v_{i_2})$ 
    end for
  i_2 > i_1 > i

```



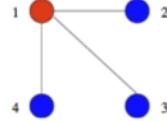
Dato questo, posso codificare valori. **Ovviamente userò più registri di quel che ho bisogno**, ma nella differenza inserisco informazione.

Più nello specifico: tutto si basa sull'ipotesi che le variabili siano numerate. Fatto ciò, si identificano le triplettie clorabili.

Triplette colorabili

Dato un grafo n-colorabile $G = (V, E)$, $\{v_1, v_2, v_3\}$ è una tripletta colorabile se:

- $v_1, v_2, v_3 \in V$ sono effettivamente tre variabili
- $(v_1, v_2), (v_1, v_3), (v_2, v_3) \notin E$ non hanno archi fra loro
- v_1, v_2, v_3 hanno lo stesso colore



Embedding

Supponiamo di voler inserire un messaggio binario $m = m_0 \dots m_k$ dove ogni m è un bit {0,1}.

Per ogni vertice v_i che non è già in una tripletta (e parto dal primo i), trovo i primi v_{i_1}, v_{i_2} che sono dello stesso colore e non sono già in una tripletta. Aggiungo o meno l'arco in più in base a se in m_i volevo codificare uno 0 o un 1.

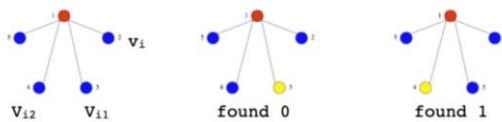
Quindi, è fondamentale ordinare le variabili perché m_i si basa sull'ordine!

Recognition

Evidentemente l'estrattore fa la stessa cosa: se trovo triplettie che non hanno archi connessi ma c'è il colore diverso, a seconda dell'indice della tripletta so se era 0 o 1.

```

for each vertex  $v_i \in V$  which is not already in a triple
  if possible find the nearest two
    vertices  $v_{i_1}$  and  $v_{i_2}$  such that
       $v_{i_1}$  and  $v_{i_2}$  are the same color as  $v_i$ ,
      and  $v_{i_1}$  and  $v_{i_2}$  are not already in triple.
    if  $v'_i$  and  $v'_{i_1}$  are different colors
      found a 0
      add edge  $(v_i, v_{i_1})$ 
    else
      found a 1
      add edge  $(v_i, v_{i_2})$ 
    end for
  
```



Qui la proprietà ridondante è l'allocazione dei registri (ne sto usando più del dovuto)

Considerazioni

- **Facile da attaccare**: basta riallocare i registri, per esempio con il compilatore...

- **Data rate basso:** un bit per ogni 3 variabili non live contemporaneamente; devo vedere quante variabili ho a disposizione :/
- **Sicuramente è stealthy** ☺ Il codice sembra lo stesso “originale”, ed è difficile accorgersi che c’è stato l’embedding
- **Altamente credibile:** non è in un punto preciso ma in tutto il codice, quindi è distribuito ed è molto difficile trovare messaggi che vengono fuori “per coincidenza”-

Watermarking dinamico

Inseriamo la firma in qualcosa che viene effettivamente calcolato – quindi **nella semantica** del programma.

Sono per natura **più resistenti**, perché ci stiamo nascondendo nella semantica, e quindi **non verrà eliminato dagli ottimizzatori** (a meno di analisi profonde che rilevino calcoli inutili, ma difficilmente sono fatti in automatico).

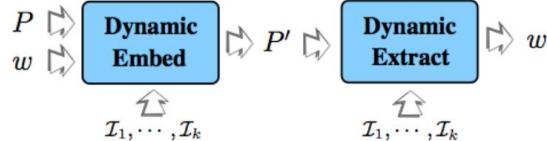
È resiliente a:

- **Attacchi sottrattivi:** difficile individuarlo
- **Attacchi distorsivi:** le trasformazioni solitamente preservano la semantica, quindi preservano anche questo attacco
- **Attacchi collusivi:** è difficile comparare la semantica di due programmi su qualche input

Quindi, c’è un aspetto profondo che ci dice che il watermarking dinamico è più resistente: il watermarking dinamico è una proprietà estensionale nel codice. Il teorema di Rice ci dice che le proprietà estensionali, ovvero quelle che sarebbero rispettate anche da un altro programma con la stessa semantica, non sono decidibili a meno che siano proprietà “sciocche” (tutti e nessuno).

! Il watermarking dinamico è una proprietà **estensionale** e per il teorema di Rice non sono decidibili ☺

Il watermarking dinamico prende in ingresso un programma, un watermarking e un input (e anche una chiave). Viene costruito su una particolare esecuzione, quindi se eseguo il programma con quel particolare input a un certo punto viene costruito il watermark (e il calcolo avviene sempre, ma produce il watermark solo su quell’input).



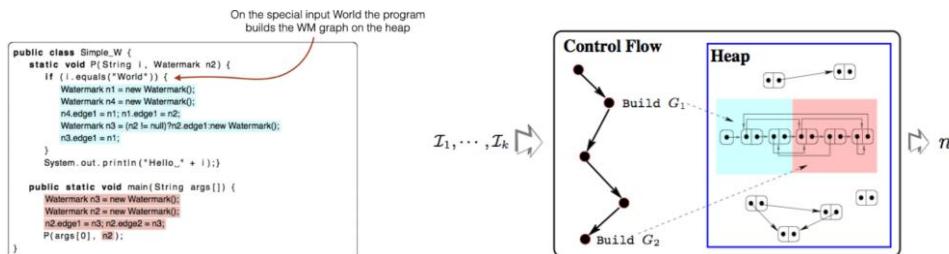
Al solito è bene cercare idee in cose già fatte e problemi difficili per nascondere le proprietà.

Dynamic data-structure watermark

Dynamic graph-based software fingerprint. Collberg et al. TOPLAS 2007

Embeddano il watermark nelle strutture dati dinamiche (heap, stack).

Assomiglia molto al processo dei predicatori opachi, che costruivano strutture dati dinamiche particolari tali per cui era tutto opaco. Qui ho una **sequenza di input che mi costruisce una struttura dati con una certa shape**, la quale codifica il watermark; solo un certo input mi genera il watermark.



È molto buono: un attaccante distortivo dovrebbe mettere mano alle strutture dati senza interferire con la semantica. Poiché questa analisi dinamica è complessa, soprattutto con le destructive updates (=a ogni update tolgo e metto cose).

L'idea è che sia una sequenza di input, immaginando un programma che può essere stimolato in più punti.

Valutazione

- **Stealthiness:** il codice sembra codice normale
- **Resilience:** Molto buona! Un attaccante distortivo potrebbe mettere mano alla costruzione delle strutture dati dinamiche per manomettere il watermark, e dovrebbe farlo assicurandosi di non interferire con la semantica. Poiché l'analisi di strutture semantiche è complessa (soprattutto in presenza di disruptive update, che sono NP), è difficile che l'attaccante vada a mettere mano a queste cose – sarebbe troppo costoso! Quindi è molto resistente.

Implementazione

Experience with software watermarking, Palsberg et al, ACSAC 2000

Altri ricercatori hanno implementato e testato la cosa. Hanno provato a marcare una serie di programmi (qui sotto), e li hanno eseguiti con i reference input; si è osservato che:

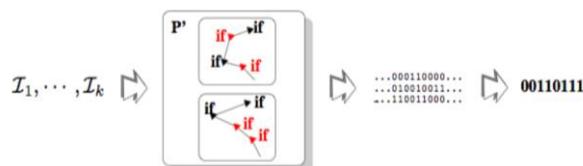
- Il tempo di esecuzione prima e dopo peggiora solo marginalmente
- Prendendo due offuscatori di java, hanno testato che il watermark resiste.

program	description	test input
javac	a compiler for Java	the JavaCup source code
javadoc	a Java API documentation generator	the JavaCup source code
JavaCup	an LALR parser generator for Java	the CORBA grammar
JTB	JTB [16] is a frontend for The Java Compiler Compiler from Sun Microsystems	the Java 1.2 grammar
JavaWiz	the watermarking system reported in this paper	the JavaCup source code
compress	a java virtual machine spec benchmark	some tar files shipped with compress
BLOAT	BLOAT [9] is a Java bytecode optimization tool	the JavaCup source code

Path-based watermarking

Dynamic path-based software watermarking, Collberg et al. PLDI 2004

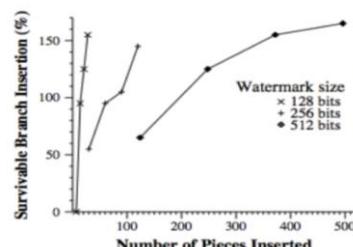
Dato un input, guarda la sequenza di if che vengono eseguiti e dà uno 0 se eseguo true e 1 se false (o viceversa). Quindi l'esecuzione codifica un numero!



Valutazione

- **Stealthiness:** molto stealthy
- **Resilience:** si spacca con l'offuscamento, basta flippare 😞 Quindi l'idea è di fare la ridondanza. Per attaccare il watermark, se l'attacco non è l'edge flip, basta anche aggiungere altri if che spezzino le sequenze. Con la ridondanza, l'attaccante doveva aggiungere il 50% di codice in più per scappare.

Quando l'attacco consiste nell'aggiungere altri if, posso risolvere con la ridondanza. In questo modo, l'attaccante deve aggiungere almeno il 50% del codice in più (dato che era molto ridondata).



Insomma, ci portiamo a casa che se anche ammettessi che l'attaccante è in grado di compromettere parte dell'info che inserisco, posso inserire tante volte la cosa.

Abstract watermarking

An abstract interpretation-based framework for software watermarking, Cousot and Cousot. POPL 2004

Sta in mezzo fra statico e dinamico: esegue il programma su dominio astratto, ovvero il watermark è un'invariante su un dominio astratto (basically è dinamico ma non su un solo valore, su un insieme!)

- Statico, perché per estrarre il valore non serve eseguire il programma (?)
- Dinamico, perché il valore è nella semantica

La chiave è il dominio astratto corretto. Il watermark è un'invariante che si presenta solo su quel dominio astratto.

Codifica

L'idea è di includere un "grande" numero nel programma. Per farlo posso usare il teorema cinese del reto, spezzettando una variabile grande in tante variabili piccole, e poi:

Supponiamo che c_i sia la chiave che vogliamo inserire.

- Introduciamo una variabile W che possiamo usare
- Inizializziamo W a $P(1)$, dove P è un polinomio che quando viene valutato a 1 ritorna c_i in $\frac{Z}{n_i Z}$, ovvero in modulo n valuta sempre a c_i
- W viene sempre aggiornata durante l'esecuzione, in $Q(W)$, che aggiorna la variabile in modo che quello che ottengo è sempre c_i .

Ergo, la semantica concreta vedrà che il valore viene sempre aggiornato a una certo valore; la semantica astratta invece vede che il valore rimane costante in $\frac{Z}{n_i Z}$.

Dopo di che, per andare a nascondere la cosa:

- $P(x)$ è un polinomio di secondo grado, in cui:
 - I coefficienti dipendono da c_i ; per far valere la cosa che valutano a 1 scegliamo $P(x) = x^2 + k_1 * x + k_0$ con $k_1 = -(1 + c_i)$ e $k_0 = 2 * c_i$
 - Per nascondere meglio c_i i coefficienti k_0 e k_1 sono aumentati in momenti randomi di un certo modulo n_i , per cui ho $k_1 = -(1 + c_i) + r * n_i$ e $k_0 = 2 * c_i + r_o * n_i$
- Se W è una variabile morta, non la dichiariamo ;)
- Il polinomio Q viene costruito similmente, come $Q(x) = ax^2 + b * x + c$, con a e b numeri randomici non troppo grandi e c scelto ad hoc affinché $c_i = Q(c_i)$ in $\frac{Z}{n_i Z}$

Infine, inserisco i polinomi nel codice – posso farlo col metodo di Horn.

Estrazione

Alla base dell'estrazione ho che per estrarre i c_i devo sapere il "segreto" – ovvero che vanno guardati in modulo n – perché altrimenti non vedo che sono costanti, ma vedo dei valori che cambiano.

Ad ogni esecuzione, dunque, analizzo P_w e guardo che è sempre costante a meno della costante.

Valutazione

- **Data-rate alto**
- **Non è stealthy**: è strano avere questi numeri grandi
- **Resilienza: difficile da individuare ma facile da rimuovere**: devi capire che quei numeri sono inutili. Però facendo il program slicing mi accorgo facilmente che non influiscono l'output e li posso eliminare. La risposta è che posso usare le tecniche di offuscamento per legare fittiziamente queste variabili.

Esempio

```
int f = -158657; initialization
for (...)
```

W = 21349
key = 3001

I valori di f in modulo sono sempre lo stesso numero.

Esempio IRL della prof: teorema cinese del resto

Loro, ad esempio, hanno sviluppato per un'azienda un algoritmo di watermarking fatto sul teorema cinese del resto: presi tutti numeri primi, codifico numeri (=firme) che sono più piccoli del prodotto dei numeri primi scelti. La proprietà fondamentale è che poi prendo il numero che voglio codificare e mi calcolo il resto (il modulo) della firma con ogni numero primo. Se i numeri primi sono n, ho una sequenza di n resti.

Il teorema cinese del resto dice che se hai i resti e i numeri primi puoi ricostruire la firma modulo il prodotto. Avresti infinite soluzioni a quel sistema di equazioni (perché sono tutte soluzioni in modulo n). Loro usavano i primi come chiave, e inizializzavano alcune variabili in modo tale che solo su un input segreto si inizializzavano generando i numeri giusti per estrarre il watermark.

Più in generale, si può aggiungere qualunque truccetto che si conosce sui numeri e inserirlo su variabili esistenti (o nuove, purché siano stealthy) e legarne il valore all'input corretto. Nel loro caso, le variabili erano sempre inizializzate ma il sistema di equazioni dava la soluzione giusta solo col giusto input.

arie cose che non dipendono davvero dagli input, ma creano cose IN MODULO.

Birthmarking

Viene eliminata la fase di embedding, e c'è solo la fase di estrazione. L'idea è che nel codice c'è qualcosa che rappresenta ogni altra versione che posso creare di quel codice.

Quindi, voglio capire cosa "caratterizza" in modo univoco il mio programma: un algoritmo, una struttura dati, un certo modo di far qualcosa, delle variabili... È qualcosa che se è presente in un altro programma allora quel secondo programma deriva dal mio.

Manca la parte di embedding: il birthmark non è inserito dall'esterno, ma è contenuto al suo interno. Sono delle proprietà: non possono essere totalmente sintattiche (es. "chiamo tutte le variabili mila1mila2mila3), ma guardano cose più strutturali sull'idea di come è stato risolto il problema.

Questi disegnini sotto non c'entrano niente ma ammirate quanto fa schifo la grafia con l'over-smoothing del Surface.
NON COMPRATE UN SURFACE ciao

