Digital Whisper Technologic Papers

Speculative Type Confusion

מאת אופק קירזנר ואדם מוריסון

הקדמה

מאז ש-Spectre ו-Meltdown, חלוצות תקיפות ההרצה הספקולטיבית, התפרסמו בראשית שנת 2018, החל עיסוק נרחב בוקטור הספקולטיבי, הגנתי (מיטיגציה) והתקפי (חולשות חדשות). תקיפת הרצה ספקולטיבית מנצלת side effects הנגרמים כתוצאה מספקולציה שגויה של המעבד לטובת הדלפת מידע (info leak) מהתוכנית המותקפת.

מתקפות Spectre v1, המנצלות מיס-ספקולציה של פקודות conditional branch, מוצגות לרוב בעזרת דוגמה של עקיפת bounds check של גישה למערך בעזרת הרצה ספקולטיבית. אלא שמסתבר שגורמים רבים תפסו את Spectre כאילו היא מוגבלת רק לעקיפה של bounds check. כך, למשל, ההגנות שיושמו ב-Linux kernel מגנות רק על גישות למערכים.

אולם באופן כללי, תקיפת Spectre v1 יכולה להשתמש בכל סוג של מיס-ספקולציה של conditional branch שגורמת לנתקף להריץ קוד באופן שגוי. במאמר זה נסקור את תוצאות המחקר המשותף שלנו, שבו חקרנו את הפגיעות של Linux לוקטור אחר של Spectre v1 בשם *speculative type confusion*, שבו המיס-ספקולציה של conditional branch מנוצלת על מנת לגרום לנתקף להריץ קוד עם משתנים שתכנם מ-type שגוי וע"י כך לגרום לדליפת מידע. מצאנו שניתן לנצל את מנגנון ה-eBPF של Linux כדי לייצר speculative type confusion באופן פעיל; שהקומפיילר יכול ליצור חולשות כאלו כתוצאה מאופטימיזציות תמימות; וכן שהן באות לידי ביטוי במנגנון הירושה ב-kernel.

המאמר המקורי התפרסם ב-USENIX Security Symposium באוגוסט 2021.

רקע: Spectre v1

הרצה ספקולטיבית

בשביל להבין מה היא Spectre, צריך להבין לפני כן מה היא הרצה ספקולטיבית, מנגנון המאפשר למעבד שיפור ביצועים משמעותי. המעבד קורא פקודות מהזיכרון בקצב מאוד גבוה וכאשר הוא נתקל ב-branch, הוא צריך לחשב אותו על מנת לדעת איך להתקדם לפקודה הבאה. על מנת לא לעצור את ה-frontend (הרכיב במעבד שקורא את הפקודות מהזיכרון) בזמן חישוב ה-branch, ולאפשר לו התקדמות מתמדת, ה-frontend מבצע פרדיקציה (חיזוי) של תוצאת ה-branch, וממשיך לקרוא להריץ פקודות בצורה ספקולטיבית ע"ס חיזוי זה. לאחר סיום חישוב תוצאת ה-branch, אם מתברר שהחיזוי היה מוטעה, המעבד מבצע rollback לפקודות ב-branch השגוי ולכל הפקודות שהורצו אחריו, ולאחר מכן מבצע את ה-branch הנכון וממשיך לקרוא ולהריץ את הפקודות במסלול הנכון של התוכנית.

נדגיש, כי הספקולציה היא אופטימיזציה שנועדה לשיפור ביצועים. ברמה האבסטרקטית, המתכנת לא חשוף לפרטי מימוש אלו. החולשה ב-Spectre נוצרה מכיוון שבפועל, ה-rollback של פקודות שבוצעו לאחר חיזוי לא נכון אינו מושלם, ופקודות כאלו משאירות אחריהן "שרידים" בצורת side effects מיקרו-ארכיטקטוניים שניתן לזהותם, כפי שנראה מיד.

Array Bounds Check Bypass

בקטע הקוד הבא:

```
if (x < array1_len) { // branch mispredict: taken
  y = array1[x];      // read out of bounds
  z = array2[y * 4096]; } // leak y over cache channel
```

מתבצעת בדיקה ש-x, ה-offset במערך 1array חוקי. במידה והוא חוקי, מתבצעת גישה למערך ולאחר מכן שימוש בערך שנקרא בתור index למערך נוסף. תחת הרצה מספר פעמים של הגאדג'ט הנ"ל עם ערכי x חוקיים, המעבד "לומד" שברוב המקרים ערך ה-predicate הוא true. הרצה של הגאדג'ט הזה כאשר array1_len לא נמצא ב-cache תפתח חלון ספקולציה, שבו המעבד ימשיך להריץ פקודות בצורה ספקולטיבית ע"ס חיזוי ה-branch. מכיוון שהמעבד "למד" שלרוב x חוקי, הוא יכנס לתוך ה-branch. כלומר, אם נבחר x=secret_ptr-array1, תתבצע גישה ספקולטיבית ל-secret - גם אם **x** לא חוקי. אם חלון הספקולציה "רחב" מספיק כדי שיתבצע בתוכו גם השימוש בערך secret בתור האינדקס ל-array2, הערך הזה יטען ל-cache והמעבד.

מאוחר יותר, כאשר המעבד יבין שהחיזוי היה מוטעה ויעשה rollback לספקולציה השגויה, הוא לא ינקה את הערך מה-cache. כלומר, ההרצה הספקולטיבית תשאיר שאריות מיקרו-ארכיטיקטוניות שניתן להבחין בהן באמצעות cache side channel.

המאמר המקורי של Spectre מ-2018 הגדיר את Spectre באופן כללי ומדויק ככל ניצול של conditional branch misprediction, כאשר ההתקפה המתוארת מעלה הובאה שם כדוגמה קונקרטית לתקיפה. למרות הגדרה כללית זו, החולשה הובנה על ידי רבים בקהילת האבטחה באופן חלקי, כאילו היא מוגבלת רק ל-bounds check bypass. ולראיה, המספר רב של מחקרים ושיטות המציעים הגנות מפני Spectre v1 שפורסמו ומגנים רק מפני סוג תקיפה כזו. אלא שבפועל, ישנם גאדג'טי קוד מסוגים אחרים מאשר המתואר מעלה אשר מהווים חולשת Spectre - למשל speculative type confusion, כפי שיוסבר בחלק הבא.

Speculative Type Confusion

הפונקציה הבאה, הכתובה בשפת C, משתמשת במנגנון ירושה. הארגומנט הוא מצביע ל-Base, והפונקציה מסיקה את סוג האובייקט "החבוי" בזמן ריצה, על סמך הערך בשדה type. אם תוקף גורם להרצת הפונקציה עם אובייקט שאינו מסוג TYPE1, בעוד ה-branch הראשון אומן לחזות TYPE1, הגאדג'ט ידליף ערך כלשהו ברוחב 32 ביט (באופן פוטנציאלי ערך סודי), השמור באובייקט שעבר בפועל.

```
void syscall_helper(struct Base* obj) {
    if (obj->type == TYPE1) {
        struct Type1* o = (struct Type1*) obj;
        leak(o->value);
    }
    if (obj->type == TYPE2) {
        ...
    }
}

struct Base {
    enum Type type;
};

struct Type1 {
    struct Base base;
    ...
    uint32_t value;
};
```

extended Berkeley Packet Filter, הוא מנגנון ב-Linux kernel המאפשר למשתמש בהרשאות נמוכות להריץ לוגיקות כחלק מהקרנל, למשל לצורך עיבוד פקטות או ניטור קריאות מערכת. על מנת לאפשר זאת, המנגנון נשען על מודל אבטחה שחושף לקוד המשתמש יכולות מוגדרות היטב, למשל גישה לאזור מוגבלים בזיכרון או שימוש בסט מוגבל של פונקציות. המנגנון מוודא זאת בזמן טעינת הקוד באמצעות static verification. ה-verifier מסמלץ את כל ה-control flow האפשריים בקוד שנטען, ומוודא שהם מקיימים תכונות מסוימות ולא מפרים תכונות אחרות.

נכון לזמן המחקר שלנו, התברר כי ה-verifier לא סימלץ speculative flows ועל כך נשענת החולשה שמצאנו (ואשר נסגרה לאחר הדיווח שלנו). בפרט, הצלחנו לטעון את הגאדג'ט הבא:

```
// r0 = ptr to a map array entry (verified ≠ NULL)
// r6 = ptr to stack slot (verified ≠ NULL)
// r9 = scalar value controlled by attacker
```

```
  r0 = *(u64 *)(r0) // miss
A:if r0 != 0x0 goto B
  r6 = r9
B:if r0 != 0x1 goto D
  r9 = *(u8 *)(r6)
C:r1 = M[(r9&1)*512];//leak
D:...
```

מדוע קטע הקוד הזה יכול לשמש כגאדג'ט של Spectre v1? מדובר בחלק מקוד eBPF שעובר verification. ב-flow זה בריצה, ברגיסטר 0r יש כתובת חוקית למערך, ברגיסטר 6r יש כתובת חוקית למחסנית וברגיסטר 9r יש סקלר בעל ערך שרירותי הנשלט על ידי התוקף. אם מצליחים לאמן את שני ה-branches, הראשון (A) לחזות 0r==0, והשני (B) לחזות 0r==1, ה-flow מתייחס לסקלר בשליטת התוקף ככתובת בזיכרון, ובכך מתקבל גאדג'ט כפי שהדגמנו מקודם המאפשר קריאת זיכרון שרירותי ממרחב הכתובות של ה-kernel.

נשים לב כי נותרו מספר אתגרים משמעותיים, אשר אותם אנו פותרים במאמר:

- אימון שני ה-branches - מכיוון שהם mutual exclusive, ריצה (דטרמיניסטית) של אחד תפגע באימון של השני, ולכן יש לאמן אותם בצורה מיוחדת.
- התמודדות עם העובדה שה-covert channel "משדר" את הערך הסודי בזמן ריצת ה-kernel, אבל התוקף שרוצה "לקלוט" את הערך מוגבל לריצה ב-user space.

בחלק הקודם הראנו כיצד ניתן לייצר באופן מלאכותי גאדג'ט המאפשר speculative type confusion, אבל האם ניתן למצוא כאלו מבלי להנדס אותם? מתברר כי קיימת אפשרות שהקומפיילר ייצר גאדג'ט מסוג זה. כמוטיבציה, נביט בדוגמה הבאה:

```
volatile char A[256*512];
bool predicate;
char* foo;

void victim(char *p,
            uint64_t x,
            char *q) {
    unsigned char v;

    if (predicate) {
        p = (char *) x;
        *q |= 1;
    }
    if (!predicate) {
        v = A[(*p) * 512];
    }
    foo = p;
}
```

(a) C code

```
# args: p in %rdi
#       x in %rsi
#       q in %rdx

   # first "if":
   cmpb   $0x0,(predicate)
B1:je     L1 # skip 2nd if
   # assignment to p:
   mov    %rsi,%rdi
   # assignment to q:
   orb    $0x1,(%rdx)
   # second "if":
   cmpb   $0x0,(predicate)
B2:jne    L2
   # deref p & leak
L1:movsbl (%rdi),%eax
   shl    $0x9,%eax
   cltq
   movzbl A(%rax),%eax
L2:mov    %rdi,(foo)
   retq
```

(b) Emitted x86 code.

משמאל, ניתן לראות קטע קוד ב-C ומימין תוצר הקומפילציה שלו ע"י gcc (גרסה 4.8.2 עם דגלון O1). בדוגמה זו, המשתנה x נשלט על ידי התוקף. נשים לב, שהקומפיילר דורס את רגיסטר rsi (הארגומנט p) עם התוכן של rdi (המשתנה x), מכיוון שהדבר מאפשר לו להשתמש באותה פקודה השמה ל-foo בסוף הפונקציה. בנוסף, הקומפיילר חושש כי השמה ל-q עשויה לשנות את predicate (אם q מצביע ל-predicate), ולכן מכניס בדיקה נוספת ל-predicate ב-if block השני. תוקף המאמן את שני ה-branches כך ששניהם יחזו "not taken" יוכל לקבל Spectre gadget כמתואר למעלה.

איך למצוא חולשות כאלו?

על מנת למצוא חולשות פוטנציאליות מהסוג הזה, ביצענו static analysis בבינארי של לינוקס (5.4.11). מכיוון שאנליזה הרמטית היא משימה בלתי אפשרית (בזמן פולינומיאלי), ביצענו כמה רלקסציות. תחילה, בחרנו להתמקד במציאת גאדג'ט כזה מבלי לעסוק בשאלות האם וכיצד ניתן להשמישו (ישנן הרבה דרכים להשמשה, שינוי setup עשוי להשפיע על היכולת להשמיש גאדג'ט, שינוי עתידי עשוי להפוך גאדג'ט לשמיש).

השתמשנו ב-radare2 על מנת לבצע static analysis וב-triton ל-taint tracking ו-symbolic execution. Radare2 הוא כלי הנדסה לאחור, המאפשר טעינה ואנליזה של קוד (בפרט x86). בחרנו להשתמש בו על מנת לטעון לזיכרון את ה-kernel של לינוקס ולחלץ את עץ הקריאות של הפונקציות שניתחנו.

Triton מאפשר taint analysis והרצה סימבולית, כלומר - בהינתן קטע קוד, "הכתמה" של משתנה והרצת הקוד באופן סימבולי על מנת לבדוק כיצד הכתם מפעפע למשתנים אחרים (ומכך ניתן להסיק ערכים פוטנציאליים של משתנים).

האנליזה מנתחת קריאות מערכת (syscall), משום שלהן יש ממשק מוגדר היטב וניתן לסמן את הארגומנטים כברי שליטה על ידי התוקף. בהסתכלות על כל execution path אפשרי (תחת הרצה ספקולטיבית, ומכאן שכל כיוון של branch מובא בחשבון) של קריאת מערכת, אנו מחפשים פקודה הניגשת לזיכרון מכתובת לא בטוחה (נשלטת על ידי המשתמש) בנתיב אחד, וגם ניגשת לכתובת בטוחה (נשלטת על ידי ה-kernel) בנתיב אחר.

שילוב כזה, מעיד על כך שהפקודה משמשת לטעינה מהזיכרון באופן נומינאלי (ה-kernel ניגש אליו באופן דטרמיניסטי) אבל שניתן לייצר הרצה ספקולטיבית שבה הטעינה מתבצעת ממשתנה בשליטת התוקף.

יש לציין, כי האנליזה אינה שלמה (complete, מוצאת את כל הפקודות הרלוונטיות) וגם אינה נאותה (sound, כל פקודה שנמצאת אכן יכולה לשמש לטעינה ספקולטיבית מכתובת זיכרון בשליטת התוקף), בעקבות מגבלות חישוב והיכולת לסמלץ את זיכרון ה-kernel ואת הפונקציות שבו. לכן, ביצענו וידוא ידני של התוצאות.

תוצאות

אחת התוצאות המעניינות שמצאנו, היא דפוס הנובע מאופטימיזציית מקום, כפי שמודגם בקטע הקוד הבא:

```
syscall(foo_t* uptr) {
  foo_t kfoo;
  // some code
  if (uptr)
    copy_from_user(&kfoo,
                   uptr,
                   ...);
  f(uptr ? &kfoo : NULL);
  // rest of code
}
```

(a) C pattern

```
# args: uptr in %rdi
  ...
  testq %rdi, %rdi
  je L # jump if %rdi==0
  # set copy_from_user args
  ...
  # %rbp contains addr of
  # stack buffer
  mov %rbp, %rdi
  call copy_from_user
L:callq f
```

(b) Emitted x86 code.

Listing 7: Reusing registers for a function call.

בקריאת מערכת זו, ישנו ארגומנט אופציונלי (יכול להיות NULL) שמצביע לזיכרון במרחב הכתובת של ה-user. בתחילת הפונקציה ישנה בדיקה אם המשתמש התכוון להשתמש בארגומנט זה (כלומר, סיפק ערך שאינו NULL). אם הוא NULL, מעבירים את הארגומנט כפי שהוא (רגיסטר rdi) להמשך הקוד. במידה והוא אותחל, הוא מועתק לזיכרון kernel, ודורסים את rdi בערך של המצביע החדש. כלומר, על מנת לחסוך פקודות, נעשה שימוש מחדש ב-rdi.

אימון של ה-branch הזה שלא לדרוס את rdi (כלומר, לחזות שהערך הוא NULL) ולאחריו תקיפה עם ערך שאינו NULL עשויות לגרום לזליגת מידע מה-kernel, משום שהפונקציה f שנקראת תיקרא מכתובת בזיכרון שהתוקף שולט בה. נשים לב, כי ה-branch המדובר משווה ערך שנמצא ברגיסטר, ועל פניו הדבר מקשה לייצר חלון ספוקלציה מספק למתקפה.

בפועל, נראה כי ההתנהגות של המעבד תלויה ב-branches קודמים ב-pipeline ותחת תנאים מסוימים אפשר להגדיל את חלון הספקולציה. פרטים נוספים במאמר.

ירושה ב-Linux Kernel

כמעורר תיאבון אחרון, נצרף את מנגנוני הירושה בקרנל של לינוקס:

```
struct Common {
 void* private;
};
struct Derived {...};

void foo(Common* c) {
 Derived* d = c->private;
```

(a) Private field.

```
struct Common {...}
struct Derived {
 struct Common common_data;
 ...
}
void foo(Common* c) {
 Derived* d = (Derived*) c;
```

(b) Casting.

```
struct Derived {
 ...
 struct Common common;
 ...
}
void foo(Common* c) {
 Derived* d = container_of(c, Derived*, common);
```

(c) Contained structs.

Listing 12: Linux data inheritance patterns.

מכיוון שקוד ה-kernel כתוב ב-C, והשפה לא תומכת בירושה בצורה אינהרנטית, מפתחי Linux הכניסו מספר מנגנונים למימוש המנגנון. המקבילה ל-class ב-C היא struct, והירושה ממומשת באחת משלוש דרכים:

a. באמצעות מצביע מאובייקט האב לאובייקט הבן, אשר מומר לאובייקט הבן כתלות בהקשר.
b. באמצעות הכלה של אובייקט האב בתחילת הכתובת בזיכרון שבה מוקצה אובייקט הבן, כך שמצביע לאובייקט הבן הוא גם מצביע לאובייקט שממנו הוא נורש.
c. באמצעות הכללה של מנגון (b), כאשר ההכלה של האב היא באיזשהו היסט באובייקט הבן, וחישוב של ההיסט באמצעות המאקרו container_of.

האם ניתן מתוך זה למצוא speculative type confusion gadgets? כלומר, האם ניתן למצוא זוג פונקציות, שהן מימוש של אותו ממשק (פונקציה וירטואלית) עבור שני אובייקטים שונים, המקיימים:

- בפונקציה הראשונה יש קריאה מזיכרון ומנגנון הדלפה שלו (side channel). הקריאה מהזיכרון היא מכתובת השמורה באובייקט.
- באובייקט של המימוש השני (overload), יש באותו offset של הכתובת ערך שנשלט על ידי התוקף.

במידה ומצאנו כזה זוג, האם אפשר יהיה להשמיש אותו - לאמן כך שספקולטיבית קוד של האובייקט הראשון יקרא עבור האובייקט השני? איך השתמשנו ב-smatch (תשתית static analysis ל-C) על מנת לסרוק את קוד ה-kernel ומה הממצאים שלנו - במאמר. [רמז: אפשר למצוא, קשה להשמיש].

מיטיגציה

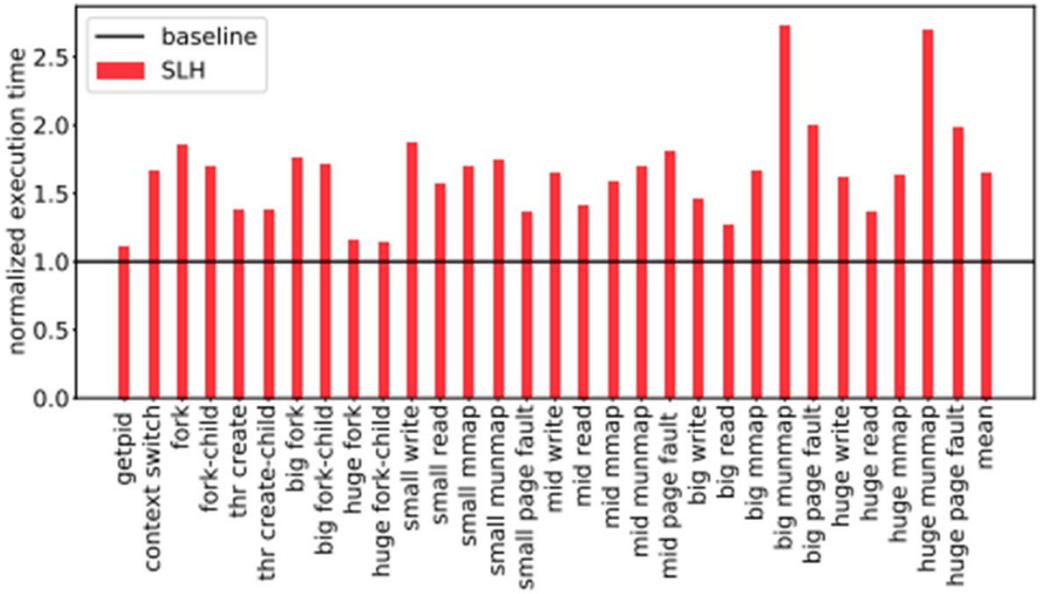
בשונה מ-array bounds check bypass, ל-speculative type confusion אין מבנה מוגדר היטב הניתן לזיהוי בקלות, על כן מיטיגציה ידנית היא בלתי אפשרית.

בבדיקה על הקומפיילרים הרלוונטים (gcc, llvm, msvc, icc), נראה שכולם מייצרים גאדג'ט פגיע כפי שהדגמנו מעלה. למעט gcc, כל הקומפיילרים הגדולים מציעים דגלון מיטיגציה אוטומטית נגד Spectre v1. ב-llvm הכניסו מיטיגציה בשם SLH, speculative load hardening, אשר חוסמת השמשה של גאדג'טים מסוג זה באופן הרמטי (ראה מסגרת). msvc (מיקרוסופט) ו-icc (אינטל) תומכים במיטיגציה מלאה (החוסמת גאדג'ט זה), אולם הם ממליצים להשתמש במיטיגציה המבוססת על זיהוי דפוסים פגיעים וחסימה כירורגית שלהם. מיטיגציה זו אינה מזהה גאדג'ט מהסוג המתואר מעלה, מה שמצביע על הבנה מוטעית של Spectre v1.

הגישה ב-gcc היא שמפתח יעשה שימוש ב-barrier למניעת ספקולציה במקומות שבהם ישנה סכנה, כלומר שהוא יזהה באופן ידני היכן ראוי להשתמש במיטיגציה. כאמור, זוהי גישה שלא מחזיקה מים, מכיוון שהגאדג'ט של Spectre v1 יכול להיווצר בזמן קומפילציה, מבלי שהמפתח מודע לכך.

***Speculative Load Hardening** היא מיטיגציה המכניסה קוד ללא branches, על מנת לייצר data dependency בין כל טעינה מכתובת בזיכרון לכל ה-branch predicates שקדמו לה. הדבר, מונע טעינה מהזיכרון ב-flow ספקולטיבי ובכך מונע לחלוטין speculative type confusion.*

אולם, יש לכך השלכה חמורה על הביצועים, כפי שניתן לראות בגרף הבא:



Slowdown of Linux 5.4.119 kernel operations due to SLH.

סיכום

הראינו כיצד וקטור נוסף של Spectre v1 בא לידי ביטוי בקרנל של לינוקס, פעם אחת כתוצאה מהנדסה של תוקף, פעם שנייה כתוצר קומפילציה ופעם שלישית כתוצאה משימוש נומינלי במנגנוני הירושה ב-Linux. לגאדג'ט הרלוונטי אין מבנה מוגדר היטב והוא יכול להיווצר כתוצאה משינויי קוד אקראיים ואופטימיזציות של הקומפיילר. הדבר הופך גישת מיטיגציה המזהה דפוסים פגיעים או גישה ידנית לבלתי אפשריות, ומצריך מחקר נוסף באשר לפתרון שיספק הגנה תוך פגיעה מינימלית בביצועים. בפרט, יש לבחון את הגישה הקיימת ב-Linux kernel וייתכן שגם במערכות אחרות.

על הכותבים

אופק קירזנר סיים תואר שני במדעי המחשב בהנחיית אדם מוריסון, באוניברסיטת תל אביב. כיום הוא מוביל את תחום התשתיות הקוונטיות בחברת Classiq, העוסקת בתוכנה למחשבים קוונטיים.

אדם מוריסון הוא חבר סגל בביה"ס למדעי המחשב באוניברסיטת תל אביב. נושאי המחקר שלו הם אבטחה וביצועים של מערכות מחשב, מרמת המיקרו-ארכיטקטורה דרך מערכת ההפעלה ועד לאלגוריתמים. עבודותיו זכו בפרס Intel Hardware Security Academic Award, מספר פרסי Best Paper, וכן נבחרו כ-Top Picks ו-Top Picks Honorable Mention ע"י המגזין IEEE Micro.