УНИВЕРЗИТЕТ У НОВОМ САДУ

ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ

Марко Ђуковић

АНАЛИЗА АЛАТА ЗА ПРОМЕНУ РАЗУМЉИВОСТИ ПРОГРАМА НА БАЗИ ЕНЕРГЕТСКЕ ЕФИКАСНОСТИ ИЗВРШАВАЊА

ДОКТОРСКА ДИСЕРТАЦИЈА

Нови Сад, 2019.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:		
Идентификациони број, **ИБР**:		
Тип документације, **ТД**:		Монографска публикација
Тип записа, **ТЗ**:		Текстуални штампани материјал
Врста рада, **ВР**:		Докторска дисертација
Аутор, **АУ**:		Марко Ђуковић
Ментор, **МН**:		Ван. проф. др Ервин Варга
Наслов рада, **НР**:		Анализа алата за промену разумљивости програма на бази енергетске ефикасности извршавања
Језик публикације, **ЈП**:		Српски
Језик извода, **ЈИ**:		Српски
Земља публиковања, **ЗП**:		Србија
Уже географско подручје, **УГП**:		АП Војводина, Нови Сад
Година, **ГО**:		2019.
Издавач, **ИЗ**:		Ауторски репринт
Место и адреса, **МА**:		ФТН, Трг Доситеја Обрадовића 6, 21000 Нови Сад
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)		7/104/139/14/74/0/3
Научна област, **НО**:		Електротехничко и рачунарско инжењерство
Научна дисциплина, **НД**:		Електроенергетика
Предметна одредница/Кључне речи, **ПО**:		Обфускација (маскирање) кода / Заштита кода/ Утицај на енергетску ефикасност
УДК		
Чува се, **ЧУ**:		Библиотека ФТН-а, Трг Доситеја Обрадовића 6, 21000 Нови Сад
Важна напомена, **ВН**:		
Извод, **ИЗ**:		У овој докторској дисертацији анализиран је утицај једне од техника заштите софтвера, позната као маскирање (енг. *obfuscation*), на енергетску ефикасност извршавања кода. Циљ рада је да проучи колико овакви захвати утичу на промену профила потрошње електричне енергије, односно рангирање алата за промену разумљивости програма на основу енергетског профила за чије генерисање је развијена програмска подршка. Тестирање је реализовано коришћењем различитих комерцијалних алата над релевантним тест сценаријима и резултати су приказани уз одговарајућу анализу.
Датум прихватања теме, **ДП**:		
Датум одбране, **ДО**:		
Чланови комисије, **КО**:	Председник:	Ванр. проф. др Дарко Чапко
	Члан:	Ванр. проф. др Срђан Вукмировић
	Члан:	Доц. др Милан Гаврић
	Члан:	Ред. проф. др Љубомир Лазић
	Члан, ментор:	Ванр. проф. др Ервин Варга

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed document
Contents code, **CC**:	PhD thesis
Author, **AU**:	Marko Đuković
Mentor, **MN**:	Associate Professor PhD Ervin Varga
Title, **TI**:	The analysis of the tools for program intelligibility variability conditioned by energy efficiency of execution
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian
Country of publication, **CP**:	Serbia
Locality of publication, **LP**:	Autonomous Province of Vojvodina, Novi Sad
Publication year, **PY**:	2019.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	7/104/139/14/74/0/3
Scientific field, **SF**:	Electrical and Computer Engineering
Scientific discipline, **SD**:	Power Engineering
Subject/Key words, **S/KW**:	Code obfuscation/ Code protection/ Impact on energy efficiency
UC	
Holding data, **HD**:	Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, 21000 Novi Sad
Note, **N**:	
Abstract, **AB**:	This doctoral dissertation analyze the influence of one of the software protection techniques known as obfuscation, to the power efficiency of code obfuscation. The aim of the dissertation is to study the effect of these techniques on the change of power profile consumption, i.e., ranking of tools for changing the program intelligibility based on energy profile for the generation of witch a program support has been developed. The testing is realized by using various commercial software for relevant test scenarios and the results are presented with the corresponding analysis.
Accepted by the Scientific Board on, **ASB**:	
Defended on, **DE**:	
Defended Board, **DB**: President:	Associate Professor PhD Darko Čapko
Member:	Associate Professor PhD Srđan Vukmirović
Member:	Assistant Professor PhD Milan Gavrić
Member:	Full Professor PhD Ljubomir Lazić
Member, Mentor:	Associate Professor PhD Ervin Varga

Menthor's sign

Захвалница

*Пре свих, огромну захвалност дугујем својим родитељима, оцу **Миомиру** и мајци **Милени** који су најзаслужнији за све што сам до сада постигао.*

*Такође, изузетну захвалност дугујем свом брату **Мирку**, на безрезервној подршци и мотивацији која никада није изостала.*

*Велико хвала професорима који су несебичним дељењем свог широког знања поставили чврсте темеље за даљи рад и истраживање у области софтверског инжењерства, **проф. Дарку Чапку**, **проф. Срђану Вукмировићу**, **проф. Љубомиру Лазићу** и **проф. Милану Гавришу**.*

*Захвалио бих се свим колегама и пријатељима на подршци и помоћи приликом истраживања и израде дисертације а посебно: **Милошу Гајићу, Невену Ковачком** и **Сави Ђукићу**.*

*Посебно желим да се захвалим **Дини Тењи** на бескрајном разумевању, подршци у тешким моментима и подстреку током израде ове дисертације.*

*Посебну захвалност дугујем свом ментору, **проф. Ервину Варги**, на неисцрпном стрпљењу, саветима, мотивацији и експертизи без које представљени истраживачки рад и докторска дисертација не би били могући.*

Резиме

У овој докторској дисертацији анализиран је утицај једне од техника заштите софтвера, позната као маскирање (енг. *obfuscation*, у даљем тексту обфускација), на енергетску ефикасност софтверских апликација. Циљ рада је да проучи колико овакви захвати утичу на промену профила потрошње електричне енергије, односно рангирање алата за промену разумљивости програма.

Софтверски инжењеринг постаје све развијенија индустријска грана а самим тим и технике заштите добијају све више значаја, како због огромних финансијских губитака, тако и због крађе интелектуалне својине. Обфускација представља технику заштите која претвара оригинални програм, у други, семантички еквивалент, знатно тежи за разумевање и инверзни инжењеринг. Алати који омогућавају овај вид заштите називају се обфускаторима, односно, компајлерима специјалних намена.

Потреба да се заштите права интелектуалне својине, као и потреба да се спречи злоупотреба кода који ради на свим врстама аутономних уређаја повећава потражњу за висококвалитетним алатима за промену разумљивости програма, тј. обфускаторима. Тренутно, главна метрика за процену квалитета обфускованог кода заснива се на аспектима као што су безбедност и функционална исправност. Ипак, потрошња електричне енергије игра централну улогу у преносним уређајима, јер је трајање батерија и издржљивост обично уско грло у постизању прихватљивог нивоа задовољства корисника системом. Сходно томе, критеријуме за избор одговарајућег обфускатора кода треба проширити на тај начин да се у обзир узме и димензија енергетске ефикасности.

У овој докторској дисертацији предложен је нови начин за процену квалитета обфускатора кода са аспекта потрошње електричне енергије, у чију сврху је развијена софтверска подршка за генерисање профила потрошње као основног критеријума за даљу анализу утицаја различитих техника обфускације:

- лексичка обфускација,
- обфускација тока извршавања,
- обфускација података.

Анализа утицаја реализована је над адекватним тестним сценаријима различите комплексности, коришћењем различитих комерцијалних обфускатора, док су у дисертацији приказани резултати за следеће комерцијалне алате:

- *Confuser*
- *Agile.NET*
- *Smart Assembly*

Abstract

This doctoral dissertation analyze the influence of one of the software protection techniques known as obfuscation, to the power efficiency of code obfuscation. The aim of the dissertation is to study the effect of these techniques on the change of power profile consumption, i.e., ranking of tools for changing the program intelligibility based on energy profile.

Software engineering is becoming an increasingly important concern in the IT sector, and therefore the software protection techniques are gaining momentum, not only due to major financial losses, but also because of intellectual property theft. Obfuscation is a protection technique which converts the original program into another, semantic equivalent, more complex for understanding and for reverse engineering. The tools which implement these techniques are called obfuscators, i.e, special-purpose compiler.

The need to protect intellectual property rights, and the need to prevent the abuse of code which operates on all types of autonomous devices increases the demand for high-quality tools for changing the software intelligibility, i.e, obfuscators. Currently, the main metrics for the obfuscated code quality assessment is based on aspects like security and functionality. Nevertheless, the power consumption plays a crucial role in mobile devices, since battery consumption and durability are usually the bottleneck in achieving an acceptable level of system usage. To that effect, the criteria in choosing the appropriate obfuscator need to be extended as to take into account power efficiency.

This doctoral dissertation presents a new way for obfuscator quality assessment from the power consumption aspect, for the purpose of which a software support for the consumption profile generation has been developed. The testing has been realized for the following obfuscation techniques:

- lexical,
- control flow obfuscation,
- data obfuscation,

Analysis of impact has been carried out on adequate test scenarios of different complexity, using different commercial obfuscators, while the dissertation presents the results for the following ones:

- *Confuser*
- *Agile.NET*
- *Smart Assembly*

Садржај

Списак слика

Списак табела

Списак скраћеница

ICT	*Information Communication Technologies* (информационо комуникационе технологије)
IT	*Internet Technologies* (интернет технологије)
API	*Application Programming Interface*
GoF	*Gang of Four*
SSL	*Secure Sockets Layer*
FPGA	*Field Programmable Gate Array*
OP	*Opaque Predicate*
NOP	*No Operation*
USB	*Universal Serial Bus*
CLR	*Common Language Runtime*
CIL	*Common Intermediate Language*
MSIL	*Microsoft Intermediate Language*
CPU	*Central Processor Unit*
HTML	*Hypertext Markup Language*
ABD	*Abnormal Battery Drain* (абнормално пражњење батерије)
JVM	*Java Virtual Machine* (јава виртуелна машина)
SII	*Software Inverse Engineering* (инверзни инжењеринг софтвера)

1. Увод

У овој глави представљен је утицај технике маскирања (у даљем тексту обфускације), као најзаступљенијег облика заштите софтвера од крађе интелектуалне својине, на енергетску ефикасност и дат је преглед стања у овој области. Након тога, објашњена је потреба, односно мотив за овим истраживањем и представљени су његови основни циљеви, тј. анализирани су алати за промену разумљивости програма. На крају, описана је организација ове докторске дисертације.

1.1 Предмет истраживања

Изузетно брз развој у области информационо комуникационих технологија (*ICT*) подразумева и додатне нивое зависности од потрошње електричне енергије. Ова чињеница је посебно битна у области развоја софтверских решења за мобилне уређаје. Споменути тренд неминовно прати већа потрошња електричне енергије што због повећања серверских постројења, тако и због нових центара података [1].

IT (енг. *Information Technologies*) сектор данас користи око 8% укупне производње електричне енергије на глобалном нивоу, са тенденцијом раста [2]. Трећину потрошње чине центри података (енг. *data centers*), трећину чине комуникационе мреже док последњу трећину чини корисничка опрема [3]. Потрошња електричне енергија је одувек била предмет забринутости у одређеним подручијима *IT*-а, као што су сензорске мреже и батеријски напајани уређаји. Међутим, за друге системе, као што су *data centers*, ово постаје област од интереса тек од скоро [4]. На пример, 2012. године, сваки дан, размењено је 145 милијарди *email*-ова, извршено 4.5 милијарди претрага на *Google* сервисима, подигнуто (енг. *uploaded*) 104 000 сати видео материјала на *Youtube* сервис и произведено 40 000 гигабајта података на *Large Hardon Collider*-u [5]. *Data centers* компаније *Google* (њих 900 000) су 2010. године потрошили 1.9 милијарди киловат часова [kWh] [6], док су 2012. године сви *data centers* широм света потрошили електричне енергије еквивалентно производњи 30 нуклеарних електрана [7].

Један од главних изазова са којима се суочава савремено софтверско инжењерство је потрошња електричне енергије самих апликација током рада (извршавања), тј. њихова енергетска ефикасност. Сам термин енергетске ефикасности и уведене метрике у овом раду обрађују квалитете атрибута који се односе на одрживост (енг. *sustainability*), као што је број асемблерских инструкција, време извршавања програма и потрошња електричне енергије (детаљно објашњено у прилогу 1).

Редукција потрошње није битна само за преносне уређаје, већ и у глобалном смислу. Тренутно на тржишту постоје наменски компајлери за посебне софтверске пакете који су намењени очувању ниског нивоа потрошње електричне енергије док су још раније циљеви приликом развоја компајлера били везани за брзину рада и утрошак меморије. Сваки облик наведене оптимизације представља врло комплексан задатак с

обзиром да извршни код мора да постигне прихватљиве перформансе и да промишљено искористи расположиве рачунарске ресурсе. Ипак, без обзира на софистицираност компајлера, приликом генерисања најбољег бинарног репрезента са аспекта енергетске ефикасности, укупна потрошња је диктирана од стране самог изворног кода. Другим речима, немогуће је оставити компајлеру на решавање сва питања везана за потрошњу електричне енергије. Уколико алгоритми и њихова имплементација захтевају веће количине електричне енергије приликом извршавања, компајлеру се оставља врло мало простора за ублажавање овог проблема.

Како се апликације извршавају на различитим врстама уређаја, њихова изложеност проницљивим појединцима са малициозним намерама је потенцијално већа. Стога је уложен велики напор како би се софтверски пакети заштитили од инверзног инжењеринга и модификације, као и крађе интелектуалне својине. Један од видова заштите представља техника обфускације кода (енг. *obfuscation*). Основна идеја је веома једноставна и односи се на реорганизацију кода у циљу повећања неразумљивости приликом инверзног инжењеринга, уз истовремену гаранцију семантичке еквиваленције оригиналном коду, што представља императивни захтев.

Упркос свему, обфускација тренутно представља најпопуларнију технику заштите спречавања илегалног присвајања интелектуалне својине. Цена примене ове технике је нарушавање перформанси у погледу меморије, времена извршавања, величине извршне библиотеке а нарочито потрошње електричне енергије.

Ова докторска дисертација има за циљ да укаже на проблем повећања потрошње електричне енергије услед примене различитих техника обфускације кода. У том циљу, предложена је нова техника, заснована на профилима потрошње. Ово је изузетно важан проблем код мобилних уређаја због њихових лимитираних енергетских ресурса као и веома захтевних хардверских карактеристика. Један од доприноса ове дисертације је и потенцијални корак унапред у разумевању сврхе подизања сигурности кода по цену смањења енергетских ресурса. Део резултата истраживања ове докторске дисертације објављен је као научни рад [8].

1.2 Преглед стања у области

Анализа утицаја обфускованог кода на потрошњу електричне енергије је слабо истражена област и тренутно постоји мали број радова који се баве конкретно овом проблематиком. Постоје повезани радови који се баве потрошњом за специфичне случајеве, при чему се сама област утицаја на енергетску ефикасност може поделити на неколико целина:

- утицај рефакторинга кода [9, 10, 11],
- утицај дефинисања структура података [12],
- утицај дизајн образаца (енг. *design pattern*) [13, 14],
- утицај сервиса [15],
- утицај вишенитног кода (енг. *multithreaded code*) [16, 17],

- утицај алгоритама [18, 19, 20, 21].

Аутори у [10] истражују утицај рефакторинга кода на енергетску ефикасност у случају детекције одступања од образаца (енг. *anti-pattern*). Експерименти су спроведени над два пројекта отвореног кода *Informa* (http://informa.sourceforge.net) и *NekoHTML* (http://qualitascorpus.com), који садрже одређен број "*God classes*", односно класа које имају проширене одговорности, комплексне су и као такве представљају типичан пример анти патерна. Резултати су показали да примена рефакторинга повећава потрошњу електричне енергије у опсезима од 7 до 20 процената. Мерење је реализовано коришћењем уређаја *Voltcraft Energy Logger 4000*, који мери тренутну потрошњу електричне снаге сваке секунде, у ватима (енг. *watts* (W)). Преведно у мерене вредности, резултати показују да је потрошња снаге код пакета *Informa* повећан за 7.56W током 60s извршавања, док је код *NekoHTML* потрошња увећана за 20.1W током 22s извршавања.

Хунт, Санту и Цезе у [12] истражују утицај различитих облика приступа структурама података на енергетску ефикасност. Приступ заједничким структурама података у вишенитним апликацијама мора бити исправно синхронизован како би се обезбедила конзистентност и интегритет података. Управо споменута синхронизација представља чест проблем са перформансама апликација овог типа. Структуре података без закључавања (енг. *lock-free*), представљају алтернативу традиционалним методама синхронизације и као такве имају већи потенцијал не само са аспекта бољих перформанси и скалабилности него и у области боље енергетске ефикасности. Аутори елаборирају везу између перформанси и потрошње енергије различитих *lock-free* структура података и њихових традиционалних еквивалената. Резултати показују да *lock-free* структуре често имају боље перформансе и троше мање енергије него традиционални модели засновани на имплементацији закључавања приступа.

Бунс у [14] даје приказ утицаја коришћења софтверских дизајн образаца (енг. *design pattern*) [22, 23] приликом развоја софтвера за мобилне уређаје који се заснивају на Јава платформи (енг. *Java*), као и њихово међусобно поређење на основу енергетске ефикасности. У раду су анализирани следећи образци: *Façade, Abstract Factory, Observer, Decorator, Prototype, Template Method*. Мерење је реализовано коришћењем апликације *PowerTutor* за следеће моделе телефона: *NexusOne, Galaxy SII, Transformer*. На основу добијених резултата, аутори заккључују да примена наведених принципа развоја, негативно утиче на енергетску ефикасност. Повећање времена извршавања, се креће у опсегу од 0.3% - 132%.

Аутори у [24] приказују утицај примене *GoF* (енг. *Gang of Four*) софтверских дизајн образаца [25, 26] као и његових алтернатива на енергетску ефикасност. Комплекснија анализа утицаја дизајн образаца на развој апликација приказана је у раду [13].

Манотас, Сахин, Клаус, Полок и Винблад у [15] анализирају утицај различитих серверских станица на потрошњу електричне енергије веб апликација. Кроз контролисане емпиријске експерименте, откривено је да се потрошња самих апликација значајно разликује у зависности од сервера који се користи за обраду њихових захтева.

Осим тога, сами сервери су мање или више енергетски ефикасни у зависности од самих функција које се извршавају у оквиру апликација.

Пинтор, Кастор и Ли у [16] истражују утицај руковања нитима на потрошњу електричне енергије. Разматрана су три модела за управљање нитима у области конкурентног програмирања: а) експлицитно креирање нити, б) фиксно повезивање и ц) стандардно. Додатно, анализирани су следећи параметри за споменуте моделе: а) број нити, б) стратегија поделе задатака и ц) карактеристике обрађених података. Резултати показују да се потрошња углавном повећава са већим бројем нити, док се драстично смањује како се број активних нити приближава броју језгара централне процесорске јединице. Тренутна потрошња снаге за два различита тестна окужења, износила је 8.54W и 6.87W када се користила 1 нит (енг. *thread*). У случају коришћења 32 нити, ове вредности су износиле 88.05W и 14.27W. Сличном проблематиком се баве аутори у [17] где је акценат стављен на утицај примене сихронизационих шаблона на енергетску ефикасност.

Адел, Аурелион, Роман и Лионел у [18], представљају оквир (енг. *framework*), за праћење енергетске учинковитости апликација у реалном времену. Њихов приступ се заснива на "*OSleven - POWERAPI*" библиотеци која врши процену снаге потрошње процеса на основу различитих фактора као што су оптерећење мреже, централна процесорска јединица, утрошак меморије, итд. У циљу бољег појашњења "цурења" енергије у софтверским алатима, аутори, коришћењем ове библиотеке добијају могућност детаљније анализе утицаја избора програмских језика као и алгоритама на енергетску ефикасност.

Марк Андерсон у [27], презентује систем за детекцију злоћудног кода анализирајући шаблоне динамичке потрошње електричне енергије у току извршавања помоћу профила потрошње. Предложено решење је корисно у контролисаним срединама (рутери, прекидачка опрема, итд.) где је жељено понашање система унапред познато.

Тивари у [28] елаборира приступ базиран на мерењима за одређивање нивоа потрошње електричне енергије на нивоу асемблерске инструкције. Николаидис и Лаополус у [29] представљају методологију за естимацију потрошње наменских процесора у великим системима док рад [30] елаборира модел за естимацију потрошње заснован на методама константних параметара. Рад [31] даје опис потрошње на нивоу групе инструкција напредних рачунарских система.

Сахин у раду [32] анализира утицај различитих обфускационих техника заштите софтвера на енергетску ефикасност апликација, писаних за Андроид (енг. *Android*) платформу. У раду [9] је спроведена емпиријска студија утицаја рефакторинга на енергетску ефикасност. Тестирање је спроведено за 197 апликација над којима је имплементиран неки од 6 најчешће коришћених типова реорганизације кода. Резулати су показали да примена рефакторинга може значајно да повећа потрошњу електричне енергије самих апликација, и као такви отварају нову димензију а уједно и скрећу пажњу развојним инжењерима о термину енергетске ефикасности, тј. потрошње електричне

енергије саме апликације током извршавања. Више о овом типу се може пронаћи у раду [33].

Марион, Андреас и Јан у [34] дефинишу приступ и начин развоја софтвера за мобилне телефоне (Андроид платформу), са циљем смањења потрошње електричне енергије посматрано са апликативног нивоа. Овај приступ укључује дефинисање, откривање и реструктуирање енергетски неефикасних делова кода. Предложени начин има за циљ да скрене пажњу развојним инжењерима о утицају на енергетску ефикасност приликом развоја самих апликација. Идеја рада, настала је због чињенице да мобилни уређаји, као и мобилне мреже, у Немачкој годишње потроше 12.9 PJ (енг. *petajoules*), што износи око 3.5 TWh (енг. *terawatt hours*) терават часа, што је еквивалнт четворомесечној производњи нуклеарне електране Емсланд [35]. Уштеда енергије је потврђена различитим техникама мерења, као што су: *file based, delta B, energy profiles* [36]. Резултати су верификовани употребом апликације Андромедар [37] која је коришћена за валидацију енергетске ефикасности пре и после рефакторинга.

Родригез, Лонго и Зунино у раду [38] истражују утицај објектно оријантисаног дизајна на потрошњу електричне енергије, такође на апликативном нивоу. Анализа је извршена на неколико апликација, писаних за Андроид оперативни систем. Приликом тестирања коришћен је оквир (енг. *framework*) *Robotium,* који омогућава аутоматизовано извршавање тестова Андроид апликација док је потрошња електричне енергије мерена коришћењем алата *Power Monitor*. Резултати су показали да висококвалитетно објектно оријентисано дизајниране апликације значајно повећавају потрошњу електричне енергије за 6% - 70%.

Хиндле у [39] истражује утицај промене софтвера на потрошњу енергије као и везе међу софтверским метрикама. Тестирање је извршено над апликацијама: *Firefox, Vuze, rTorrent*, и то над више различитих верзија сваке од њих. Резултати су показали да потрошња енергије није конзистентна кроз различите верзије, али исто тако није пронађена посебна веза током еволуције верзија.

Тонини, Фишер и Блисорара у [40] анализирају утицај две најпопуларније развојне праксе препоручене од компаније Гугл (енг. *Google*) за Андроид апликације у погледу перформанси и потрошње електричне енергије. Резултати су показали да избегавање коришћења јавних метода *getters/setters* значајно побољшава перформансе, као и енергетску ефикасност.

Рад [41] се бави идентификацијом енергетски захтевних *API* (енг. *Application Program Interface*) позива за мобилне апликације засноване на Андроид платформи. Резултати су показали да *API*-и везани за графички интерфејс и обраду слика, као и рад са базама података имају израженије потребе за енергијом. Такође је показано да коришћење идентификатора приступа *getter/setter* повећава потрошњу електричне енергије, због чега се често праве компромиси између примене енкапсулације и енергетске ефикасности.

Шуман и Есам у [42] истражују утицај алгоритама за сортирање на енергетску ефикасност. Анализом алгоритама *Shellsort, Shakersort, Quicksort, Selectionsort,*

Mergesort, Insertionsort и *Bubblesort* утврђено је да је као компромисно решење између перформанси и енергетске ефикасности најбоља опција *Insertionsort* алгоритам. Тестирање је реализовано на низовима од десет милиона елемената.

Рад [43] се фокусира на истраживању утицаја различитих сигурносних механизама заштите софтвера на енергетски аспект, као што су криптографски алгоритми и сигурносни протоколи. Тестирање је реализовано на *SSL* (енг. *Secure Sockets Layer*) протоколу и у обзир су узети сви фактори као што су величина шифрованих порука, механизми аутентификације и трансакциона величина. Утицај дизајна криптографских алгоритама и сигурносних протокола на енергетску ефикасност је детаљније описан у радовима [44, 45, 46].

Атул, Санкаран и Јитиш у [47], баве се моделовањем утицаја обфускације кода на енергетску ефикасност, сигурност и перформансе наменских уређаја (енг. *embedded devices*). Студија је реализована над 4 различита тестна сценарија, коришћењем *MiBench* платформе, за 3 типа обфускације: лексичка, обфускација података, тока извршавања. Резултати показују значајан утицај примене различитих техника обфускације како на перформансе, тако и на енергетску ефикасност.

Банеску, Очоа и Претшнер у [48] предлажу развојну платформу (енг. *framework*) за карактеризацију модела као и способност одбране (степен жилавости) од аутоматских напада. Тестирање је реализовано над апликацијама за аутоматску проверу лиценци, док је за механизам аутоматског напада коришћена *KLEE* [49] апликација.

Аутори Фанг, Ву, Ванг и Ки у раду [50] анализирају различите нивое комплексности [51 - 55] софтвера, на основу чијих резултата предлажу механизам за избор оптималног начина заштите методама обфускације. Поред енергије и сигурности, утицај на перформансе је ништа мање битан фактор приликом избора механизма заштите, нарочито код наменских (енг. *embedded*) уређаја где је потребно задовољити ограничења у реалном времену.

Каинт у [56] предлаже раличите типове хардверске заштите тајних информација електричних кола код *FPGA* (енг. *Field Programmable Gate Array*). Његов приступ укључује моделовање трансформација на *FPGA*, у циљу побољшања перформанси, као и сигурности самих апликација. Додатно, постоје бројни радови који се баве моделовањем потрошње електричне енергије наменских уређаја.

Економу у раду [57] и Иски у [58] предлажу модел потрошње електричне енергије базиран на мерачима перформанси (енг. *performance counters*), док Кан у [59] и Санкаран у [60] моделују потрошњу за вишепроцесорске системе користећи приступ статистичког учења. Поред споменутих радова који се баве повезаним темама, радови [61, 62] приказују нову област истраживања која се тиче детекције енергетских рупа у софтверским апликацијма.

У раду [62], аутори анализирају програмску подршку *"eDoctor"* за аутоматску дијагнозу абнормалног пражњења батерија за Андроид мобилне апликације. Апликација користи концепт фазе извршења како би идентификовала понашање енергије у апликацији што је кључ у процесу проналажења проблематичних сектора. На основу

резултата дијагнозе, *eDoctor* предлаже потенцијална решења крајњим корисницима. Ефикасност апликације је верификована од стране 31 корисника, при чему је вештачки генерисано 17 ситуација абнормалног пражњења батерија (*ABD*). Резултати су показали да је *eDoctor* успешно детектовао 47 од 50 *ABD* ситуација.

Лиу, Чанг и Чунг се у раду [63] баве анализом утицаја "енергетских рупа" Андроид апликација на животни век батерија. За потребе тестирања развијена је апликација *"GreenDroid"*, која имплементира модел извршавања андроид апликација на основу ког се врши дијагноза тестних случајева. Овакав приступ се ослања на аутоматску анализу сензорских података у различитим стањима и на тај начин се врши идентификација критичних сектора, односно места где постоје такозване енергетске рупе.

Боди, Лу и Димитрос у [64] анализирају утицај сензорских компоненти на животни век батерије преносних уређаја. Аутори предлажу нову сензорску архитектуру за мобилне телефоне где се свака обрада података анализира, и по потреби пребацује на наменске процесоре мале снаге. Овакав приступ омогућава уређајима непрекидно очитавање података при малој потрошњи енергије. Резултати су показали да предложена архитектура може бити три реда величине енергетски ефикаснија у односу на досадашње конвенционалне приступе.

Значајна истраживања су реализована на пољу прецизног мерења потрошње енергије. Ова област се може поделити на 3 сегмента: хардверска инструментација [65], симулационо оријентисани приступ [66, 67] базиран на симулацији процесорских циклуса, посматрано са стране архитектуре где се врши процена потрошње енергије сваког извршеног циклуса. Трећи сегмент представља естимационо оријентисан приступ [68, 69], који се заснива на изградњи енергетских модела утицаја, који се касније користе за процену потрошње електричне енергије.

Литке, Зотос, Чацигеоргију и Стефанидес у [70], представљају нови приступ за мапирање утицаја дизајна софтвера на потрошњу електричне енергије. Коришћена је метода поређења профила потрошње тестираних апликација пре и након примене софтверских дизајн образаца. Што се развојних пракси тиче и њиховог утицаја на побољшање енергетске ефикасности, више се може пронаћи у [71, 72].

Преглед литературе показује да постојеће методологије, као и актуелни приступи анализи утицаја различитих облика софтверског инжењеринга на енергетску ефикасност не указују на проблеме обфускационих техника. Међутим, аутори у [8] дају предлог архитектуре за генерисање профила потрошње на нивоу асемблерских инструкција као једног облика метрике мерења квалитета софтвера. Рад појашњава утицај различитих обфускационих техника и даје класификацију истих по степену утицаја на енергетску ефикасност.

1.3 Потреба за истраживањем и циљеви истраживања

Први концепти обфускације се спомињу од стране Дифија и Хелмана [73] у концептима размене кључева.

С обзиром да енергија постаје веома битан фактор у дизајнирању наменских рачунарских система, значајна средства се улажу како би се потрошња електричне енергије минимизовала [74, 75]. Захваљујући великој популарности и брзом развоју информационих технологија, софтверске компаније се све више суочавају са опасношћу од крађе интелектуалне својине. Захваљујући техникама инверзног инжењеринга, злонамерни корисници могу са лакоћом открити разне битне технолошке иновације/тајне, све у циљу разумевања пословне логике. Постоје разни облици заштите програмског кода: енкрипција, обфускација, *code morphing*, *security through obscurity*, итд. Обфускација представља најчешће коришћену технику, која се спроводи да би се смањио ризик од неовлашћеног приступа изворном коду који укључује губитак интелектуалног власништва, налажења рањивости софтвера као и економских губитака које појединац или компанија могу да претрпе.

Преглед литературе показује да још није обрађен проблем естимације додатне потрошње електричне енергије на нивоу асемблерских инструкција услед примене различитих обфускационих техника. Наиме, већина наведених метода су оријентисане на идентификацију утицаја различитих развојних пракси на енергетску ефикасност, док је у овој докторској дисертацији обрађен утицај 3 најчешће коришћена типа обфускације кода: лексички, обфускација података и обфускација тока извршавања.

Општи циљ истраживања у овој докторској дисертацији јесте анализа утицаја обфускације кода на потрошњу електричне енергије у случају програмске подршке писане за .НЕТ (енг. *.NET*) платформу, за горе споменута 3 типа заштите кода.

Ради постизања наведеног циља, идентификоване су следеће хипотезе:

1. Х1: Показати да обфускација кода мења профил потрошње у зависности од типа обфускације као и тестираног узорка.
2. Х2: Показати да обфускација тока извршавања генерише енергетски захтевнији профил потрошње од лексичке обфускације, односно код који троши више електричне енергије приликом свог извршавања.
3. Х3: Показати да обфускација података генерише енергетски захтевнији профил потрошње од профила насталог применом обфускације тока извршавања.

Формирати низ тестних сценарија са различитим степеном комплексности како би се детаљније уочиле разлике у квалитету заштите обфускационим методама. Тестирањем показати утицај различитих нивоа обфускације на потрошњу електричне енергије. Тестирање реализовати за следеће софтверске пакете:

- *Confuser*,
- *Agile.NET*,

- *Smart Assembly,*

Развити софтверску подршку за потребе генерисања профила потрошње електричне енергије. На основу добијених резултата, извршити поделу коришћених софтверских пакета по анализираним критеријумима.

1.4 Преглед докторске дисертације

У другој глави докторске дисертације представљени су актуелне модели заштите софтвера, како законски, тако и технички. Затим су у глави три дате теоријске основе најчешће коришћених обфускационих техника заштите са акцентом на лексичку, заштиту података и заштиту тока извршавања. Четврта глава представља архитектуру софтверске подршке развијене за генерисање профила потрошње, са освртом на концепт инверзног инжењеринга. Такође, објашњени су основни појмови везани за технике инверзног инжењеринга. У петој глави, представљени су резултати анализе обфускатора изворног кода и њихов утицај на енергетску ефикасност за анализиране софтверске пакете.

Закључци докторске дисертације представљени су у шестој глави, док је коришћена литература дата у седмој глави. На крају докторсе дисертације, налазе се прилози и биографија аутора.

2. Заштита софтвера

Сигурност софтвера представља веома битан фактор у данашњој *IT* индустрији због великих финансијских губитака који се годишње мере милијардама долара. Софтверски пакети се често испоручују у неконтролисаним или мање сигурним окружењима као што су разне државне институције, универзитети, истраживачки институти, итд. У извештају који је објавило удружење за анализу софтверског пословања [76], комерцијална вредност софтверске пиратерије за 2010. годину је износила око 60 милијарди долара, док Гартнер у свом извештају [77] наглашава да употреба програмских језика који користе интерпретер, односно језика који су независни од платформе на којој се извршавају као што је Јава, представљају посебан изазов у области пиратерије и крађе интелектуалне својине. Код оваквих језика се изворни код преводи у такозвани међујезик (бајткод) који садржи све неопходне информације. Злонамерни корисници (хакери), могу са лакоћом да изврше реконструкцију бајткода у оригинални и на тај начин дођу до информација од интереса (криптографски кључеви, алгоритми, као и друге користе информације).

У циљу заштите интелектуалног власништва, као и финансијског интереса, произвођачи су приморани имплементирати разне софтверске и хардверске облике заштите. Основни облици оваквих заштита подразумевају проверу серијског броја, хардверских особина машине на којој је софтвер инсталиран, аутентификацију, проверу лиценци, итд.

Тренутно, на тржишту не постоје 100% ефикасне методе заштите софтвера. Основни циљ и најефикаснијих метода је повећање времена неопходног за уклањање заштите са намером демотивације злонамерних корисника. Пракса је показала да су у неким ситуацијама потребне године да би се појавила пиратска верзија, што је сасвим довољан период да сама апликација изгуби на вредности и да се на тај начин спречи значајнија финансијска штета.

Као комплексан производ, софтвер оставља могућност уградње вишеструких заштита у различитим сегментима програмског кода, што компликује процес “разбијања”. Код оваквих заштита, постоји шанса да пиратске верзије привидно обављају своју функционалност (нпр. покретање апликација), да би у каснијим фазама рада долазило до нестабилног рада, а неретко и до “пуцања”, односно гашења самих апликација.

Класификација рачунарске сигурности се по Мејну и Оршоту у раду [78] може поделити на три типа:

- **сигурност података** – област која се бави интегритетом и поверљивошћу података у процесу преноса и складиштења,
- **мрежна сигурност** – област која има за циљ заштиту мрежних ресурса, услуга и уређаја,

- **софтверска сигурност** – област која штити софтвер од неовлашћеног приступа, коришћења и модификације, а која је обрађена у овом поглављу.

У наставку овог поглавља биће дат осврт на различите видове заштите софтверских производа, како правних тако и техничких док је у глави 3 детаљно обрађена техника заштите методом обфускације.

2.1 Законска заштита

Један од основних предуслова за борбу против софтверске пиратерије јесте доношење правних аката (закона) у циљу регулисања права аутора као и крајњих корисника [79]. Изазов при доношењу закона је немогућност јасног сврставања софтвера у познате категорије интелектуалног власништва, јер се софтвер може посматрати на два начина. Са једне стране, изворни код, као и начин имплементације различитих алгоритама и синтакси одређеног програмског језика се могу посматрати као литерарно дело и као такво га је могуће заштити ауторским правом. Са друге стране, софтвер као механизам, може обављати користан посао и такве ствари се најчешће штите патентима. Друга димензија која прави проблем приликом доношења закона за заштиту софтвера је степен заштите коју је неопходно имплементирати. Наиме, компанијама је у интересу да имају што бољу заштиту како би имале и већи финансијску корист док крајњи корисници желе да софтвер буде што доступнији и отворенији. Сходно споменутом, према законском облику заштите, разликујемо следеће категорије софтверских решења:

- **јавни домен** (енг. *public domain*) – софтвери ове категорије остављају могућност корисницима да безусловно користе, умножавају, дистрибуирају, па чак и продају само решење без дозволе аутора.
- **отворени код** (енг. *open source*) – софтвери ове категорије су бесплатни за коришћење, дистрибуцију и умножавање. Корисницима се оставља и могућност измене изворног кода, али под условом да нова верзија и даље буде *open source*.
- ***freeware*** – софтвер ове категорије је бесплатан за дистрибуцију и коришћење, али за разлику од *open source* варијанте, није дозвољена измена оригиналног кода. У овој категорији аутор задржава право на софтвер.
- ***shareware*** – за разлику од *freeware* категорије, софтвери ове категорије лиценцним споразумом подразумевају да се аутору уплати одређена свота новца. Бесплатна употреба софтвера је временски ограничена, док у случају потребе додатног коришћења, потребно је купити лиценцу. Добра страна ових софтвера је што су лиценце веома јефтине, како би се привукао што већи број корисника, као и повећан облик дистрибуције истог.
- **комерцијални** – софтвери ове категорије се плаћају, не смеју се дистрибуирати, копирати или мењати.

2.1.1 Ауторско право

Заштита ауторским правом [80] представља најзаступљенији начин заштите било ког интелектуалног власништва, односно заштите аутора од нелегалног коришћења његовог дела као што је:

- умножавање,
- измене и допуне,
- дистрибуција,
- јавно коришћење (финансијска корист).

Ауторско право штити оригиналну имплементацију као и начин приказа идеје, али не и саму идеју, што уједно представља проблем овог облика заштите. Ако бисмо ово применили у свету информационих технологија, то значи да бисмо ауторским правом заштитили извршни и изворни код, организацију и структуру, упите, документацију у било ком облику, али не и програмске алгоритме као и коришћене математичке поступке. Ауторско право такође штити од неовлашћеног копирања, али не и од конкуренције која може самостално да развије сличан софтвер. Овај облик заштите се често користи због једноставне применљивости као и једноставности реализације.

2.1.2 Патент

За разлику од ауторског права, патент [81] штити оригиналност неког рада и представља најмоћнији законски облик заштите. У софтверској индустрији, патентом се штите идеје, алгоритми, као и разни математички поступци, али не и сам код. Патент представља заштиту изума који издаје држава са циљем спречавања производње и продаје истог или сличног производа. Мана овог облика заштите је цена, комплексност као и време неопходно за издавање (неретко се чека и неколико година) права на патент. Заштита патентом у *IT*-у је мање заступљена због чињенице екстремно брзог развоја саме индустије и користи се за заштиту фундаменталних ставки за које се процени да ће опстати на тржишту минимум десет или више година.

2.1.3 Лиценца

Лиценца представља проширење ауторског права за неки софтвер, односно корисник не купује софтвер већ лиценцу за његово коришћење, док аутор остаје власник производа. Лиценца представља посебну дозволу где је веома јасно дефинисан начин коришћења софтвера од стране корисника, тј. прописано је на колико рачунара се софтвер сме инсталирати, сврха коришћења (образовна, комерцијална, основна, итд.), као и време трајања.

2.2 Техничка заштита

Највећи недостатак законских мера заштите представља применљивост у пракси, тј. веома је тешко открити прекршитеље и још теже судски доказати кривицу. Управо ови проблеми су мотивисали произвођаче да своје производе заштите разним техничким мерама. Ови облици заштите би се могли поделити на две категорије: хардверски и софтверски. У овом поглављу ће бити дат кратак осврт на различите облике заштите у обе категорије.

2.2.1 Адаптер

Заштита адаптером [82] (енг. *dongle*) представља хардверски облик заштите који подразумева повезивање на неки од портова (углавном *USB*). Адаптер углавном садржи хардверски имплементиран кључ за омогућавање приступа одређеној апликацији. Сама апликација унутар себе имплементира методе које комуницирају са *dongle* кључем. Ако адаптер није присутан или ако пошаље погрешан кључ, софтвер престаје са радом. Поред заштите од нелегалне дистрибуције, адаптер такође може да онеспособи одређене делове кода, у зависности од купљене верзије. Хаковање самог адаптера је веома тешко, па се злонамерни корисници одлучују да једноставније решење, тј. "пробијање" метода за комуникацију са апликацијом.

Генерално, овај облик заштите се данас скоро више и не користи због следећих разлога:

- компликована инсталација и потреба за новим верзијама управљачког софтвера за исправан рад адаптера,
- висока цена, јер је сваком кориснику потребно испоручити посебан адаптер,
- немогућност дистрибуције путем глобалне мреже (интернета).

2.2.2 Фиксна шифра

Овај модел заштите [82] представља најједноставнији облик који подразумева уношење шифре од стране корисника која је статичка и не зависи ни од каквих параметара као што су лични подаци корисника, или подаци о машини. Лоша страна овог облика заштите је што се једном откривена шифра може употребити неограничен број пута. Предност овог облика заштите је то што исправна шифра за регистрацију не мора бити смештена на неком оптичком медијуму већ се може налазити у самом коду. До те шифре је веома тешко доћи анализом програмског кода, јер се обе шифре (шифра коју уноси корисник и шифра са којом се пореди) пре трансакције енкриптују, односно преводе у нове вредности након чега се врши поређење.

Злонамерни корисници овај проблем заобилазе тако што преправљају делове кода за проверу шифре. Да би се спречило овакво пробијање, неопходно је користити регистрациону шифру за декриптовање одређених делова кода, односно, ако је део кода

енкриптован помоћу регистрационе шифре, онда га је могуће декриптовати само помоћу те исте шифре. На тај начин се спречава извршавање кода након примене методе премошћавања (преправљања).

2.2.3 Динамичка шифра

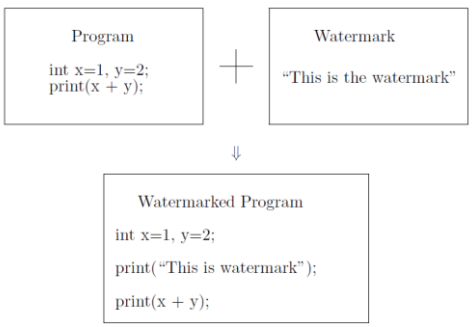
За разлику од фиксне/статичке, заштита софтвера динамичким/генеричким шифрама представља ефикаснији метод [82]. Шифре се генеришу на основу података о кориснику као што су адреса, име компаније, лични подаци власника, итд. Корисник апликације шаље произвођачу софтвера генерисану шифру, на основу које му се прослеђује адекватна активациона шифра која му омогућава рад апликације.

2.2.4 Хардверски серијски број

Овај вид заштите [82] представља најчешће коришћен хардверски метод заштите софтвера. Метода функционише на тај начин што се након инсталације генерише псеудо-случајни идентификациони број машине након чега апликација енкриптује тај број и снима га у посебну датотеку. Активациона шифра апликације се генерише помоћу серијског броја испоручене верзије софтвера и из псеудо генерисаног идентификационог броја машине.

2.2.5 Заштита воденим жигом (енг. software watermarking)

Заштита софтвера воденим жигом [83] представља методу која се заснива на уграђивању тајних информација у сам код апликације, како би се осигурало власништво над производом, а уједно и очувала оперативна семантика програма. На овај начин се носиоцима ауторског права омогућава утврђивање власништва извлачењем тајне поруке из неовлашћене копије. Поред техника које штите софтвер од пиратерије, софтверски водени жиг [84, 85] је јединствен по томе што нема за циљ да спречи софтверску пиратерију него настоји да докаже да се иста десила. Једноставна илустрација ове технике приказана је на слици 2.1, где водени жиг представља реченица "*This is watermark*".



Слика 2.1 Једноставан пример уграђеног воденог жига [83]

Што се техника напада тиче, односно откривања водених жигова, разликујемо 4 приступа:

- **инверзни инжењеринг** – једна од најчешће примењених техника када се жели анализирати интелектуална својина.
- **анализа тока извршавања** – анализом историје извршавања, конкретно условних петљи, улазних и излазних секција као и тачки гранања, злонамерни корисници долазе до битних информација.
- **анализа изворног кода**,
- **анализа *stack/heap* меморијског простора**.

Кристијан Колберг у радовима [86, 87] описује 3 битне особине/својства заштите софтвера техником воденог жига:

- **жилавост** (енг. *resilience*) – представља способност издржавања напада на ову технику заштите,
- **невидљивост** (енг. *stealth*) – квантификује разлику између типова инструкција које се користе за уградњу воденог жига и других инструкција које се користе за друга програмска израчунавања,
- **брзина преноса података** (енг. *data rate*) – мери трошкове које намеће техника воденог жига као што су: повећање величине кода, време извршавања и утрошак меморије.

Software watermarking се може поделити на неколико начина, како по својим својствима тако и по функционалности. Најчешће разликујемо следеће класификационе категорије водних жигова: према намени, техници извлачења, степену робусности, видљивости.

Наменски водени жиг

Ова категорија се односи на крајњу функционалност заштите [88, 89], где сваки водени жиг има јединствен циљ у таксономији:

- превенциони – жиг за спречавање неовлашћеног коришћења софтвера,
- потврдни – жиг који дефинише власништво над софтвером,
- ознака дозволе – жиг који дефинише степен модификације софтвера,
- афирмациони – жиг који осигурава аутентификацију крајњем кориснику.

Подела према техници извлачења/откривања

Ова категорија се може поделити на две класе [90], статички и динамички водени жигови.

Статички водени жиг

Овај облик воденог жига налази се у апликативној извршној датотеци. На пример, може се налазити у иницијализованом делу података извршног фајла, или у стринг (енг. *string*) ресурсима. Постоје два типа статичких водених жигова:

- **жигови за податке** (енг. *data watermarks*) – складиште се најчешће у заглавље извршне датотеке (нпр. *.exe*) или у датотекама за дебаговање (енг. *debug*), као што је рецимо *.pdb*.
- **жигови за код** (енг. code *watermarks*) – складиштени су у стварном скупу инструкција, односно у самом коду апликације.

Један од недостатака ове методологије је што се лако може уклонити помоћу обфускационих техника. Додатно, статичком анализом кода може се открити структура, као и информације садржане у самој апликацији, и самим тим релативизује цео процес заштите.

Динамички водени жиг

Динамички водени жигови се креирају у току рада апликације (енг. *runtime*) и складиште се у извршним стањима програма, за разлику од водених жигова за податке који се ослањају на статичку семантику. Када апликација покрене унапред одређену улазну секвенцу, она улази у стање које представља водени жиг. Постоје 3 типа динамичких водених жигова:

- **ЕЕ жиг** (енг. *easter egg watermark*) – најчешћи и најпопуларнији тип динамичког жига који се активира након уношења посебно припремљене улазне секвенце, као што је низ знакова за ауторска права.
  Мана овог воденог жига је што се лако детектује и уклања, како ручним тако и аутоматских методама.
- **жиг за праћење динамичког извршавања** – водени жиг се уграђује у траг (енг. *trace*) извршног програма када се покреће са специфичном улазном секвенцом. Детекција и уклањање се може извршити праћењем адреса тока извршавања. Код асемблерског кода би то подразумевало праћење стања стека или регистара. Додатни начин уклањања ове заштите је могућ применом обфускационих техника као и неким облицима оптимизације кода.
- **жиг динамичке структуре података** – ови типови водених жигова су уграђени у структуре података саме апликације. С обзиром да се не креира излаз као у ЕЕ жигу, злонамерним корисницима је далеко теже да пронађу стање у коме постоји водени жиг. Пример овог типа жига може бити команда "*if*" која наводи да ли је достигнуто одређено стање

N, након чега се иницијализује одређена структура садржајем или идентификаторима воденог жига.

Недостатак овог облика заштите је што постоје обфускационе трансформације кода које се могу користити за промену динамичког стања и изобличавање оваквог воденог жига. На пример, променљиве се могу поделити на неколико нових тако што се генеришу функције које одржавају семантику рада, тј. омогућавају конверзију између оригиналних и новодобијених података. На сличан начин се може урадити и инверзно, тј. спајање две или више променљивих уз генерисање адекватних функција за одржање функционалности.

Подела према степену робусности

Робусност је један од најважнијих аспеката дигиталног воденог жига. Ова категорија се може поделити на три класе, на крхке, полукрхке и робусне водене жигове. Крхки жигови се користе у верификацији интегритета софтвера, као и у системима који дозвољавају ограничено коришћење. Крхки водени жиг је жиг који се не може открити након што се деси и најмања модификација. Полукрхки водени жиг је жиг који је отпоран на "бенигне" трансформације, док се не открива након "малигних" трансформација. Робустан жиг је отпоран на одређене класе трансформација, што аутоматски не значи да је бољи од крхког. Крхки и полукрхки се најчешће користе за детекцију малигних трансформација и заштиту интегритета дигиталних сигнала.

Робусни водени жигови користе се за превенцију неовлашћеног коришћења, у системима за потврду власништва [90] и у системима за заштиту од копирања.

Подела према видљивости

Према функционалностима које корисник софтвера може да искуси, ова категорија се може поделити на две класе, на видљиве и невидљиве водене жигове. Видљиви жигови често се веома јасно приказују како би их сви могли видети и како би на тај начин деловали као средство одвраћања од злоупотребе (нпр. новчанице). Један од примера је и лого који телевизијски емитери често додају у угао видео записа. Невидљиви жигови су заступљенији и могу се открити једино коришћењем тајне шифре, недоступне крајњем кориснику.

3. Обфускација

Масовном коришћењу интерактивног садржаја на интернету као што су анимације, видео записи, игре, итд. значајно је допринела компанија *Sun Microsystems* када је креирала програмски језик Јава (енг. *Java*), средином деведесетих година. *Java* је замишљена да буде архитектурално независна с обзиром да су на интернет, разним хетерогеним хардвером повезане различите архитектуре. Ова независност реализована је захваљујући *JVM*, на којој се извршава бајткод, односно код који је независан од било ког програмског језика. Као такав, бајткод садржи скоро све информације о изворном коду, што оставља велики простор за злоупотребу, односно крађу интелектуалног власништва. Обфускација представља технику заштите која претвара оригинални програм, у други, семантички еквивалент, знатно тежи за разумевање и инверзни инжењеринг.

Циљ заштите софтвера техником обфускације јесте трансформација изворног кода апликације до момента када постаје неразумљив аутоматским алатима за програмско разумевање или док резултати анализе постану некорисни злонамерним корисницима.

Мотив за заштиту софтвера кроз обфускацију произилази из проблема софтверске пиратерије, која се може резимирати као процес инверзног инжењеринга кода са намером крађе интелектуалне својине комерцијалних софтверских пакета. Апликације попут *Skype VoIP* клијента [91], *SDC Java DRM* [92], као и многе друге, користе технику обфускације као вид заштите. Кристијан Колберг је са осталим ауторима у [93, 94] први формално дефинисао обфускацију кода као семантички одрживу трансформацију функције O која мапира програм P у програм $O(P)$. Међутим, после историјских резултата до којих је дошао Барак у [95], где је доказао да ни један алат за обфускацију неће потпуно успети да заштити све програме, изгледи за дизајнирањем савршеног алата (*black-box*) за заштиту свих класа програма су пали у воду.

У наставку поглавља су дате две дефиниције технике обфускације са пратећим мерама квалитета истих, као и детаљан приказ теоријских основа постојећих и анализираних обфускационих техника у овој докторској дисертацији. На крају поглавља приказане су предности и мане овог облика заштите.

3.1 Колбергова дефиниција

На основу Колбергове дефиниције [94] обфускација се дефинише на следећи начин:

Нека је $P \xrightarrow{\mathcal{T}} P'$ трансформација изворног програма P у програм P'. Трансформација $P \xrightarrow{\mathcal{T}} P'$ је обфускациона трансформација ако и само ако и P и P' имају

очекивано понашање. Прецизније, да би $P \xrightarrow{\mathcal{T}} P'$ била легална трансформација обфускације, следећи услови морају да буду задовољени:

- ако се програм P не изврши успешно или се изврши са одређеним грешкама, онда P' може а и не мора да се изврши,
- у супротном, P' мора да се изврши успешно и да има исти резултат као и P.

Квалитет трансформација се мери различитим метрикама које обухватају комплексност кода као што су:

- **цикломатична сложеност** (енг. *cyclomatic complexity*) [51] – сложеност функције се повећава са бројем условних израза,
- **угњеждена сложеност** (енг. *nesting complexity*) [96] – сложеност функције се повећава са већим нивоом угњеждавања условних грана програма,
- **сложеност структуре података** (енг. *data structure complexity*) – сложеност се повећава са сложеношћу статичких структура података декларисаних у програму, односно комплексност низа је сразмерна његовој димензији и комплексношћу типа самог елемента.

Коришћењем наведених метрика, аутор у [94] мери потенцијал (енг. *potency*) обфускације на следећи начин:

Нека је $\mathcal{T}$ трансформација која мапира програм P у програм P', и нека $E(P)$ представља комплексност програма P. Потенцијал $\mathcal{T}_{pot}$ трансформације $\mathcal{T}$ у односу на програм P представља меру у којој $\mathcal{T}$ мења комплексност програма P:

$$\mathcal{T}_{pot}(P) = \frac{E(P')}{E(P)} - 1 \tag{3.1.1}$$

$\mathcal{T}$ представља потенцијал трансформације обфускације ако је $\mathcal{T}_{pot}(P) > 0$, тј. ако важи да је $E(P') > E(P)$.

Ефикасност програма P' и P углавном није иста нити се тако нешто очекује. У раду се наводи да различите трансформације утичу на програм P' са аспекта веће потрошње меморије као и дужег времена извршавања. Додатне метрике које Колберг спомиње су:

- **жилавост** (енг. *resilience*) – под овим критеријумом се мери жилавост обфускованог кода на алгоритме за аутоматско демаскирање, тј. инверзну обфускацију (енг. *deobfuscation*) [93, 97]. Ова метрика узима у обзир време неопходно за конструкцију процеса демаскирања, време извршавања као и утрошени меморијски простор неопходан за рад апликације.
- **цена извршавања** (енг. *execution cost*) – под овим критеријумом мере се трошкови обфускације као што су додатно време извршавања програма, утрошак меморије и складишног простора [93, 97] између програма P' и P.
- **квалитет** (енг. *quality*) – метрика квалитета обухвата жилавост, цену извршавања као и потенцијал, у циљу дефинисања генералне метрике квалитета.

Ове три метрике неформално се мере на ненумеричким скалама (нпр. жилавост се мери на скали: тривијално, слабо, јако, веома јако, једносмерно).

Додатна корисна метрика коју аутор идентификује је невидљивост/притајеност (енг. *stealth*) обфускације. Под овим критеријумом подразумева се "колико добро се обфусковани код интегрише са остатком програма", као и степен сличности оригиналу. Метрика притајености зависи од контекста и сходно томе постоји могућност да притајеност у једном окружењу не значи аутоматски исто и за друга окружења, због чега ју је веома тешко квантификовати.

3.2 Баракова дефиниција

Барак у [95] даје формалнији приступ самом термину обфускације као и свему што из тога следи. Алат за обфускацију O, представља преводилац/компајлер који обрађује улазне податке програма P и производи их у нови програм $O(P)$ такав да за свако P важи:

- функционалност (енг. *functionality*) – $O(P)$ прорачунава исту функцију као и P,
- полиномијално успорење (енг. *polynomial slowdown*) – величина програма и време извршавања $O(P)$ су полиномијално већи од P.

3.3 Обфускационе трансформације

У овом поглављу дат је преглед техника коришћених за анализу у овој дисертацији, а које су споменуте у [93, 94]:

- лексичка обфускација,
- обфускација тока извршавања,
- обфускација података.

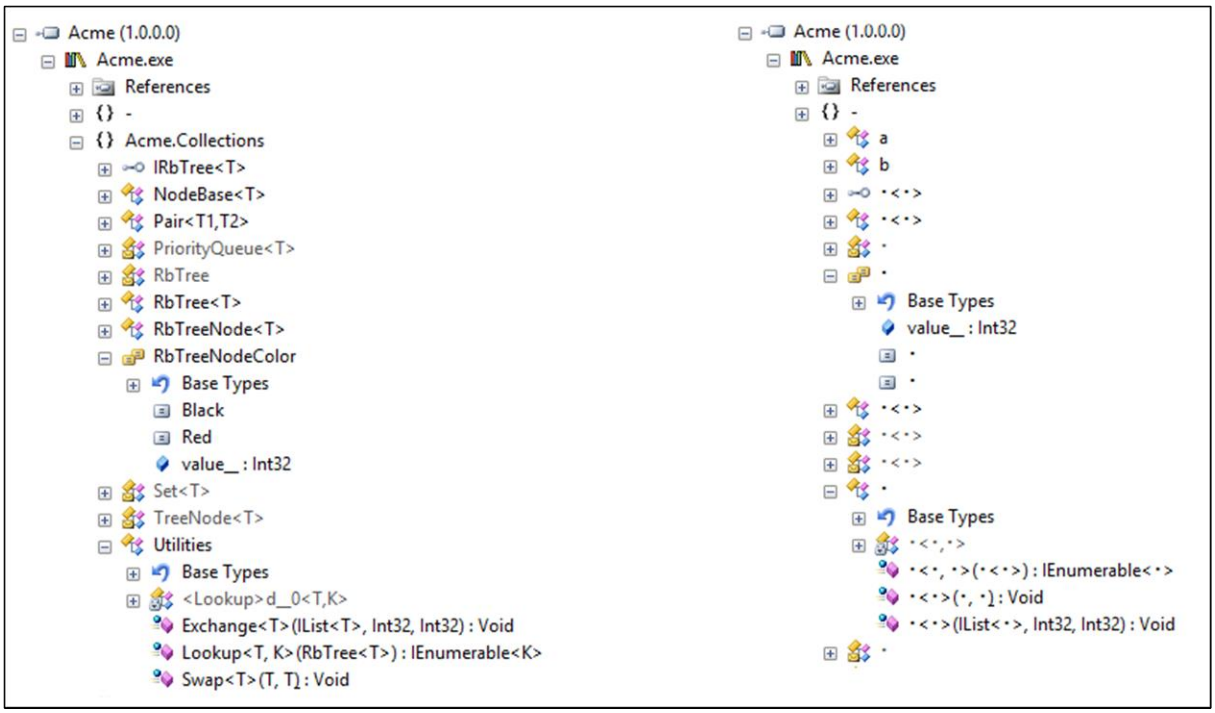
3.3.1 Лексичка обфускација

Овај облик заштите заснива се на промени лексичких параметара кода анализираног програма, и подразумева следеће:

- **уклањање коментара присутних у коду** – коментари се често уносе као део документације и као такви, омогућавају разумевање логичких целина за које су везани (семантичка повезаност), што злонамерном кориснику олакшава процес крађе интелектуалне својине. Због наведеног, један сегмент лексичке обфускације подразумева уклањање коментара из кода.
- **форматирање кода** – подразумева уклањање вишка празних врста, редова и корекција увлачења кода [98],

- **шифровање/промена имена идентификатора** (програмских променљивих, имена класа, метода, догађаја, именских простора, параметара метода, итд.) – у овом приступу се оригиналне променљиве мењају лексички мање описним идентификаторима. Злонамернији облик коришћења ове технике би био генерисање логичких имена идентификатора који имају лажну конотацију (нпр. променљива која се зове "*totalSum*" би могла да има лажно име "*averageSum*"). На овај начин се не оставља утисак коришћења лексичке обфускације, а кориснику се шаље лажна слика о функционалности анализираног кода. Ово је једносмерна трансформација с обзиром да се оригинална имена идентификатора не могу идентификовати без додатних трошкова. Ову обфускациону технику садрже сви алати на тржишту, како комерцијални тако и софтвери отвореног кода. Rad [45] даје неколико примера са резултатима заштите кода обфускацијом идентификатора од чега су посебно издвојена два. Приказани су експерименти на два различита софтверска пакета са две врсте напада (аматерски и професионални). Резултати су показали да заштита кода обфускацијом идентификатора значајно повећава време код искусног/професионалног нападача, док је код аматера то време вишеструко веће. Ефикасност ове технике на примеру Јава програмског језика је дата у [99, 100] док се о проширењима основне идеје може више пронаћи у [101].

Лексичка обфускација је веома заступљена и не сматра се снажним видом заштите, јер се може лако уклонити баш из разлога што се не бави семантиком програма. На слици 3.1, приказан је изглед кода пре (лево) и након лексичке обфускације (десно), инверзно преведен (у даљем тексту декомпајлиран) у пакету *.NET Reflector*.

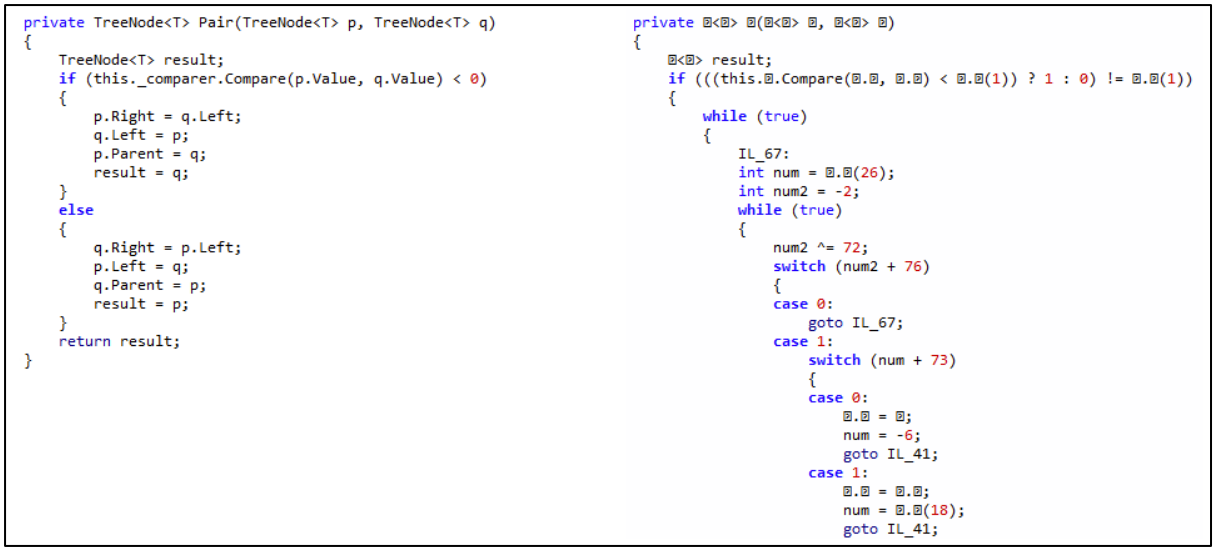


Слика 3.1 Лексичка обфускација (оригинал (лево), обфускован код (десно)) [102]

3.3.2 Обфускација тока извршавања

Насупрот лексичкој трансформацији која се бави синтаксним изменама програма, трансформације тока извршавања се баве семантичким изменама. Идеја ове врсте обфускације је промена тока извршавања оригиналног кода увођењем нових, лажних токова (вештачких грана који немају еквивалентне изворне кодове), петљи, условних грана [103]. Техника се заснива на креирању јефтиних, еластичних и неразумљивих израза (енг. *opaque predicates*, у даљем тексту "*OP*") који у суштини представљају условне изразе познате компајлеру за обфускацију, али их је веома тешко статички неутралисати у циљу инверзног инжењеринга. Rad [93] описује методу за креирање *OP*-а како би заштитили код од статичке анализе. *OP* су *bool* вредновани изрази и њихова основна идеја је креирање и манипулисање динамичким структурама података (нпр. графови) који садрже псеудоним променљиве (енг. *alias*) одржавајући одређене услове. Управо ови изрази се користе за креирање *OP*-а по потреби.

Декомпајлиран пример заштићен овом техником приказан је на слици 3.2. Са леве стране налази се оригинални, код док је са десне стране приказан део обфусковане верзије истог. Да се приметити да декомпајлирана обфускована верзија садржи далеко већи број инструкција, семантички нејасних (као што је име методе, делови кода унутар *case*-ova). Изглед обфусковане верзије зависи од самог алата, односно коришћеног алгоритма. Углавном, изглед садржи симболе нејасне људском оку, као што су кукице у примеру са десне стране, а све са циљем отежавања инверзног инжењеринга.



```
private TreeNode<T> Pair(TreeNode<T> p, TreeNode<T> q)
{
    TreeNode<T> result;
    if (this._comparer.Compare(p.Value, q.Value) < 0)
    {
        p.Right = q.Left;
        q.Left = p;
        p.Parent = q;
        result = q;
    }
    else
    {
        q.Right = p.Left;
        p.Left = q;
        q.Parent = p;
        result = p;
    }
    return result;
}
```

```
private ⍰<⍰> ⍰(⍰<⍰> ⍰, ⍰<⍰> ⍰)
{
    ⍰<⍰> result;
    if (((this.⍰.Compare(⍰.⍰, ⍰.⍰) < ⍰.⍰(1)) ? 1 : 0) != ⍰.⍰(1))
    {
        while (true)
        {
            IL_67:
            int num = ⍰.⍰(26);
            int num2 = -2;
            while (true)
            {
                num2 ^= 72;
                switch (num2 + 76)
                {
                case 0:
                    goto IL_67;
                case 1:
                    switch (num + 73)
                    {
                    case 0:
                        ⍰.⍰ = ⍰;
                        num = -6;
                        goto IL_41;
                    case 1:
                        ⍰.⍰ = ⍰.⍰;
                        num = ⍰.⍰(18);
                        goto IL_41;
```

Слика 3.2 Ток извршавања: оригинал (лево), обфускован код (десно) [102]

Аутори у раду [104] предлажу примену 3 типа заштите (бајткод, бинарно и обфускацију изворног кода) са коначним циљем ефикасније заштите тока извршавања, док се другачије методологије могу пронаћи у [105, 106].

Обфускација тока извршавања подразумева следеће:

- **агрегација**:
  - замена позива метода њиховим садржајем (енг. *inlining methods*),
  - креирање метода на основу стања (енг. *outlining statements*),
  - креирање дупликата метода са истом функционалношћу (енг. *clone methods*),
  - дуплирање и промена стања петљи (енг. *unroll loops*),
  - спајање петљи,
  - клонирање основних градивних блокова кода.
- **преуређивање/измена позиције (енг. *ordering*)**:
  - измена позиција метода као и њихових позива,
  - измена позиција условних петљи,
  - интерпретација табела.

Неразумљиви изрази (енг. *opaque predicate*)

Израз (у даљем тексту "предикат") X је неразумљив [93] (*opaque*) у програмској тачки s ако је вредност X у тачки s позната компајлеру у тренутку превођења. Нотација X_s^T (X_s^F) означава вредности предиката у анализираној тачки s које увек имају вредности *true* (*false*) док $X^?$ означава предикат чија вредност, понекад може бити *true* а понекад *false*. Неразумљиви изрази се користе за креирање лажног кода, односно генерисања/компликовања дијаграма тока извршавања. Општи пример коришћења предиката, дат је у следећој математичкој форми [107]:

$$S \Rightarrow \textbf{if}\ (X^T)\ \{S\} \tag{3.3.2.1.1}$$

$$S \Rightarrow \textbf{if}\ (X^F)\ \{S_{bug}\}\ \textbf{else}\ \{S\} \tag{3.3.2.1.2}$$

$$S \Rightarrow \textbf{if}\ (X^?)\ \{S\}\ \textbf{else}\ \{S_{copy}\} \tag{3.3.2.1.3}$$

S – формулација путање извршавања програма који се тестира,

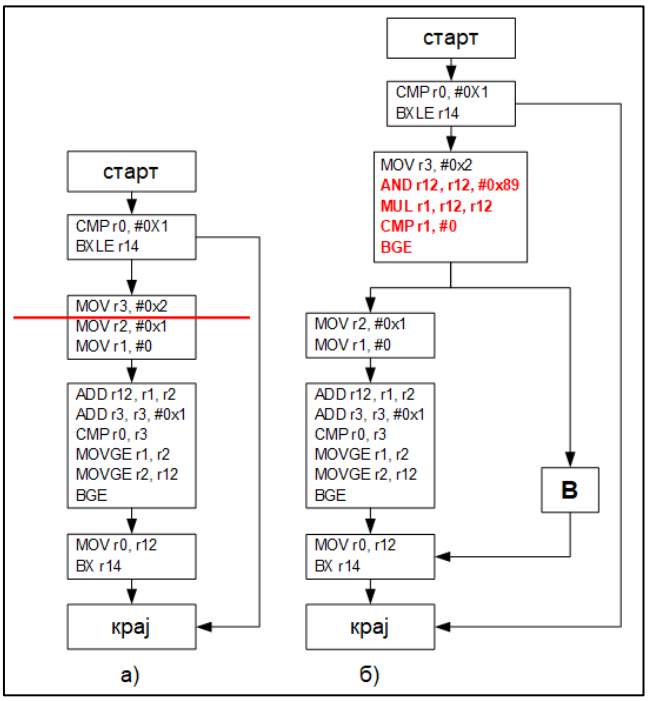
s – позиција у коду где се налази уграђени предикатски израз,

X_s^T – предикат у анализираној тачки има вредност *true*,

X_s^F – предикат у анализираној тачки има вредност *false*.

Израз (3.3.2.1.1) прикрива чињеницу да ће трансформација S увек бити извршена. Друга трансформација (3.3.2.1.2) користи копију S која садржи грешку (енг. *bug*) док трећа (3.3.2.1.3) користи функционално еквивалентну копију S, односно S_{copy}.

Сами предикати се могу заснивати на различитим математичким изразима, нпр. $n^2 + 1 \neq 0 \pmod 7$ за све целобројне вредности n. Ови изрази су или прилично лако доказиви или су до те мере компликовани да се издвајају од остатка програма (тј. нису прикривени). На слици 7, илустрован је пример коришћења предиката. Леви блок дијаграм (слика 3.3а) приказује део оригиналног асемблерског кода, где црвена линија показује позицију (трећи блок, између команди за манипулацију над регистрима) уграђивања предиката, односно позицију s. Десни блок дијаграм (слика 3.3б) приказује асемблерски код након уградње предиката (новододате команде означене црвеном бојом). Овакав, новодобијени ток извршавања је знатно тежи за инверзни инжењеринг, поготово за креирање графа тока извршавања. Више о предикатима, као и њиховим унапређењима се може видети у [108, 109].



Слика 3.3 Примена предиката: а) оригинални код, б) након примене

Ирационални графови токова

Колберг у свом раду [93] дискутује о коришћењу предиката у циљу креирања лажних израза који делују као да позивају одређење кодовне блокове који са аспекта графа тока извршавања делују ирационално. Ово такође представља један од механизама који додатно компликују/отежавају инверзни инжењеринг кода. Формулација ирационалног позива је дата изразом (3.3.2.2.1) [107]:

$$\begin{aligned} &\textbf{if } (X^F)\ \{\textbf{goto } L;\} \\ &\ldots \\ &\textbf{while } (C) \\ &\{\ldots L\colon \ldots\} \end{aligned} \qquad (3.3.2.2.1)$$

X^F – уграђени предикат са вредношћу *false*,

L – нова позиција кода за компајлер,

C – произвољни услов за излазак из петље.

Програмски језици као што су *C#* и *Java* не дозвољавају имплементацију оваквог кода приликом развоја, али би се овакав модел могао имплементирати у међујезицима (*CIL*, *Bytecode*), што би захтевало додатно компајлирање након додавања лажних кодовних секција. Било какав покушај декомпајлирања са циљем реконструкције на језик високог нивоа би довео до генерисања много компликованијег и неразумљивијег кода. Посебне варијације овог типа трансформације над међујезицима се могу пронаћи у раду [110].

Трансформације над петљама (енг. *loop transformation*)

Постоје различити облици трансформација [107] који се могу применити на петље следећег облика:

$$\begin{aligned} &\textbf{while } (condition) \\ &\{\ body\ \} \end{aligned} \qquad (3.3.2.3.1)$$

Једна од таквих трансформација подразумева промену улазне променљиве, користећи методе шифровања:

$$\begin{aligned} &i = 0; \\ &\textbf{while } (i < N) \\ &\{\ldots i \ldots \\ &\quad i = i + 1;\} \end{aligned} \qquad (3.3.2.3.2)$$

користећи шифровање $\lambda\, x.\,(2x + 1)$ петљу трансформишемо у облик:

$$\begin{aligned} &i = 1; \\ &\textbf{while } (i < 2 * N + 1) \\ &\{\ldots ((i - 1)\ /\ 2) \ldots \end{aligned}$$

$$i = i + 2; \quad \} \tag{3.3.2.3.3}$$

Други облик трансформација подразумева додавање лажних улазних променљивих како би се успоставила компликованија петља. На пример, за петљу (3.3.2.3.4):

$$\begin{aligned} &i = 0; \\ &\textbf{while } (i < 10) \\ &\{ \dots i \dots \\ &\quad i = i + 1; \} \end{aligned} \tag{3.3.2.3.4}$$

под претпоставком да се вредност i мења једино на крају петље, може се додати променљива j на следећи начин:

$$\begin{aligned} &i = 0; \\ &j = 0; \\ &\textbf{while } \big((i < 10)\&\&(j < 120)\big) \\ &\{ \quad \dots \\ &\quad i = i + 1; \\ &\quad j = j + 2 * i; \} \end{aligned} \tag{3.3.2.3.5}$$

Под претпоставком да се i и j мењају једино на крају петље, лако се примећује да је $i = 10$ услов за излазак као и раније, с тим да је након трансформације цео израз сложенији. Израз $(j < 120)$ у овој ситуацији представља предикат, јер се зна да је његова вредност увек тачна (*true*) током целог извршавања петље. Поред споменуте трансформације, аутори у раду [111] предлажу додавање лажних променљивих са циљем креирања већег степена зависности између улазних параметара. Овај вид заштите је познат као обфускација методом сечења (енг. *slicing obfuscation*). Поред споменутих, остали облици трансформација над петљама укључују дељење, где се једна петља дели на више појединачних, блокирање, промена стања, итд.

Равнање тока извршавања (енг. *control flow flattening*)

Идеја овог облика трансформације, објашњена у раду [112], заснива се на уклањању конструктора токова извршавања као што су *while* петље тако да сви блокови графа извршавања имају свог претходника и следбеника. На пример, за код из израза (3.3.2.4.1) [107]:

$$\begin{aligned} &\text{init;} \\ &\textbf{while } (condition) \\ &\{ \text{ loop\_body } \} \end{aligned} \tag{3.3.2.4.1}$$

У наведеном примеру, петља **while** се мења са **switch** блоком. Изглед након трансформације је приказан изразом (3.3.2.4.2):

$$\begin{aligned} &var = 1; \\ &\textbf{switch}(var) \end{aligned}$$

```
{     case 1:
          init; var = 2;
          break;
      case 2:
          if (condition)
          {...
              var = 3;
          ...}
          else
          {...
              var = 4;
          ...}
          break;
      case 3:
          loop_body; var = 2;
          break;
      case 4:
          var = 1;
      end;}
```
(3.3.2.4.2)

Променљива var се у овом примеру понаша као контролна променљива, јер прати извршавање кода у свакој тачки приликом додељивања вредности. С обзиром да је реконструкција оваквог вида трансформације (switch ⇒ while) лако изводљива, аутори у раду [112] предлажу додатне нивое заштите као што су додавање ирационалних скокова, додатних глобалних променљивих у виду низова, подела већих блокова на мање као и додавање показивача. Више о овом типу заштите се може пронаћи у [113].

Трансформације над методама (енг. *method transformations*)

Већина споменутих трансформација су прилично локализоване и утичу углавном на једну методу. У раду [37] аутор описује неке од најчешће коришћених облика трансформација над методама:

- *Inline methods* – овај вид трансформације подразумева замену позива методе са њеним телом, односно кодом,
- *Outline methods* – овај вид трансформације подразумева замену дела кода позивом методе,
- *Clone methods* – овај вид трансформације подразумева креирање вишеструких копија једне те исте методе применом различитих техника обфускације.

Избор позивајуће клон методе се врши у току извршавања програма у самој тачки позива,

- *Interleave methods* – овај вид трансформације се примењује код спојених метода, тј. након трансформације спајања две или више метода у једну. Трансформација подразумева увођење параметара у позиве оригиналних метода на основу којих се прави разлика која од спојених метода ће бити извршена.

3.3.3 Обфускација података

Идеја ове технике садржи се у измени података програма над којима се обављају операције као и њиховим структурама. У овом поглављу ће бити наведене особине и својства најчешће коришћених трансформација за обфускацију података. Овај облик заштите подразумева следеће трансформације:

- шифровање променљивих (енг. *variable encoding*) [114, 115],
- трансформације спајања и дељења (енг. *merging and splitting*) [94, 115, 107],
- трансформације низова (енг. *array transformations*) [94, 88, 107],
    - спајање низова
    - дељење низа,
- премештање регистара (енг. *register realignment*),
- уградња мртвог/лажног кода (енг. *dead code insertion*),
- шифровање ресурса (енг. *resource encryption*),
- транспозиција кода (енг. *code transposition*),
- виртуелизација и шифровање кода (енг. *virtualization and code encryption*).

Шифровање променљивих (енг. *variable encoding*)

Идеја овог вида заштите заснива се на шифровању (трансформацији) анализиране променљиве у израз.

Пример:

$$i \Rightarrow a * i + b \qquad (3.3.3.1.1)$$

где су a и b константе. Шифровање треба да буде такво да се у сваком тренутку, по потреби може реконструисати оригинална вредност. Под шифровањем, подразумева се јасна трансформација параметра i (и као дефиниције и као употребне променљиве). Ова трансформација захтева обфускацију дефиниције и употребе саме променљиве, као што је дато у примеру (3.3.3.1.2).

$$i = 2 \implies i = a * 2 + b \qquad j = i + 1 \implies j = \frac{i-b}{a} + 1 \qquad (3.3.3.1.2)$$

Израз као што је $i++$ представља и дефиницију и употребу. Применом трансформације из израза (3.3.3.1.1), добија се нова вредност, приказана изразом (3.3.3.1.3).

$$i++ \Longrightarrow \left(a * \frac{i-b}{a} + 1\right) + b \qquad (3.3.3.1.3)$$

Компликованија варијанта овог типа обфускације параметара, подразумева трансформацију две или више зависне променљиве. У том случају, неопходно је обезбедити лак приступ оригиналним вредностима што често представља проблем.

Приликом употребе овог облика обфускације, потребно је обратити пажњу на оптерећеност вредности анализираних променљивих са аспекта фреквенције коришћења како не би дошло до успорења, тј. повећања времена извршавања самог кода због честих аритметичких операција. Више о овом типу заштите се може прочитати у [114, 115].

Трансформације спајања и дељења (енг. *merging and splitting*)

Спајање две променљиве [107] x и y је дато изразом (3.3.3.2.1), при чему је неопходно да буду задовољени следећи услови:

- $0 \leq x < N$
- $y \geq 0$

$$z = N * y + x \qquad (3.3.3.2.1)$$

Као што можемо спајати једну или више променљивих, исто тако можемо једну променљиву поделити на две или више. Колберг је у раду [94] показао како се променљива типа *bool* може поделити, док је Драпе у раду [115] дао пример (3.3.3.2.2) како поделити променљиву x типа *integer* на две променљиве a и b тако да важи:

$$a = x\ div\ 10 \quad \text{и} \quad b = x\ mod\ 10 \qquad (3.3.3.2.2)$$

Пример коришћења трансформације из формуле (3.3.3.2.2), за израз $x++$, приказана је формулама (3.3.3.2.3) и (3.3.3.2.4).

$$a = (10 * a + b + 1)\ div\ 10; \qquad (3.3.3.2.3)$$

$$b = (b + 1)\ mod\ 10; \qquad (3.3.3.2.4)$$

Трансформације низова (енг. *array transformations*)

Као што штитимо једну променљиву, исто тако можемо извршити обфускационе трансформације над низовима, с тим да се пре било каквих акција треба утврдити да ли низови задовољавају одређене услове, тј. да не садрже изузетке, да се не прослеђују другим методама као параметри, итд.

Један од најједноставнијих начина обфускације низова представља измена индекса елемената. Оваква трансформација се реализује или шифровањем елемената низа или дефинисањем пермутација. Најпопуларније трансформације над низовима су проширење (енг. *folding*) и смањење (енг. *flattening*) о којима ће више бити речи у наставку овог поглавља. Практично гледано, проширење низа подразумева трансформацију једнодимензионалног низа величине $m \times n$ у дводимензионални низ величине $[m, n]$. На исти начин се n-димензионални низ може смањити у

једнодимензионални. Наведене трансформације се реализују применом трансформација за спајање и дељење променљивих, што је објашњено у претходном поглављу.

Спајање низова

Пример дељења низа, приказан је изразом (3.3.3.3.1) [107]:

$$\begin{array}{lcl} \textbf{int}\ [\,]\ \mathrm{A} = \textbf{new int}\ [10]; & & \textbf{int}\ [\,]\ \mathrm{A1} = \textbf{new int}\ [5]; \\ & & \textbf{int}\ [\,]\ \mathrm{A2} = \textbf{new int}\ [5]; \\ \ldots & \Longrightarrow & \ldots 8 \\ A[i] = \cdots. & & \textbf{if}\ \big((i\ \%\ 2) == 0\big)\ A1\left[\frac{i}{2}\right] = \cdots; \\ & & \textbf{else}\ A2\left[\frac{i}{2}\right] = \cdots; \end{array} \qquad (3.3.3.3.1)$$

Рад [68] даје преглед генерализоване верзије ове трансформације као и њене вишеструке примене. Подела низа A димензије n на два низа B_1 и B_2, димензија m_1 и m_2 где је $(m_1 + m_2 \geq n)$ може бити означено дефинисањем три функције на следећи начин [107]:

$$ch : [0..n) \longrightarrow \mathbb{B} \qquad (3.3.3.3.2)$$

$$f_1 : [0..n) \longrightarrow [0..m_1) \qquad (3.3.3.3.3)$$

$$f_2 : [0..n) \longrightarrow [0..m_2) \qquad (3.3.3.3.4)$$

ch - функција која проверава могућност поделе низа и задужена је за сам процес поделе, односно осигурава да новодобијени низови буду приближно исте величине.

f_1 и f_2 – инјективне функције (сви елементи оригиналног низа се пресликавају на различите кодомене), новоформираних низова, приближно истих величина.

Веза између низова A, B_1 и B_2 дефинисана следећим правилом:

$$A[i] = \begin{cases} B_1[f_1(i)] & \textbf{if}\ ch(i)\ \textbf{is}\ true \\ B_2[f_2(i)] & otherwise \end{cases} \qquad (3.3.3.3.5)$$

Веза између оригиналног низа и два новокреирана се може генерализовати тако да се низ A може поделити на више од 2 низа. Више о овој генерализацији се може пронаћи у [112] где аутор уводи нове типове података као што су листе и матрице на основу којих се добија већа могућност за примену обфускационих техника.

Дељење низа

Код технике спајања низова је неопходно обратити пажњу на одрживост редоследа елемената у новом низу. Претпоставимо да имамо низ B_1 димензије m_1, низ B_2 димензије m_2 и новодобијени низ A димензије $m_1 + m_2$. Веза између ових низова се дефинише на следећи начин:

$$A[i] = \begin{cases} B_1[(i)] & \textbf{if } i < m_1 \\ B_2[i - m_1] & \textbf{if } i \geq m_1 \end{cases} \quad (3.3.3.3.6)$$

Ова трансформација је аналогна конкатенацији две листе.

Остали облици обфускације података

У овом поглављу су наведене додатне технике обфускације које се често користе у синергији са неком од горе споменутих метода.

- **промоција промењљивих** – техника која подразумева трансформацију локалних промењљивих у објекте и на тај начин омогућава поновно коришћење у узајамно независним методама.
- **измена века трајања промењљивих** – идеја ове трансформације је промена локалних промењљивих у глобалне како би стално биле доступне током извршавања кода.
- **уградња мртвог/лажног кода (енг. *dead code insertion*)** – ова техника се најчешће користи приликом скривања тока извршавања и њена идеја је да увећа комплексност анализираног прогама чиме би отежао процес статичке анализе у циљу инверзног инжењеринга. Припада једноставнијим техникама обфускације и заснива се на уградњи нефункционалних инструкција. Пример уградње лажног кода приказан је у табелама 3.1 и 3.2.

Табела 3.1 Пример уградње лажног кода (регистри)

Инструкција	Значење
ADD Reg, 0	Reg ← Reg + 0
MOV Reg, Reg	Reg ← Reg
OR Reg, 0	Reg ← Reg | 0
AND Reg, -1	Reg ← Reg & -1

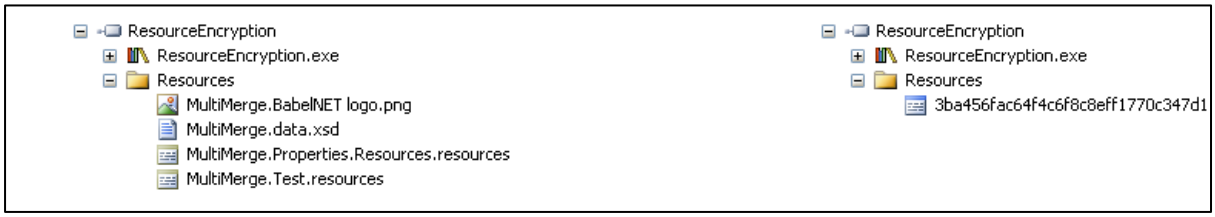
Табела 3.2 Уградња лажног кода (регистри и стек)

Инструкција	Значење
SUB CX, 2 **INC CX** **INC CX**	Вредност регистра CX се умањује за 2, након чега се двоструким позивом команде *inc* његова вредност увећава за по 1, са циљем одржања исте функционалности.
PUSH CX **POP CX**	Вредност регистра остаје непромењена, јер се вредност прво шаље на стек па враћа са стека.

Са леве стране табеле 3.1 налазе се инструкције чији је семантички еквивалент једнак инструкцији *NOP* (енг. *no operation*) која нема никакву функционалност сем повећања кашњења, отежане анализе, драматичног повећања обфускованог кода, док је конкретно

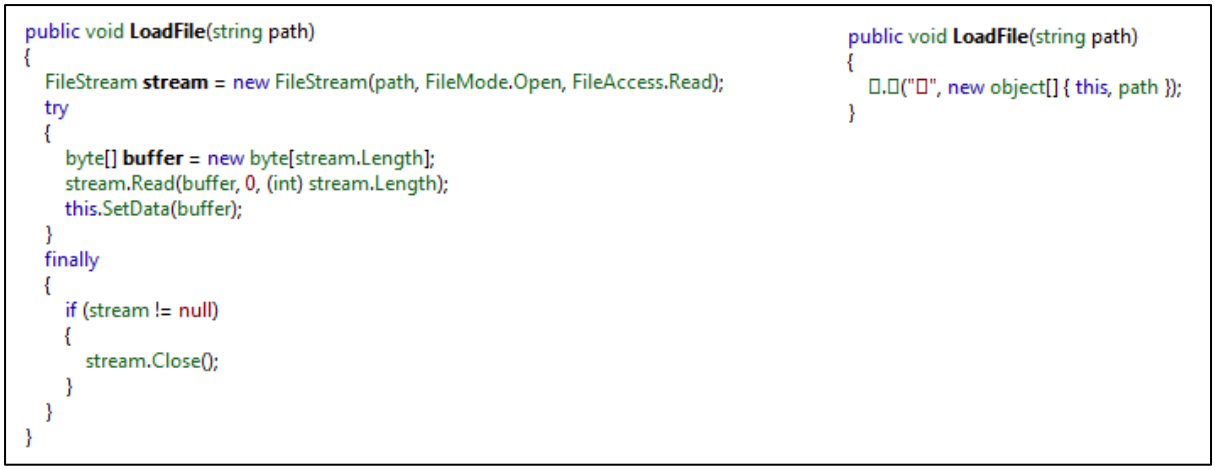
значење приказано у десној колони. Овај тип обфускације је детаљније описан у радовима [116 - 120].

- **премештање регистара (енг. *register realignment*)** – представља једноставну технику обфускације, засновану на промени регистара из генерације у генерацију при чему оригинални код остаје функционално непромењен. Сврха ове трансформације јесте онемогућавање антивирусне анализе, засноване на провери подударности потписа кода.
- **транспозиција кода (енг. *code transposition*)** – представља технику обфускације, засновану на измени редоследа блокова инструкција оригиналног кода без утицаја на функционалност. Заснива се на проналажењу независних делова кода и њихову међусобну размену. Постоје два приступа имплементације овог типа заштите. Први приступ је базиран на уградњи безусловних команди, помоћу којих се врши повезивање блоковских делова кода, док се други приступ заснива на размени независних инструкција и њиховој замени новим.
- **замена инструкција (енг. *instruction substitution*)** – представља технику обфускације, засновану на замени инструкција својим еквивалентима. Овај вид заштите има велики утицај на потпис кода, тешко га је ревидирати, посебно у случају када претходно наведена библиотека није позната.
- **шифровање ресурса (енг. *resource encryption*)** – представља технику заштите управљивих ресурса компримовањем и шифровањем. Ресурси се учитавају у току извршавања апликација, тј. само онда када се користе. Једноставан пример приказан је на слици 3.4. Са леве стране налази се оригинална библиотека, док је са десне приказан изглед шифрованих ресурса.



Слика 3.4 Шифровање ресурса: оригинал (лево), обфускован код (десно) [102]

- **виртуелизација и шифровање кода (енг. *virtualization and code encryption*)** – представља технику шифровања и виртуелизације бајт кода метода на такав начин да постају непрепознатљиве алатима за инверзно превођење кода (у даљем тексту, декомпајлерима). Заштићене методе се извршавају у тренутку коришћења (енг. *runtime*) унутар виртуелних машина. Пример овог типа заштите приказан је на слици 3.5. Са леве стране, налази се оригнални, док са десне приказана верзија шифрованог кода.



Слика 3.5 Шифровање кода: оригинал (лево), обфускован код (десно) [102]

3.4 Предности и мане

Главна предност обфускације јесте ниска цена и флексибилност саме технике. Предности:

- **заштита од напада** – техника штити од статичких и динамичких облика напада, и као резултат тога, злонамерним корисницима је потребно знатно више времена и ресурса како би постигли жељени циљ.
- **разноликост** – подразумева могућност стварања различитих верзија оригиналне верзије.
- **ниски трошкови** – техника подразумева ниске трошкове одржавања због аутоматизације процеса трансформације и компатибилности са постојећим системима.
- **платформска независност** – обфускационе трансформације се могу применити на кодове високог нивоа апстракције како би се одржала независност од платформе на којој се апликација извршава.

Мане:

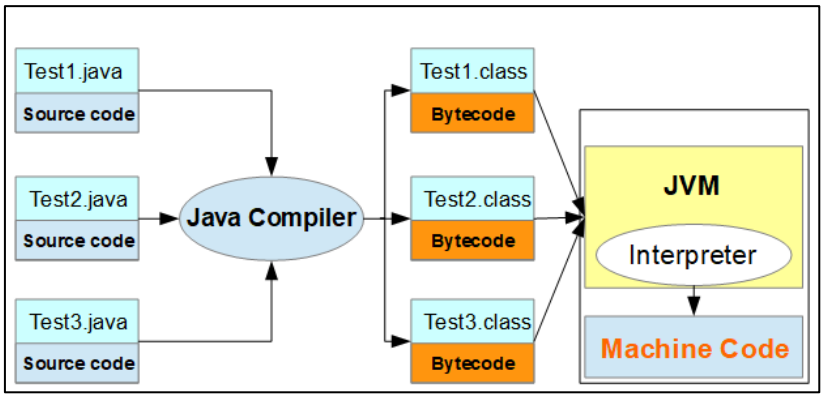
- **перформансе** – свака трансформација представља додатни трошак са аспекта утрошка меморије и времена потребног за извршење обфускованог кода.
- **савршена сигурност** – обфускационе трансформације не пружају апсолутну сигурност интелектуалне својине која инверзни инжењеринг чини немогућим. Слично енкрипцији, која у пракси никада не гарантује потпуну заштиту, тако ни обфускација не гарантује да алгоритам или изворни код никада неће бити откривени.
- **енергетска ефикасност** – у овој докторској дисертацији је показано да различите обфускационе трансформације значајно утичу на енергетску ефикасност самог кода.

4. Архитектура генерисања профила потрошње и концепт софтверског инверзног инжењеринга

4.1 Концепти анализираних платформи за развој управљивог кода

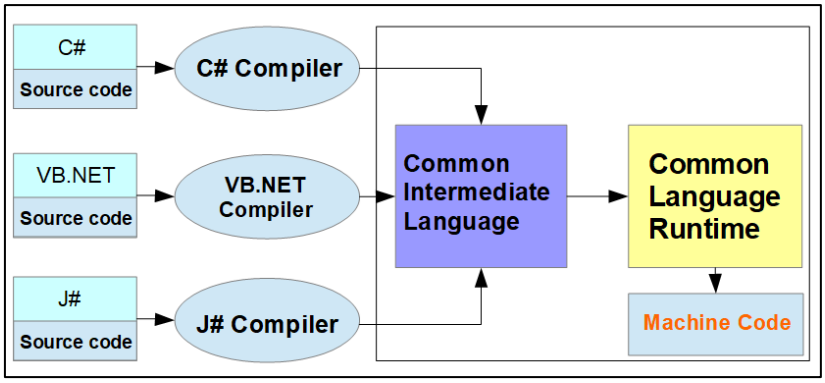
У овом поглављу, биће анализирани процеси превођења кода за програмске језике који се базирају на виртуелним машинама са акцентом на *.NET* платформу, коришћену у овој дисертацији.

Данашње платформе за развој софтвера користе програмске језике где се оригинални код преводи/компајлира у такозвани међујезик (енг. *intermediate language*). Управо оваква архитектура, односно платформа за овакве програмске језике даје могућност корисницима да помоћу специјализованих алата изврше инверзно превођење кода (енг. *decompiling*), како би дошли до изворног кода. Софтверски пакети написани у програмском језику Јава и *.NET* платформи омогућавају горе споменути ризик с обзиром да међујезик не представља бинарну репрезентацију самог програма. На слици 4.1 приказан је процес превођења оригиналног програмског кода за програмски језик Јава. Интерпретер преводи инструкције из међујезика у машински код, уз одређене оптимизације и верификације над самим асемблерским мета подацима.



Слика 4.1 Процес превођења за Јава програмски језик

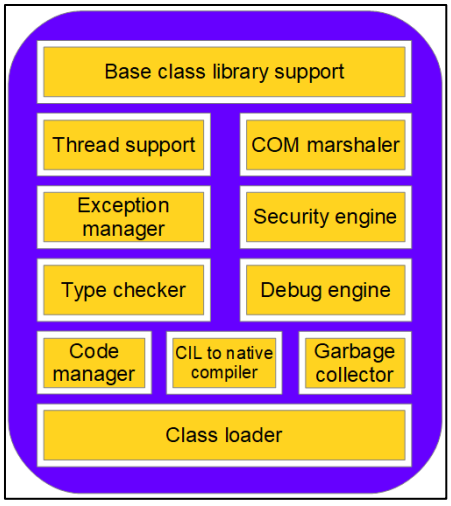
Што се *.NET* платформе тиче, концепт је врло сличан и приказан је на слици 4.2.



Слика 4.2 Процес превођења за .NET платформу

Процес превођења и интерпретације кода написаног за *.NET* платформу почиње превођењем изворног кода у неутрални међујезик *CIL* (енг. *Common Intermediate Language*), да би се потом у *CLR* (енг. *Common Language Runtime*) компоненти исти превео у бинарну репрезентацију, тј. машински код и као такав био спреман за извршавање.

У току извршавања програма, интерпретеру су потребне информације о именима класа, метода, атрибута, итд., односно сви подаци који се налазе у међујезику, што уједно представља и проблем, јер оставља могућност инверзног превођења, тј. добијања оригиналног кода. На овај начин, неовлашћени корисници имају прилику да искористе интелектуалну својину [121]. Концептуални приказ *CLR* компоненте је дат на слици 4.3, док је свака од њих описана касније у овом поглављу.



Слика 4.3 CLR (Common Language Runtime)

У *.NET*-у, *CLR* представља срце самог окружења без којег се не може замислити рад платформе. Да би се софтвер написан у *.NET* окружењу успешно покренуо, неопходно је извршити његово превођење у бинарну репрезентацију. Овде је битно разумети да сваки рачунарски систем може имати различиту архитектуру, као и сам оперативни систем који се налази на њему. Да би се софтвер успешно извршавао, независно од оперативног система, сам код мора бити преведен у изворни облик за шта

је управо задужен *CLR*. Као улаз, прихвата изворни код програма, компајлира и преводи у *MSIL* (*Microsoft Intermediate Language*) који се састоји од *CPU* независног кода и инструкција које не зависе од платформе на којој се извршава.

Компоненте *CLR*-а:

- *class loader* – компонента која служи за учитавање класа у *CLR*,
- *CIL to native compiler* – компонента која преводи *MSIL* у машински код,
- *code manager* – компонента која управља кодом током извршавања,
- *garbage collector* – компонента која управља меморијом,
- *security engine* – компонента за управљање системом безбедности,
- *type checker* – компонента за проверу типа променљивих података,
- *thread support* – компонента која омогућава вишенитну подршку апликацијама,
- *COM marshaler* – компонента која омогућава размену података између *.NET* и *COM* компоненти,
- *debug engine* – компонента која дозвољава развојним инжењерима да откривају и исправљају грешке (дебагују) за различите типове апликација,
- *exception manager* – компонента која даје могућност руковања изузецима у току извршавања,
- *base class library support* – компонента која омогућава коришћење свих неопходних апликација у току извршавања. Представља скуп типова које нуди *.NET* и састоји се из интерфејса, класа као и вредносних типова који омогућавају коришћење функционалности система.

CLR је такође надлежан за активирање објеката као и извршавање безбедносних провера над њима. Процес превођења/компајлирања изворног кода у машински се састоји из два дела:

- превођење изворног кода у *MSIL*,
- превођење *MSIL* кода у конкретан платформски језик који извршава *CLR*.

MSIL представља синтаксно једноставнији језик нижег апстракционог нивоа, који се брзо преводи у машинску репрезентацију задржавајући све особине објектно оријентисаног језика (полиморфизам, наслеђивање, апстракцију података), као и концепте као што су изузеци и догађаји. Захваљујући овом језику омогућена је:

- **платформска независност**,
- **боље перформансе** – за разлику од Јаве, чији се код интерпретира тј. директно преводи и извршава, овде не постоји губитак перформанси који се јавља услед интерпретације. Преводе се само позвани сегменти кода, а не цела апликација.
- **језичка интероперабилност** – омогућава директну комуникацију међу класама написаним у различитим програмским језицима. Захваљујући томе,

могуће је остварити везу између метода које се налазе у различитим објектима, писаним у различитим програмским језицима.

- **разликовање референтних и вредносних типова** – код референтних типова података, променљива садржи адресу где се налазе подаци, који се уједно чувају у делу меморије који се назива контролисани хип (енг. *heap*). Код вредносних типова, променљиве чувају своје податке на стеку.

4.2 Инверзни инжењеринг софтвера

Софтвер је једна од најсложенијих и најинтригантнијих технологија данашњице. Сама идеја је присутна одавно, још од времена индустријске револуције, далеко пре развоја рачунара и модерних технологија. Баш као софтверско инжењерство, софтверски инверзни инжењеринг је чист виртуелни процес који подразумева поседовање/знање разних вештина развоја рачунара и софтвера. Инверзни инжењеринг софтвера (даље у тексу *SIE*) обједињује неколико "уметности": разбијање кода, решавање загонетки, програмирање и логичку анализу. Често је зависан од платформе (енг. *platform specific*) и на њега утиче сам оперативни систем као и коришћена хардверска платформа. У овом раду, коришћен је *Microsoft Windows 8* оперативни систем из разлога што је то најзаступљенија платофрма, као и чињеница да представља најпопуларније окружење за процесе инверзног инжењеринга.

Инверзни инжењеринг представља процес анализе неког система, препознавања појединачних компоненти истог, као и њихових међусобних односа са циљем комплетне репродукције. У области софтверског инжењеринга, овај процес се везује за анализу програма [122], где се покушава статистички предвидети (апроксимирати) понашање софтвера и његових доступних стања. Углавном се користи за подршку оптимизације компајлера, аутоматску верификацију програма и за помоћ у истраживању сигурности, која је делимично обрађена у овој докторској дисертацији.

Концептуално, *SIE* представља процес откривања техничко-технолошких принципа производа или система, базирајући се на анализи његове структуре, функција и операција. Другим речима, то је дедуктивни процес развоја система до највишег нивоа апстракције почевши од крајњег производа.

Инверзни софтверски инжењеринг захтева познавање разноврсних алата за мониторисање, истраживање, откривање и остале трансформационе облике који омогућавају да неки програм буде преведен у облик који је разумљив људском оку. Већина алата приказује информације које оперативни систем прикупља о анализираној апликацији и њеном спољном окружењу, јер је управо то комуникациони канал између програма и спољног утицаја. У овој докторској дисертацији, коришћени су алати за демонтажу (енг. *disassembling*), анализу (у даљем тексту дебагери) (енг. *debugging*) и инверзно превођење/декомпајлирање (енг. *decompiling*).

4.2.1 Инверзно превођење\декомпајлирање (енг. *decompiling*)

Декомпајлери служе за претварање извршног бинарног кода у изворни код неког од језика вишег нивоа апстракције као што су Јава, C++, C#, итд. Декомпајлери могу превести код само за специфичан језик, тј. за који је сам алат развијен, односно, није могуће декомпајлером за Јаву декомпајлирати код писан у C++. Њихова предност је већа заступљеност због рада са језицима вишег нивоа апстракције који су далеко лакши и разумљивији крајњим корисницима.

4.2.2 Демонтажа (енг. *disassembling*)

Демонтажа и декомпајлирање представљају поступке који се користе приликом инверзног инжењеринга када је потребно, у циљу анализе, од извршног добити изворни код. Конкретно, демонтажом се извршни бинарни код преводи у асемблерски. С обзиром да је асемблер језик који се директно преводи у бинарни (једна асемблерска инструкција одговара једној бинарној), помоћу ових алата се може реализовати инверзна операција било које апликације, без обзира на програмски језик у коме је написана.

Постоје два приступа за демонтажу извршних датотека: статички (који је коришћен у овом раду) и динамички. Код статичког приступа се извршни фајл демонтира/анализира у стању мировања/неизвршавања док се у динамичком приступу анализа реализује за време извршавања датотеке над одређеним улазним подацима. Предност статичке демонтаже је могућност анализе целе датотеке одједном док је код динамичке та могућност ограничена само на делове кода који се извршавају у том моменту. Друга предност статичке демонтаже је време неопходно за анализу које је пропорционално величини програма, док је код динамичке анализе то време дефинисано величином броја инструкција.

4.2.3 Анализа\дебаговање

Основна идеја која се налази иза дебагера је да програмери не могу да предвиде све што њихов програм може да уради. Односно, програми су обично превише сложени за човека да предвиди сваки појединачни потенцијални исход. Дебагер је алат који дозвољава развојним инжењерима да прате стање програма који анализирају у реалном времену (корак по корак) као и способност одређивања тачке прекида (енг. *breakpoint*).

4.2.4 Типови инверзног инжењеринга софтвера

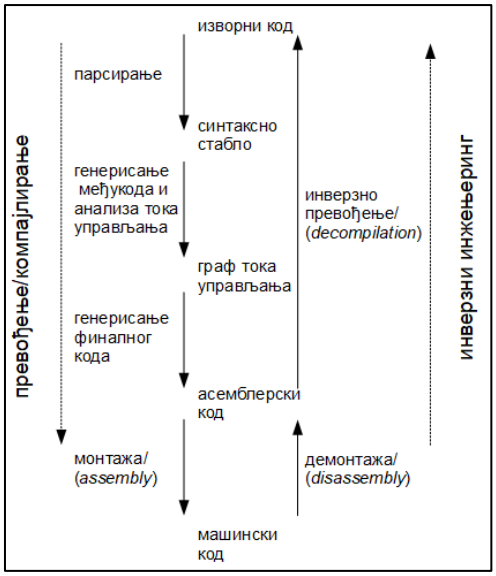
Алгоритамски посматрано, постоје 2 различита приступа у *SIE* - у:

- **динамичка анализа** - динамички приступ подразумева анализу бинарне датотеке у тренутку њеног извршавања. Овај вид инверзног инжењеринга се најчешће изводи или уз помоћ емулатора или на виртуелној машини, како не би дошло до компромитовања физичке машине у случају анализе

малициозних апликација. Са друге стране, коришћење емулиране или виртуелизоване машине нуди боље могућности за контролу и надгледање. Алати са којима се изводи овај облик анализе се називају дебагерима (енг. debugger) а најпознатији су *OllyDbg* [123], *Immunity Debugger* [124] и *WinDbg* [125]. Они постављају додатни слој интеракције између оперативног система и анализиране апликације, што омогућава праћење извршавања кода корак по корак, са могућношћу паузирања и потребне анализе/модификације меморије као и стања самог процесора.

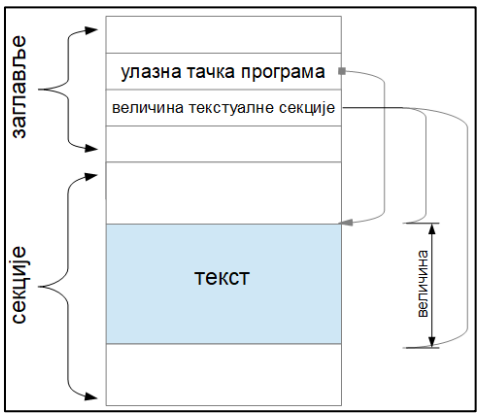
- **статичка анализа** – покушава да моделује сва могућа понашања програма и према томе, заснива се на високом нивоу апстракције. Програм који се анализира се не извршава, већ се процес изводи апстрактно, углавном коришћењем графова тока извршавања, тј. *CFG* (енг. *control flow graph*), графова програмске зависности, тј. *PDA* (енг. *program dependency graph*) и графова позива *CG* (енг. *call graph*). Овај тип анализе је детаљан, сигуран, покрива сва стања програма, али је временски захтеван и може се суочити са изазовима у случају заштићених бинарних датотека. Пре било какве статичке анализе, потребно је извршити процес демонтаже над бинарном датотеком како би се добиле валидне асемблерске инструкције и неформатирани региони података. Данас се користе два типа алгоритама за демонтажу: линеарни и рекурзивни преглед. Линеарни алгоритми анализирају бајт по бајт, представљају једноставну и брзу методу за демонтажу датотеке, али могу имати проблема приликом анализе обфускованих делова, што је у анализи тестних сценарија обрађених у овој дисертацији представљало непремостив проблем. Због наведеног, примењена је рекурзивна метода демонтаже бинарне датотеке, која је далеко мање подложна грешкама и заснива се на праћењу тока извршавања нарочито у ситуацијама када се наиђе на условне гране. У позадини алгоритма се делом извршава линеарни приступ, али у случају да се анализирају инструкције гране/скока, тада се користи рекурзивни приступ. Најквалитетнији алати за демонтажу засновани на рекурзивном приступу су *OllyDbg* і *IDAPro* [126], који су коришћени у овој докторској дисерацији.

Процес инверзног инжењеринга и компајлирања кода, приказан је на слици 4.4.



Слика 4.4 Процес компајлирања и инверзног инжењеринга

Датотека машинског кода састоји се из неколико различитих секција (секција за текст, секција за чување података који се само могу читати, адресна секција, итд) које садрже различите типове информација о програму, заједно са заглављем које описује све секције. Између осталог, заглавље садржи информације о улазној тачки програма, односно локацији у машинској датотеци где инструкције почињу са извршавањем (почетак програма), као и величини и броју инструкција [127], детаљније приказано на слици 4.5.



Слика 4.5 Структура извршне датотеке

Континуални напредак у области софтверског инжењерства задњих година донео је значајна побољшања у развоју алата за анализу софтверских пакета. Нажалост, иста технологија се неретко користи у инверзном инжењерингу са циљем проналажења рањивости као и крађе интелектуалне својине. Да би се споменуто реализовало, злонамерни корисници (хакери) морају да реконструишу интерну структуру програма како би били у могућности да идентификују делове кода од интереса. У циљу спречавања злоупотребе кода, користе се два приступа за решавање овог проблема. Први захтева константно одржавање кода у енкриптовној (заштићеној) форми док би се

декрипција реализовала по потреби током извршавања. Други приступ захтева коришћење специјализованих алата [128].

Поред своје ефикасности, овакви приступи имају и мане као што су утицај на перформансе и флексибилност. Алтернативни приступи су примене различитих обфускационих техника [129, 130, 131, 132] како би се повећао степен сигурности, са циљем повећања времена неопходног за анализу. Сходно томе, препорука је користити технике обфускације које значајно утичу на повећање степена отежаности статичке анализе као што је дато у [133, 134, 135].

Софтверски инверзни инжењеринг, са аспекта типова апликација на које се процес примењује, може поделити на две категорије [136 - 138]:

- **сигурносно оријентисане** (енг. *security related*)
  - **злонамеран софтвер** (енг. *malicious software*) – Са једне стране, творци злонамерног софтвера често користе принципе *SIE* – а, како би пронашли рањивости у оперативним системима и другим софтверским пакетима, а све у интересу "инфекције" истих. Са друге стране, творци антивирусних софтвера, такође применом *SIE* принципа, сецирају и анализирају сваки злонамеран програм са циљем идентификације степена инфекције, уклањања истог и као и превентивних акција спречавања инфицирања корисника.
  - **криптографски алгоритми** (енг. *cryptographic algorithms*)
  - **заштита дигиталних права** (енг. *digital rights management*)
- **развојно оријентисане** (енг. *development related*) – Инверзни инжењеринг може бити веома корисан за развојне инжењере, из више аспеката (провера квалитета коришћених библиотека, сазнавање драгоцених информација из конкурентског производа у циљу побољшања сопственог, процена робусности, итд.)

4.3 Генерисање профила потрошње

Концептуални приказ архитектуре процеса генерисања профила потрошње, приказан је на слици 4.6 и састоји се из следећих корака:

Korak 1. У оквиру модула "*Оригинална бинарна датотека*", извршена је припрема улазних података, односно тестних сценарија. У овој докторској дисертацији су анализирани извршни фајлови екстензије **"*.exe*"**, развијени у програмском језику *C#*.

Korak 2. У оквиру модула "*Демонтажа (disassembling) OllyDbg*", реализована је демонтажа оригиналног тестног сценарија, тј. бинарни код апликације је преведен у асемблерски и као такав био спреман за даљу обраду.

Korak 3. У оквиру модула "*Обфускација*", врши се заштита оригиналих тестних апликација употребом наведених комерцијалних алата за обфускацију кода, и то за следеће облике:

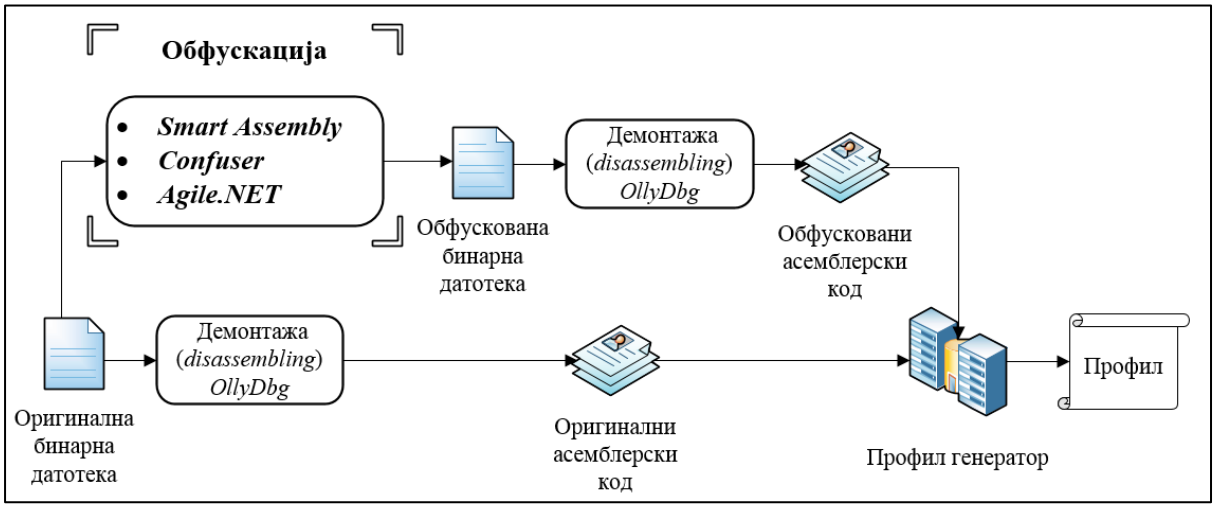
a. лексичка заштита,
b. заштита података,
c. заштита тока извршавања.

Korak 4. Модул "*Обфускована бинарна датотека*", представља заштићену тест апликацију, исте екстензије као и оригинал.

Korak 5. У овом кораку, обфускована тестна апликација пролази кроз процес демонтаже како би се добила нова асемблерска репрезентација односно модул "*Обфусковани асемблерски код*".

Korak 6. У овом кораку се врши поређење оригиналног и обфускованог асемблерског кода у специјално развијеној апликацији, на слици представљено као модул "*Профил генератор*". Детаљан опис саме апликације као и класни дијаграм, приказани су у прилогу 3.

Korak 7. У овом кораку се генеришу профили потрошње електричне енергије, који су детаљније обрађени у поглављу 5.



Слика 4.6 Генерисање профила потрошње

5 Приказ и дискусија резултата

Предложена архитектура за генерисање профила потрошње ради идентификације утицаја обфускационих техника заштите софтвера, верификована је над два тестна сценарија:

- *Тест 1*: множење матрица димензије 100×100,
- *Тест 2*: *quick sort* алгоритам за 10 000 елемената.

У овој глави, приказан је детаљан опис утицаја анализираних техника заштите кода методом обфускације и наведени су резултати за следеће категорије:

- **енергетска ефикасност** – под овим термином, реализовано је следеће:
  - *просечна струја по инструкцији (mA)* – представља идентификацију јачине струје за сваку извршену инструкцију, коришћењем вредности инструкционог модела из [31], који је дат прилогу 1. Вредност јачине просечна струје представља аритметичку средину свих извршених инструкција.
  - *енергија (mJ)* – подразумева израчунавање потрошње електричне енергије у *mJ* (мили-џулима). Начин на који је ово реализовано је објашњен у прилогу 2.
  - *профили потрошње* – графички репрезент функције јачине струје у времену. Профили су генерисани коришћењем алата "Профил Генератор" (слика 4.6), који је развијен у програмском језику C# и представља саставни део истраживачког рада ове дисертације (класни дијаграм приказан у прилогу 3). Улазни подаци за генератор број инструкција добијен применом демонтаже (објашњено у глави 4).
- **величина датотеке (KB)** – представља величину тестиране датотеке (фајла) у кило-бајтима, пре и после обфускације.
- **број асемблерских инструкција** – представља величину асемблерског инструкционог скупа (броја инструкција), пре и после обфускације.
- **време извршавања (s)** – представља време потребно да се изврши тестирана датотека (фајл) у секундама, пре и после обфускације.

Тестирање је спроведено за следеће обфускационе категорије:

- *лексичка заштита* (глава 5.1),
- *заштита тока извршавања* (глава 5.2),
- *заштита података* (глава 5.3),
- *комбинација лексичке и заштите тока извршавања* (глава 5.4),
- *комбинација лексичке, тока извршавања и заштите података* (глава 5.5).

Верификација је реализована коришћењем 3 софтверска пакета за измену разумљивости програма, чије су основне информације приказане у табели 5.1 док табела 5.2 даје детаљну спецификацију коришћеног рачунарског тест окружења.

Табела 5.1 Информације о коришћеним алатима

Алат	**Заштита**	**Цена**
Smart Assembly	Лексичка	**1565 $**
Agile.NET	Тока извршавања	**599 $**
Confuser	Података	**Бесплатан**

Табела 5.2 Спецификација тестног окружења

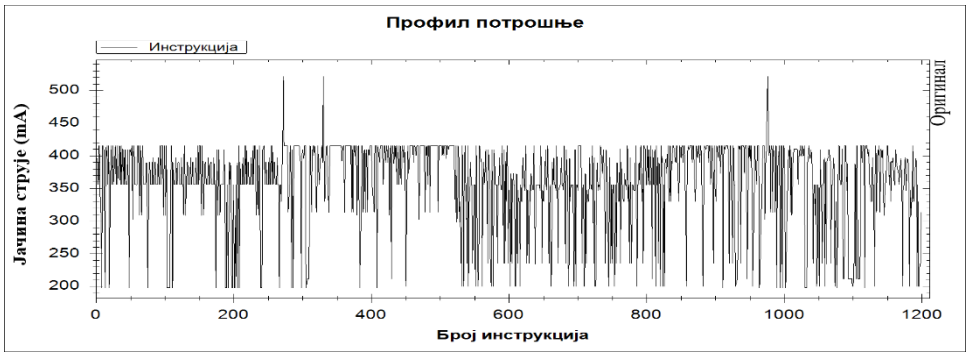
Компонента	**Опис**
Рачунар	**Процесор:** *Intel C2D E4700 (2.6GHz, 800MHz FSB/65W)* **Рам меморија:** *Kingston KTH-XW667LP/8GB (4GBx2) 667MHz* **Чврсти диск (HDD):** *WD Blue 500GB 7200RPM*
Развојно окружење	*Microsoft Visual Studio 2015 Professional Edition*
Компајлер	*Roslyn*

5.1 Лексичка заштита

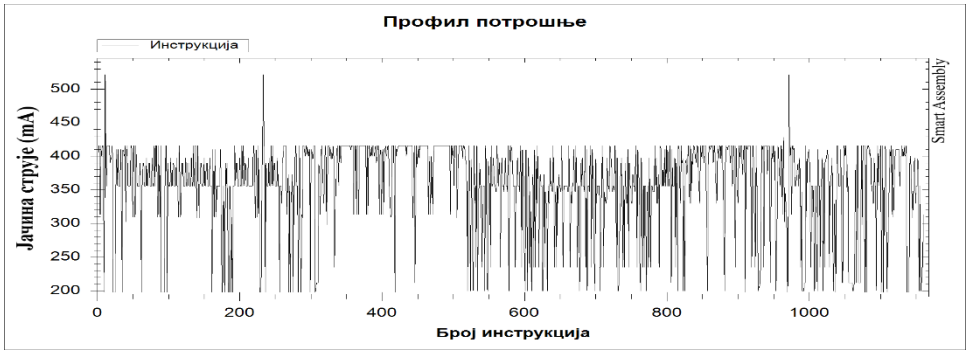
Лексичка обфускација подразумева неколико техника заштите (трансформација), док ће у овом поглављу бити приказани резултати за следеће четири:

- измена имена метода,
- измена имена типова атрибута,
- измена имена именских простора,
- измена имена параметара метода.

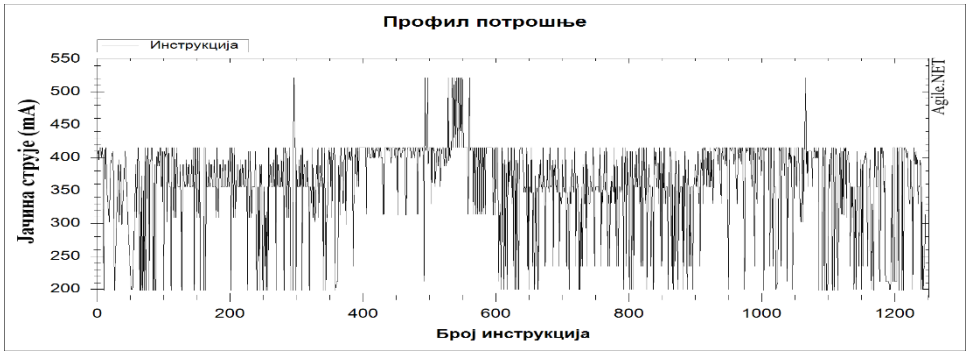
5.1.1 Множење матрица



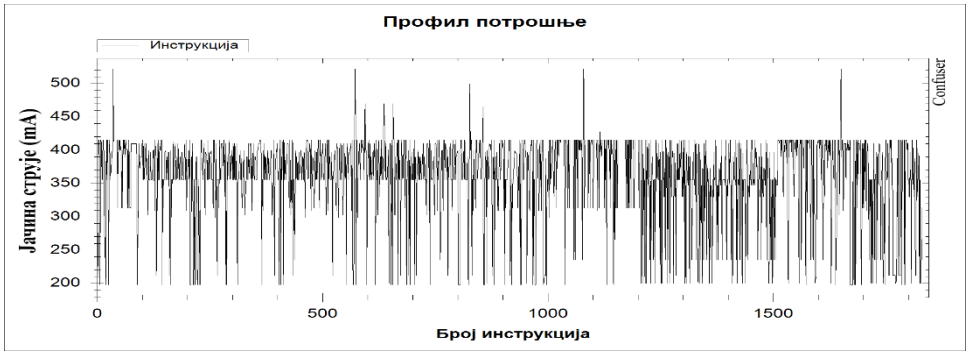
Слика 5.1 Профил потрошње немодификоване датотеке - оригинал



Слика 5.2 Лексичка обфускација - множење матрица - Smart Assembly



Слика 5.3 Лексичка обфускација - множење матрица - Agile.NET



Слика 5.4 Лексичка обфускација - множење матрица - Confuser

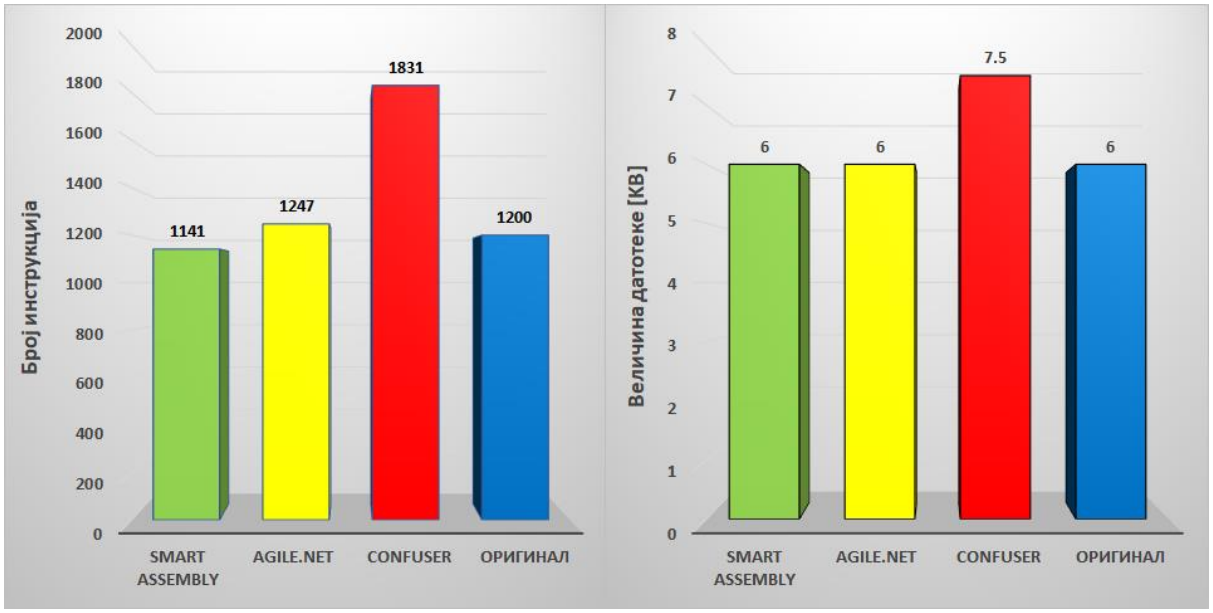
Табела 5.3 Лексичка обфускација - множење матрица

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Множење матрица	Број асемблерских инструкција	1200	Smart Assembly	1141	⇓ 5
			Agile.NET	1247	⇑ 1
			Confuser	1831	⇑ 52
	Величина датотеке (KB)	6	Smart Assembly	6	⇔
			Agile.NET	6	⇔
			Confuser	7.5	⇑ 25
	Јачина просечне струје по инструкцији (mA)	357.34	Smart Assembly	358.2	⇔
			Agile.NET	358.8	⇔
			Confuser	360.2	⇑ 1
	Време извршавања (s)	0.0137	Smart Assembly	0.0128	⇓ 7
			Agile.NET	0.0138	⇔
			Confuser	0.017	⇑ 24
	Енергија [mJ]	4.65	Smart Assembly	4.58	⇓ 1
			Agile.NET	4.95	⇑ 6
			Confuser	6.12	⇑ 31

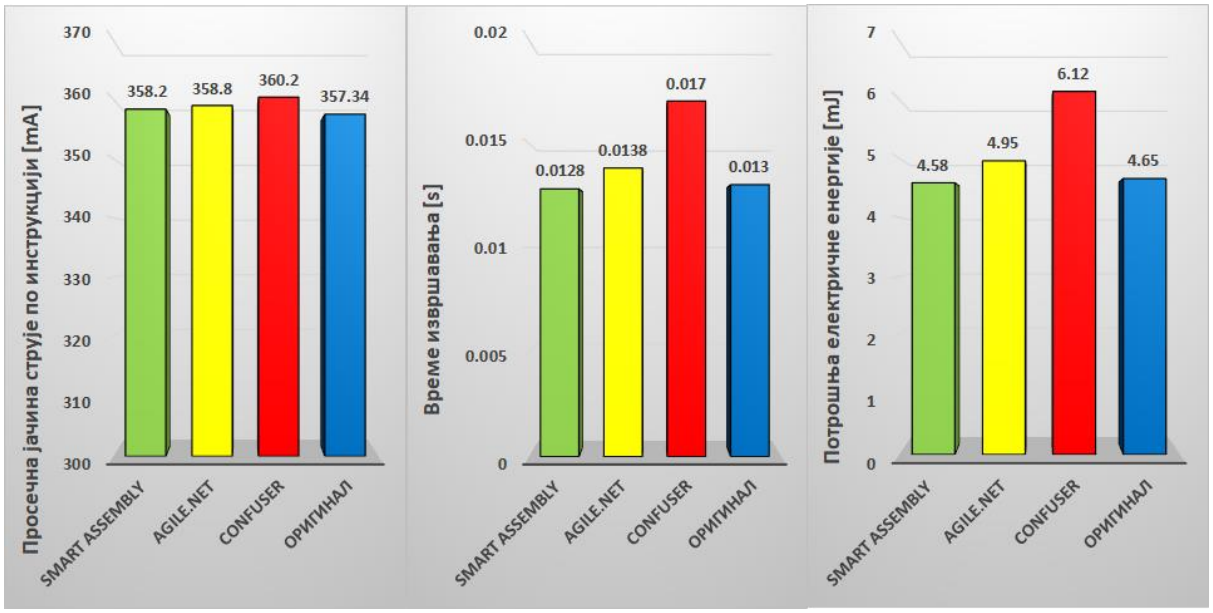
Упоредни приказ профила потрошње за тестиране алате у случају лексичке обфускације, за сценарио множења матрица приказан је на сликама 5.2, 5.3 и 5.4, док је оригинални профил потрошње дат на слици 5.1. На основу добијених резултата, обједињених у табели 5.3, може се закључити следеће:

- *Smart Assembly* – показује најбоље резултате у свим тестираним сегментима. Број извршених инструкција је за 5% мањи у односну на оригинал. Потрошња електричне енергије и време извршавања су оптимизовани у односу на оригинал.
- *Agile.NET* – показао за нијансу лошије резултате у скоро свим категоријама у односу на *Smart Assembly*, где се истиче повећање потрошње електричне енергије за 6%.
- *Confuser* – алат који је приказао знатно лошије перформансе у скоро свим катеогријама, што га класификује као најлоши алат у овом тестном сценарију. Разлог је већи број инструкција који резултира и повећањем потрошње електричне енергије за 31% у односу на оригинал.

Илустровани приказ измерених вредности из табеле 5.3, дат је на сликама 5.5 и 5.6.

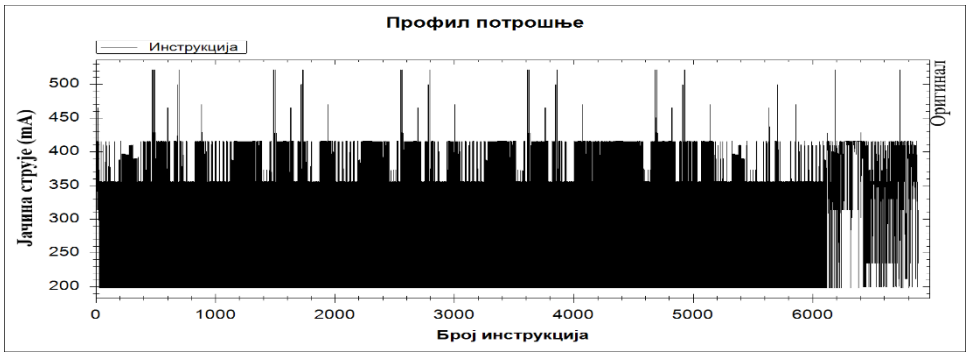


Слика 5.5 Графички приказ: број инструкција (лево), величина датотеке (десно)

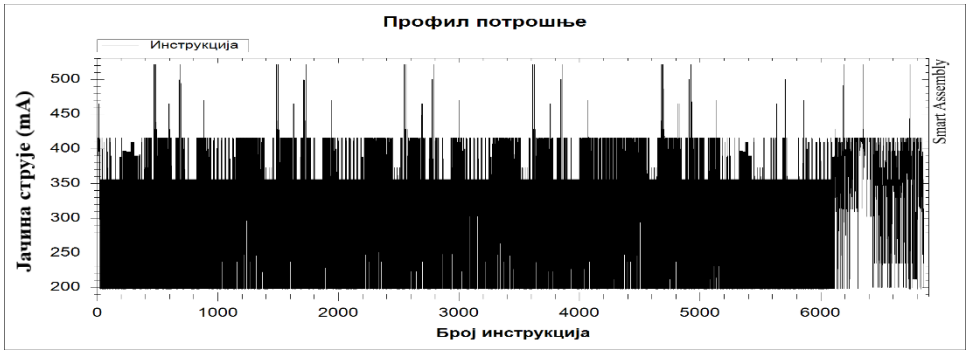


Слика 5.6 Графички приказ: јачина струје, време извршавања, потрошња електричне енергије

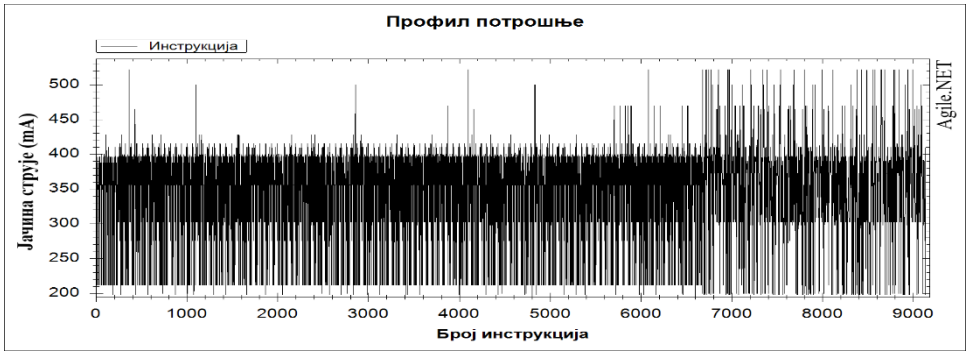
5.1.2 Quick sort алгоритам



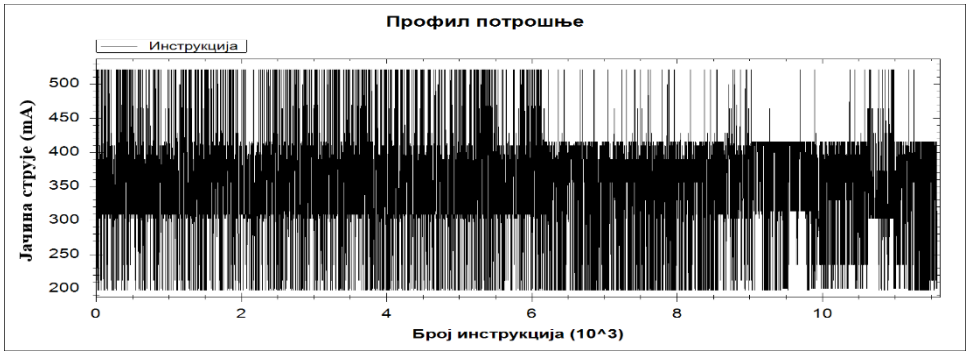
Слика 5.7 Профил потрошње немодификоване датотеке – оригинал



Слика 5.8 Лексичка обфускација - сортирање - Smart Assembly



Слика 5.9 Лексичка обфускација - сортирање - Agile.NET



Слика 5.10 Лексичка обфускација - сортирање - Confuser

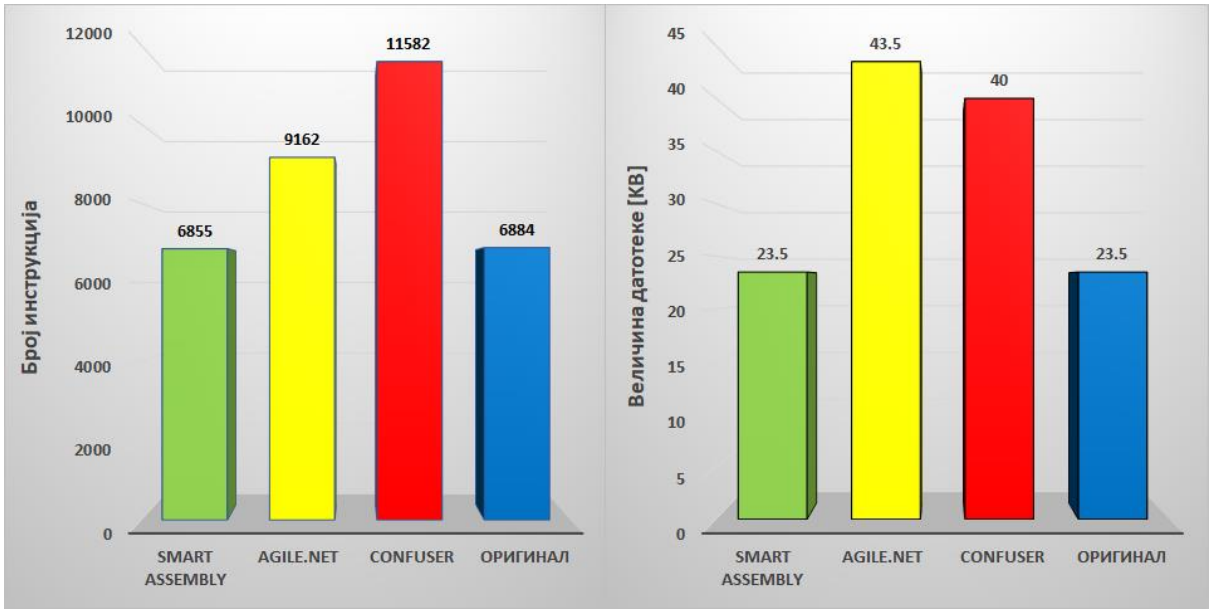
Табела 5.4 Лексичка обфускација - сортирање

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Quick Sort	Број асемблерских инструкција	6884	Smart Assembly	6855	⇓ 1
			Agile.NET	9162	⇑ 33
			Confuser	11582	⇑ 69
	Величина датотеке (KB)	23.5	Smart Assembly	23.5	⇔
			Agile.NET	43.5	⇑ 85
			Confuser	40	⇑ 70
	Јачина просечне струје по инструкцији (mA)	324.89	Smart Assembly	324.95	⇔
			Agile.NET	404.54	⇑ 24
			Confuser	356.81	⇑ 9
	Време извршавања (s)	0.178	Smart Assembly	0.17	⇔
			Agile.NET	0.22	⇑ 23
			Confuser	0.3	⇑ 70
	Енергија [mJ]	55.23	Smart Assembly	55.23	⇔
			Agile.NET	89	⇑ 61
			Confuser	95.06	⇑ 82

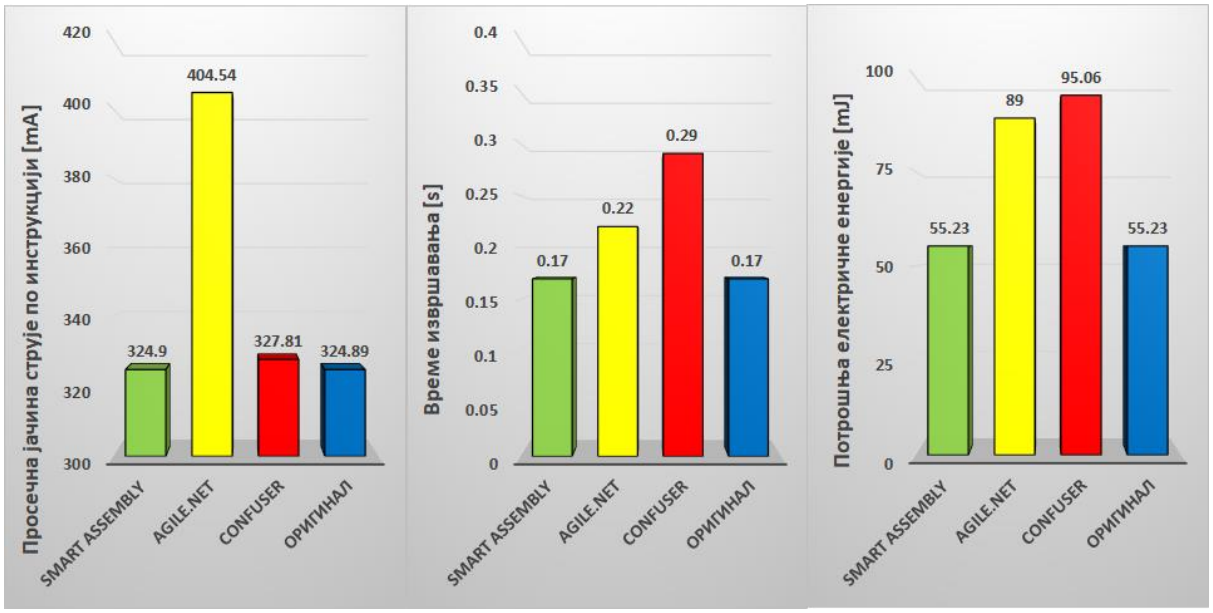
Упоредни приказ профила потрошње за тестиране алате у случају лексичке обфускације, за сценарио *quick sort* алгоритма за 10 000 елемената приказан је на сликама 5.6, 5.7 и 5.8, док је оригинални профил потрошње дат на слици 5.5. На основу добијених резултата, обједињених у табели 5.4, може се закључити следеће:

- *Smart Assembly* – као и у примеру множења матрица, показује најбоље резултате у свим тестираним сегментима. Број извршених инструкција је за 1**%** мањи у односну на оригинал. Потрошња електричне енергије, време извршавања, величина датотеке и просечна струја по инструкцији су остали на вредностима које су приближно исте оригиналним.
- *Agile.NET* – показао за нијансу лошије резултате у скоро свим категоријама у односу на *Smart Assembly*. Просечна струја по инструкцији је увећана за 24% што представља најлошији резултат од свих тестираних алата.
- *Confuser* – има најлошије перформансе у скоро свим категоријама с тим да се потрошња електричне енергије истиче са увећањем од 82%. Ово је директна последица времена извршавања које је увећано за 70% у односу на оригинал.

Илустровани приказ измерених вредности из табеле 5.4, дат је на сликама 5.11 и 5.12.



Слика 5.11 Графички приказ: број инструкција (лево), величина датотеке (десно)

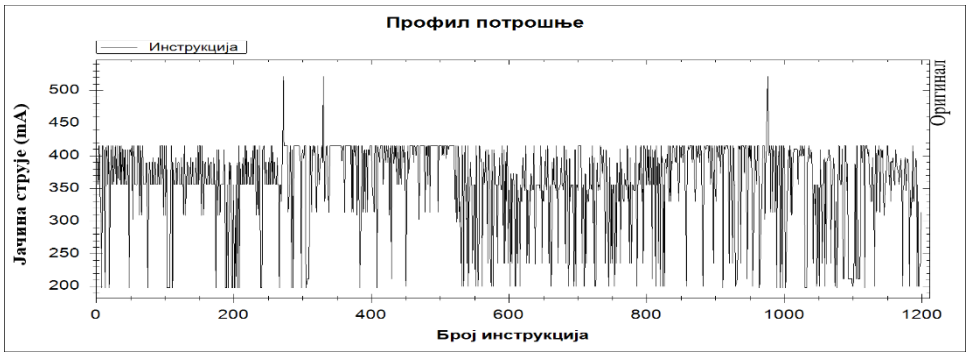


Слика 5.12 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

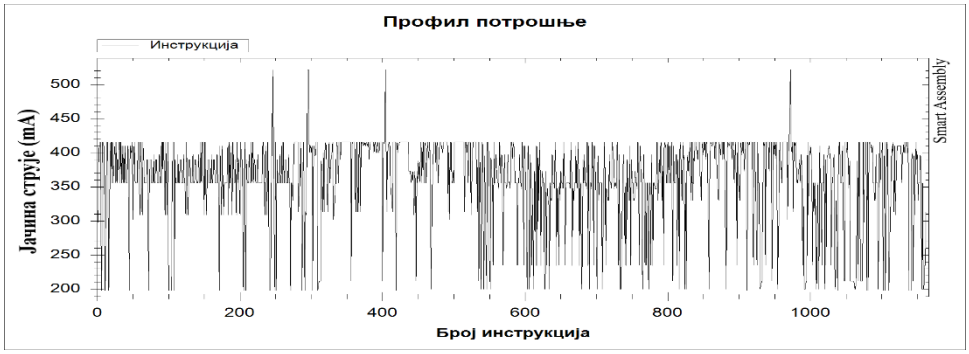
5.2 Заштита тока извршавања

Заштита тока извршавања, представља облик заштите који имплементира алгоритме за промену путање извршавања команди, посматрано са аспекта оригинала. Ово се постиже генерисањем вештачких, условних токова, петљи (лажних грана, за које не постоји репрезентација у оригиналној верзији). У овом поглављу, приказани су резултати примене овог облика заштите за споменуте алате. Детаљније, заштита тока извршавања, објашена је у глави 3.3.2, где су приказане и коришћене трансформације.

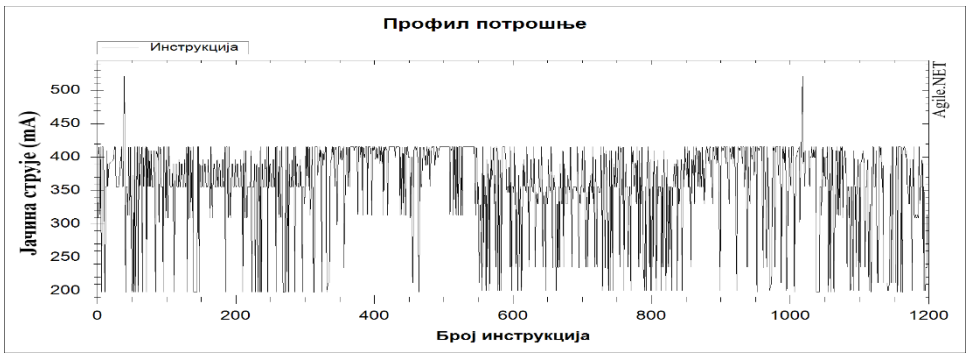
5.2.1 Множење матрица



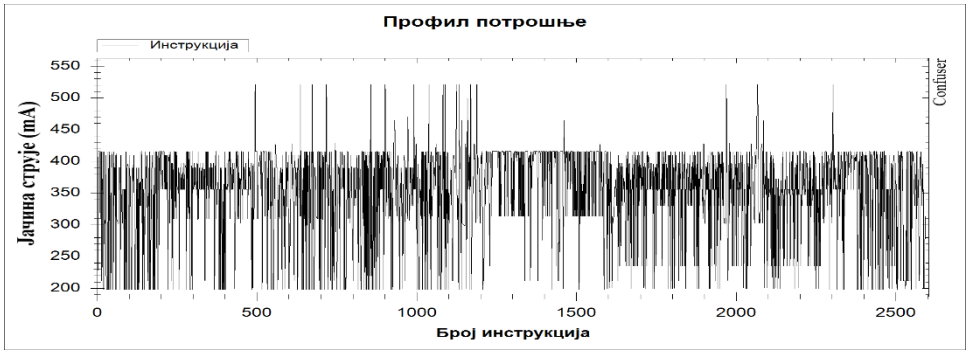
Слика 5.13 Профил потрошње немодификоване датотеке – оригинал



Слика 5.14 Ток извршавања - множење матрица - Smart Assembly



Слика 5.15 Ток извршавања - множење матрица - Agile.NET



Слика 5.16 Ток извршавања - множење матрица - Confuser

Табела 5.5 Обфускација тока извршавања - множење матрица

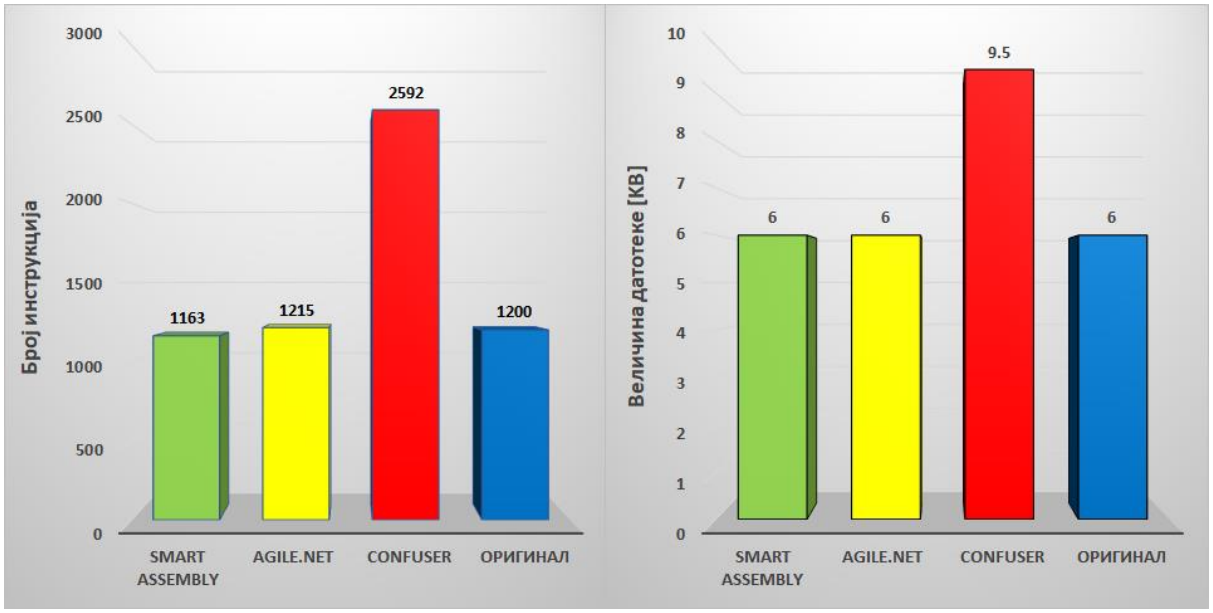
Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Множење матрица	Број асемблерских инструкција	1200	Smart Assembly	1163	⇓ 3
			Agile.NET	1215	⇑ 1
			Confuser	2592	⇑ 116
	Величина датотеке (KB)	6	Smart Assembly	6	⇔
			Agile.NET	6	⇔
			Confuser	9.5	⇑ 58
	Јачина просечне струје по инструкцији (mA)	357.34	Smart Assembly	357.36	⇔
			Agile.NET	357.33	⇔
			Confuser	346.47	⇓ 3
	Време извршавања (s)	0.0137	Smart Assembly	0.013	⇔
			Agile.NET	0.017	⇑ 30
			Confuser	0.031	⇑ 138
	Енергија [mJ]	4.65	Smart Assembly	4.65	⇔
			Agile.NET	6.07	⇑ 30
			Confuser	10.74	⇑ 131

Упоредни приказ профила потрошње за тестиране алате у случају заштите тока извршавања, за сценарио множења матрица приказан је на сликама 5.14, 5.15 и 5.16, док је оригинални профил потрошње дат на слици 5.13. На основу добијених резултата обједињених у табели 5.5, може се закључити следеће:

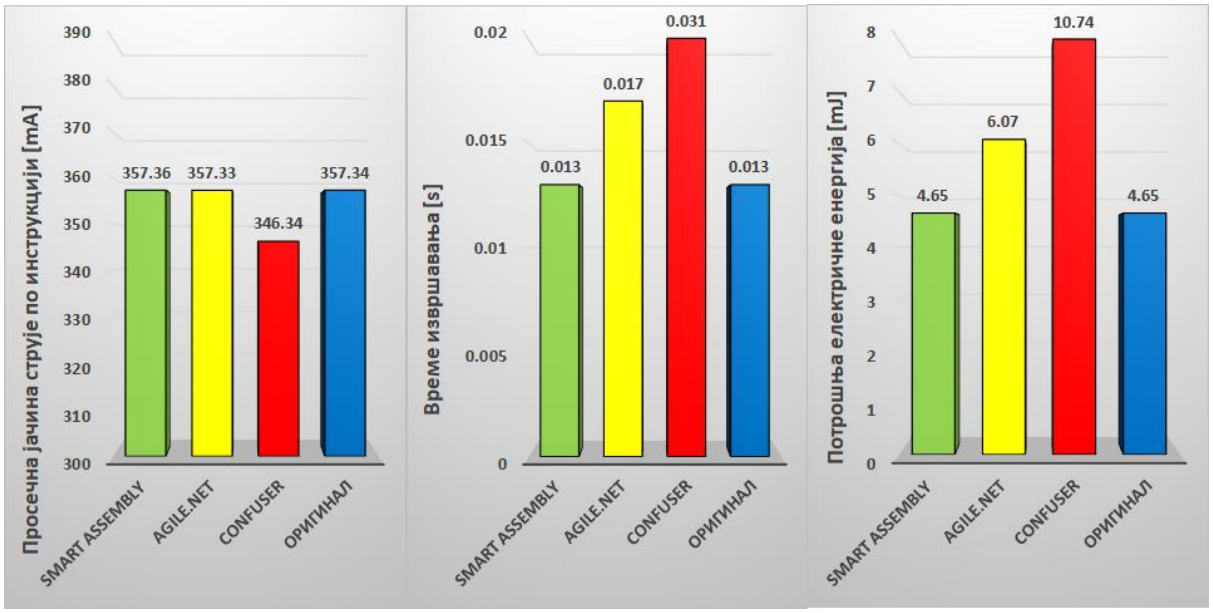
- *Smart Assembly* – показује најбоље резултате у свим тестираним сегментима, нарочито у броју инструкција које су смањене за 3% у односу на оригинал.
- *Agile.NET* – показује лошије резултате од *Smart Assembly*, где се истиче повећање потрошње електричне енергије за 30%.

- *Confuser* – за разлику од претходна два алата, овде се да приметити да су резултати знатно лошији у свим категоријама, нарочито у времену извршавања што директно утиче и на потрошњу електричне енергије (увећање од 131%). На основу добијених резултата, алат се класификује као најлошији.

Илустровани приказ измерених вредности из табеле 5.5, дат је на сликама 5.17 и 5.18.

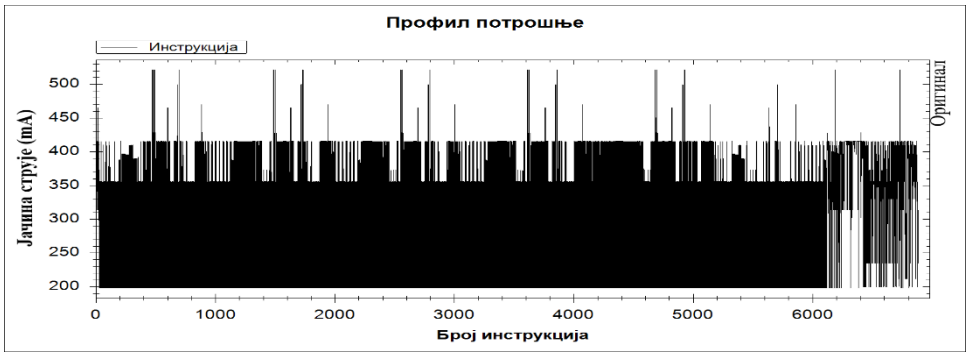


Слика 5.17 Графички приказ: број инструкција (лево), величина датотеке (десно)

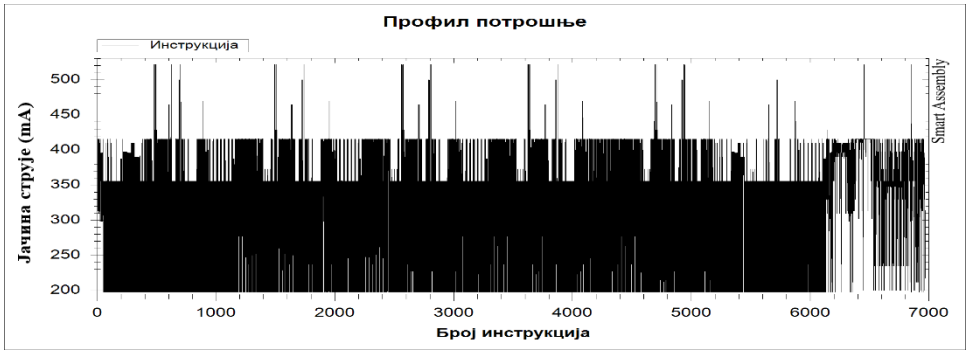


Слика 5.18 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

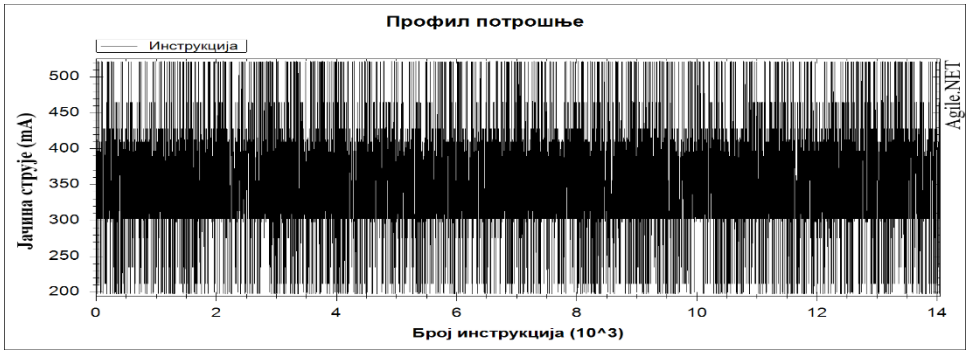
5.2.2 Quick sort алгоритам



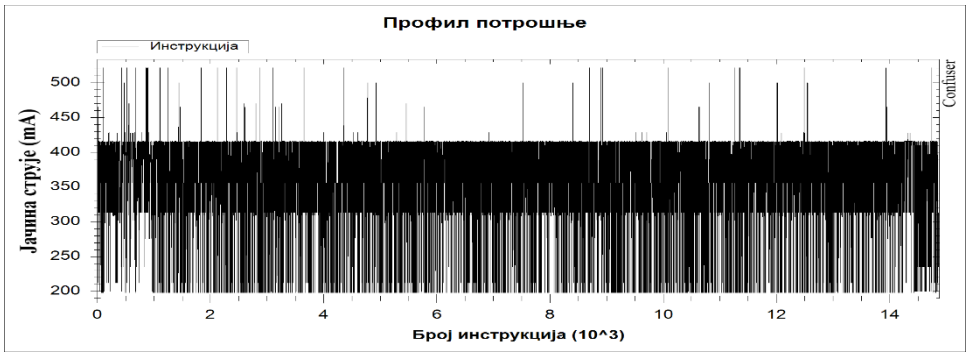
Слика 5.19 Профил потрошње немодификоване датотеке - оригинал



Слика 5.20 Ток извршавања - сортирање - Smart Assembly



Слика 5.21 Ток извршавања - сортирање - Agile.NET



Слика 5.22 Ток извршавања - сортирање - Confuser

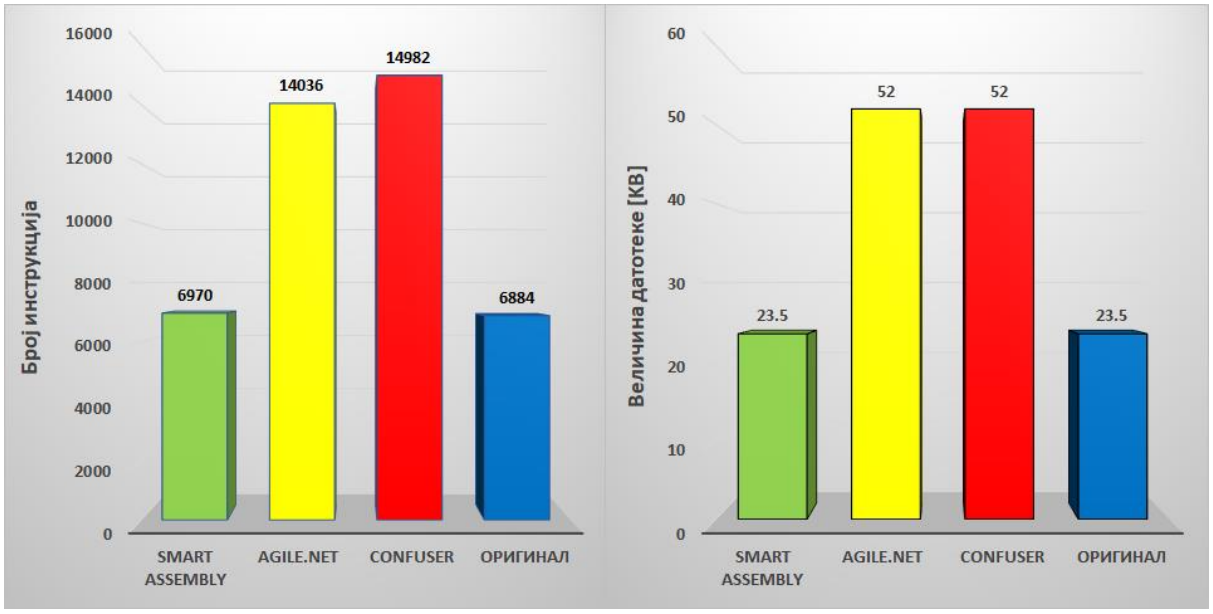
Табела 5.6 Обфускација тока извршавања - сортирање

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Quick Sort	Број асемблерских инструкција	6884	Smart Assembly	6970	⇑ 1
			Agile.NET	14036	⇑ 103
			Confuser	14982	⇑ 117
	Величина датотеке (KB)	23.5	Smart Assembly	23.5	⇔
			Agile.NET	52	⇑ 121
			Confuser	52	⇑ 121
	Јачина просечне струје по инструкцији (mA)	324.89	Smart Assembly	325.1	⇔
			Agile.NET	351.3	⇑ 8
			Confuser	328.6	⇑ 1
	Време извршавања (s)	0.178	Smart Assembly	0.18	⇔
			Agile.NET	0.33	⇑ 85
			Confuser	0.35	⇑ 96
	Енергија [mJ]	55.23	Smart Assembly	58.52	⇑ 6
			Agile.NET	115.93	⇑ 110
			Confuser	115.01	⇑ 108

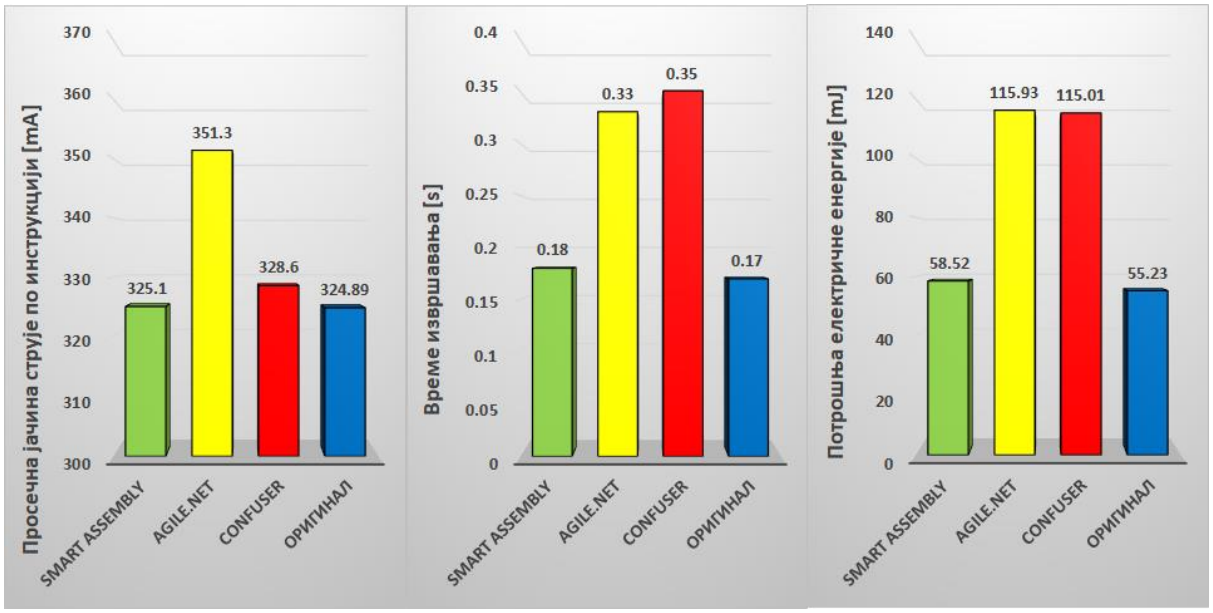
Упоредни приказ профила потрошње за тестиране алате у случају заштите тока извршавања, за сценарио *quick sort* алгоритма за 10 000 елемената приказан је на сликама 5.20, 5.21 и 5.22, док је оригинални профил потрошње дат на слици 5.19. На основу добијених резултата, обједињених у табели 5.6, може се закључити следеће:

- *Smart Assembly* – показује најбоље резултате у свим тестираним сегментима, нарочито у времену извршавања које је остало исто као код оригиналног примера.
- *Agile.NET & Confuser* – показују знатно лошије резултате од *Smart Assembly*. Сви параметри су скоро дуплирани у односу на оригинал.

Илустровани приказ измерених вредности из табеле 5.6, дат је на сликама 5.23 и 5.24.



Слика 5.23 Графички приказ: број инструкција (лево), величина датотеке (десно)

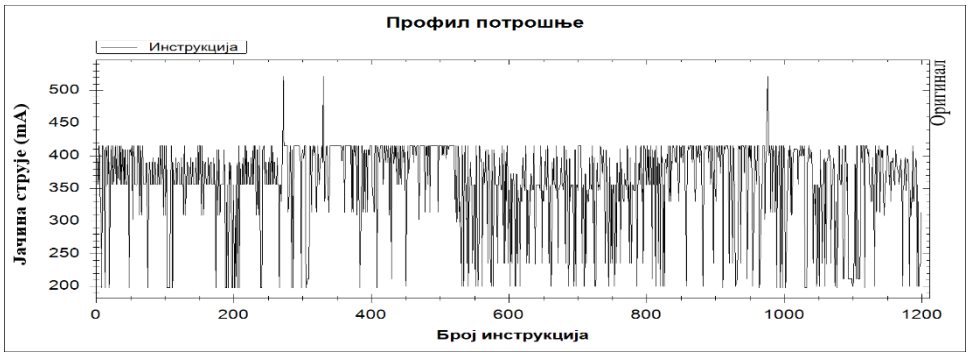


Слика 5.24 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

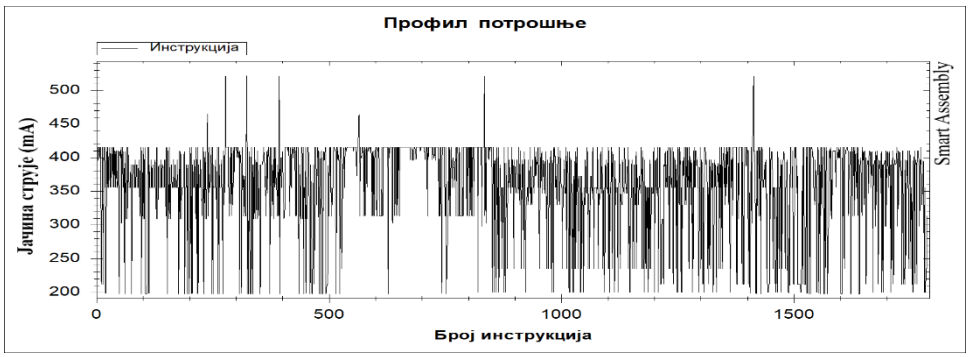
5.3 Заштита података

Заштита података представља најкомплекснији облик заштите који је детаљно обрађен у глави 3.3.3 где су приказане коришћене трансформације. У овом поглављу, приказани су резултати утицаја овог облика заштите за анализиране алате.

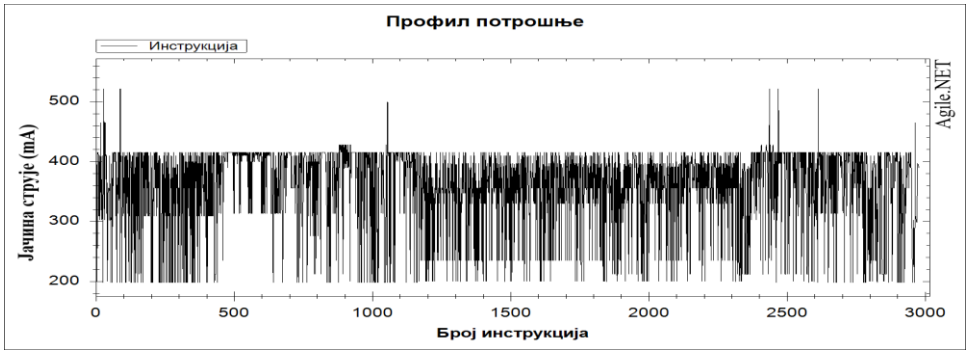
5.3.1 Множење матрица



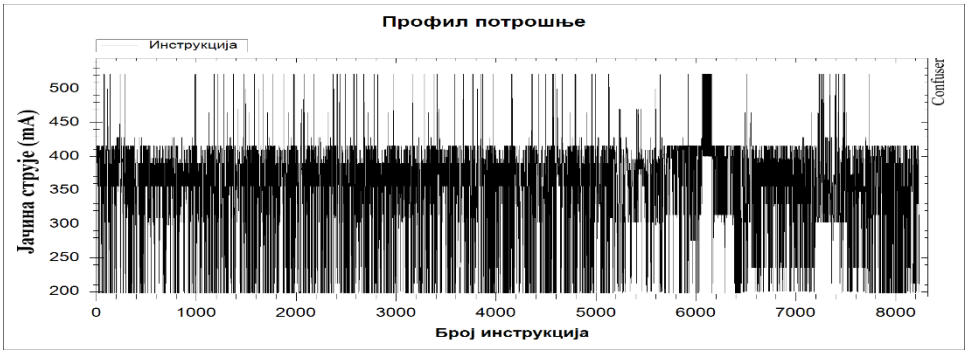
Слика 5.25 Профил потрошње немодификоване датотеке - оригинал



Слика 5.26 Обфускација података - множење матрица - Smart Assembly



Слика 5.27 Обфускација података - множење матрица - Agile.NET



Слика 5.28 Обфускација података - множење матрица - Confuser

Табела 5.7 Обфускација података - множење матрица

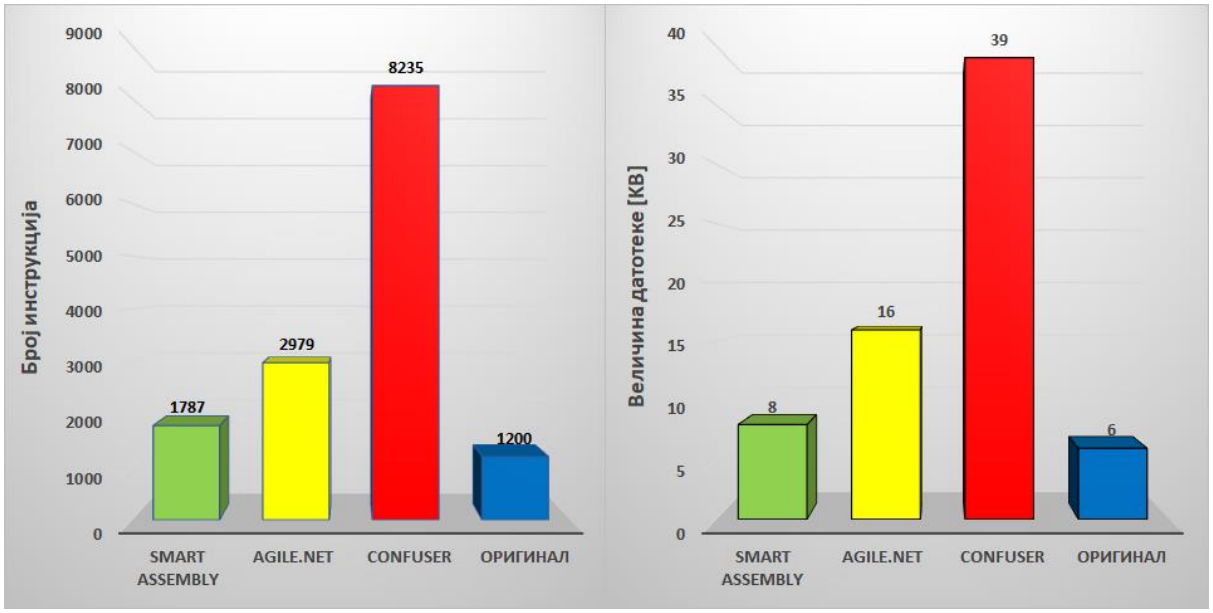
Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Множење матрица	Број асемблерских инструкција	1200	Smart Assembly	1787	⇑ 67
			Agile.NET	2979	⇑ 148
			Confuser	8235	⇑ 586
	Величина датотеке (KB)	6	Smart Assembly	8	⇑ 33
			Agile.NET	16	⇑ 166
			Confuser	39	⇑ 550
	Јачина просечне струје по инструкцији (mA)	357.34	Smart Assembly	359.3	⇔
			Agile.NET	362.4	⇑ 1.5
			Confuser	369.7	⇑ 3.5
	Време извршавања (s)	0.0137	Smart Assembly	0.015	⇑ 9
			Agile.NET	0.028	⇑ 115
			Confuser	0.073	⇑ 432
	Енергија [mJ]	4.65	Smart Assembly	5.39	⇑ 18
			Agile.NET	10.15	⇑ 122
			Confuser	26.99	⇑ 491

Упоредни приказ профила потрошње за тестиране алате у случају заштите података, за сценарио множења матрица приказан је на сликама 5.26, 5.27 и 5.28, док је оригинални профил потрошње дат на слици 5.25. Овај вид заштите има највећи утицај на енергетску ефикасност, као и на пратеће перформансе,што се може видети из резултата, обједињених у табели 5.7.

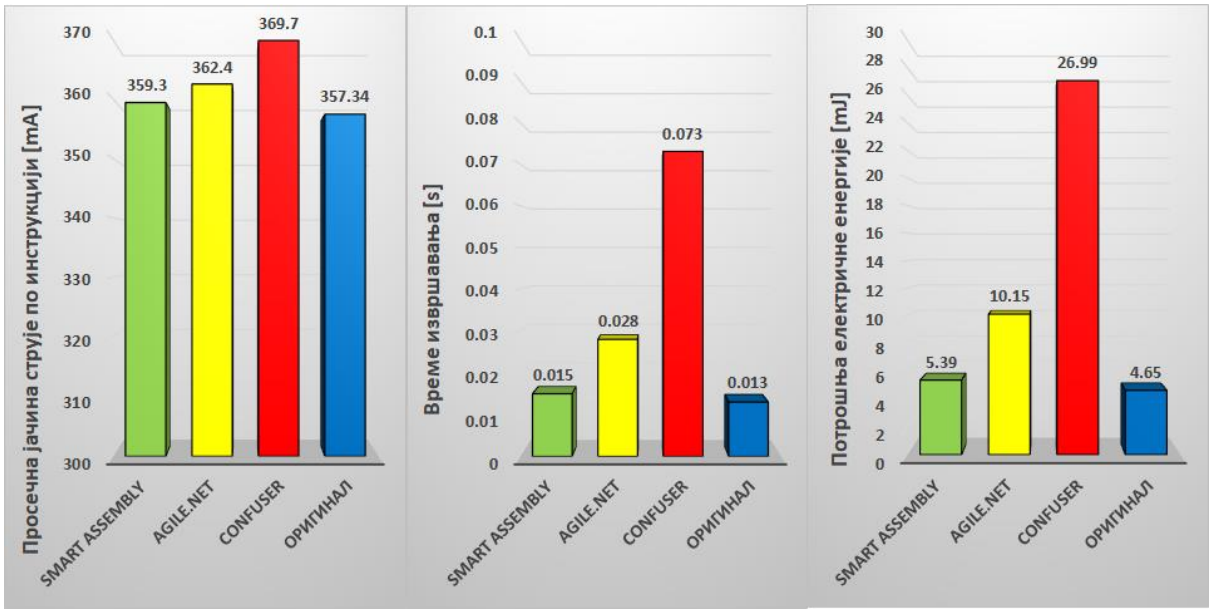
- *Smart Assembly* – показује боље резултате у односу на конкуренцију, нарочито у категорији потрошње електричне енергије, где је идентификовано увећање од само 18%.
- *Agile.NET* – показује знатно лошије резултате од *Smart Assembly*, где се истиче повећање потрошње електричне енергије за 122%.

- *Confuser* – за разлику од претходна два алата, овде се да приметити да су резултати вишеструко лошији у свим категоријама, нарочито у времену извршавања и количини потрошене електричне енергије чије повећање износи 491% у односу на оригинал. На основу добијених резултата, да се приметити да алгоритми које користи овај алат очигледно генеришу много веће инструкционе блокове а самим тим и сам алат класификују као најлошију опцију.

Илустровани приказ измерених вредности из табеле 5.7, дат је на сликама 5.29 и 5.30.

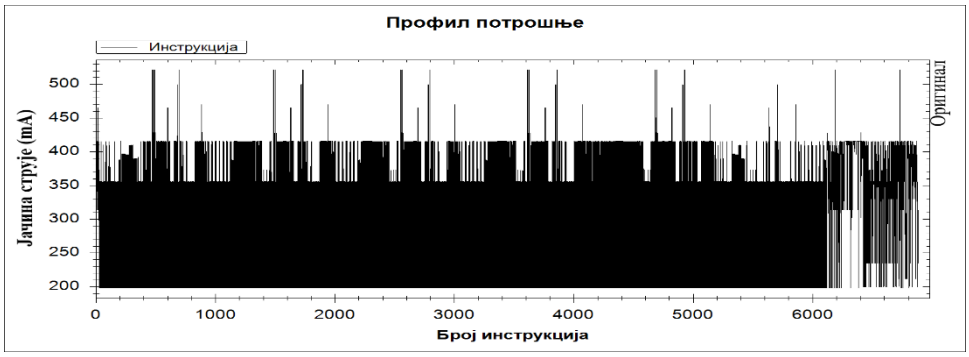


Слика 5.29 Графички приказ: број инструкција (лево), величина датотеке (десно)

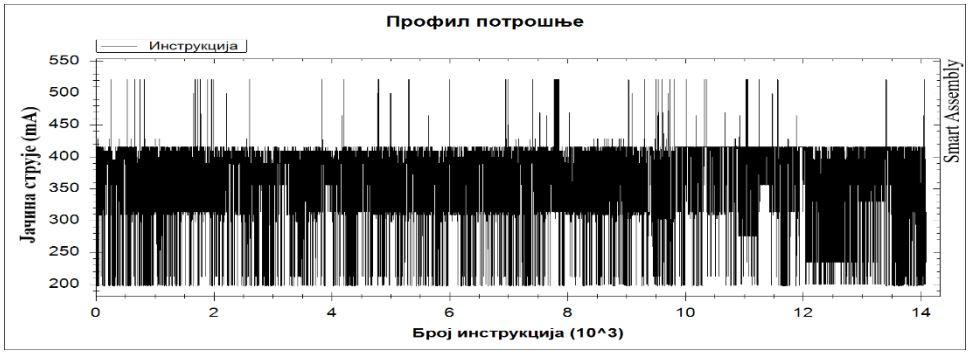


Слика 5.30 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

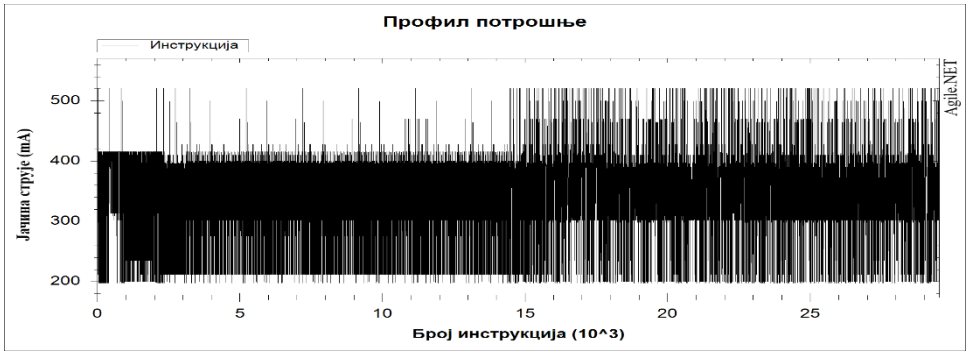
5.3.2 Quick sort алгоритам



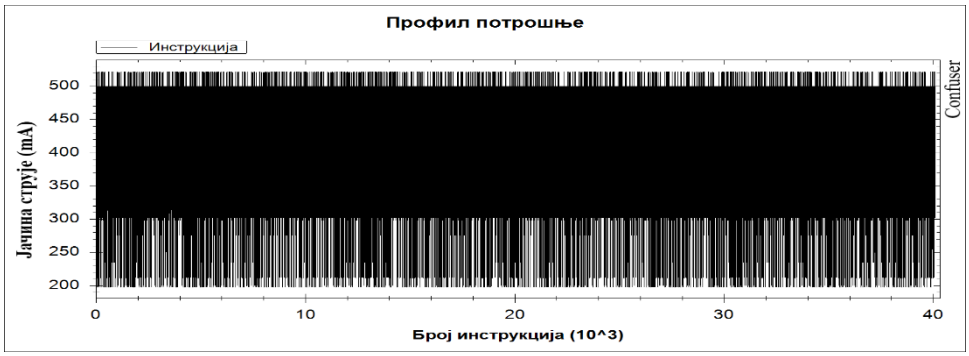
Слика 5.31 Профил потрошње немодификоване датотеке - оригинал



Слика 5.32 Обфускација података - сортирање - Smart Assembly



Слика 5.33 Обфускација података - сортирање - Agile.NET



Слика 5.34 Обфускација података - сортирање - Confuser

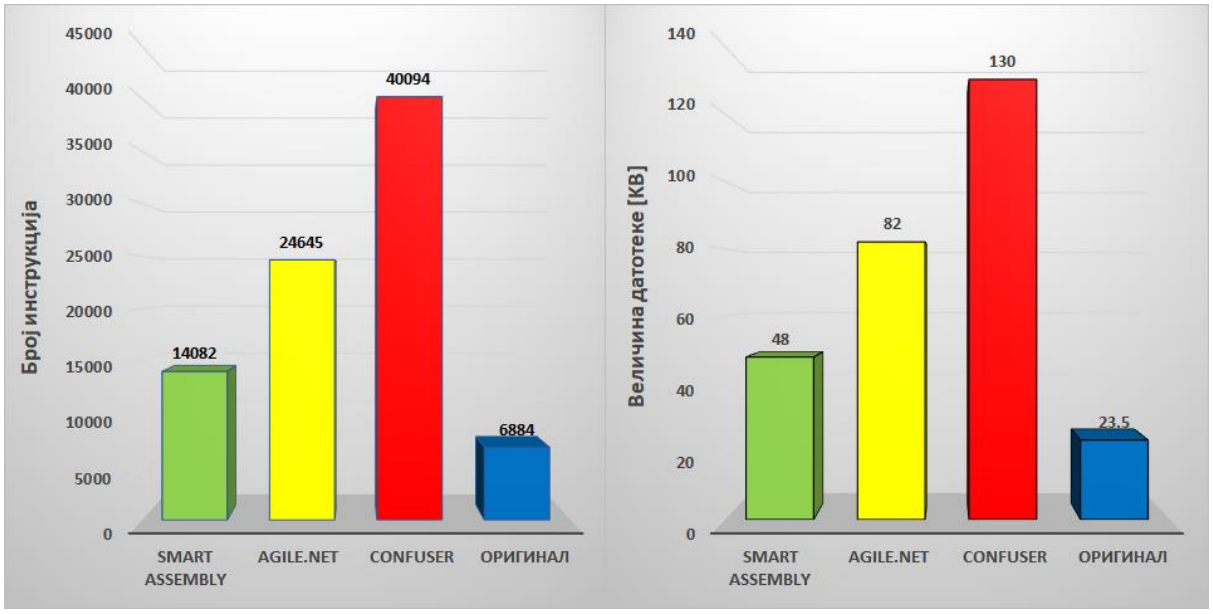
Табела 5.8 Обфускација података - сортирање

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Quick Sort	Број асемблерских инструкција	6884	Smart Assembly	14082	⇑ 104
			Agile.NET	24645	⇑ 258
			Confuser	40094	⇑ 482
	Величина датотеке (KB)	23.5	Smart Assembly	48	⇑ 104
			Agile.NET	82	⇑ 248
			Confuser	130	⇑ 453
	Јачина просечне струје по инструкцији (mA)	324.89	Smart Assembly	362.7	⇑ 11
			Agile.NET	366.5	⇑ 11
			Confuser	412.9	⇑ 27
	Време извршавања (s)	0.178	Smart Assembly	0.35	⇑ 96
			Agile.NET	0.65	⇑ 265
			Confuser	1	⇑ 461
	Енергија [mJ]	55.23	Smart Assembly	126.95	⇑ 129
			Agile.NET	238.23	⇑ 331
			Confuser	412.9	⇑ 491

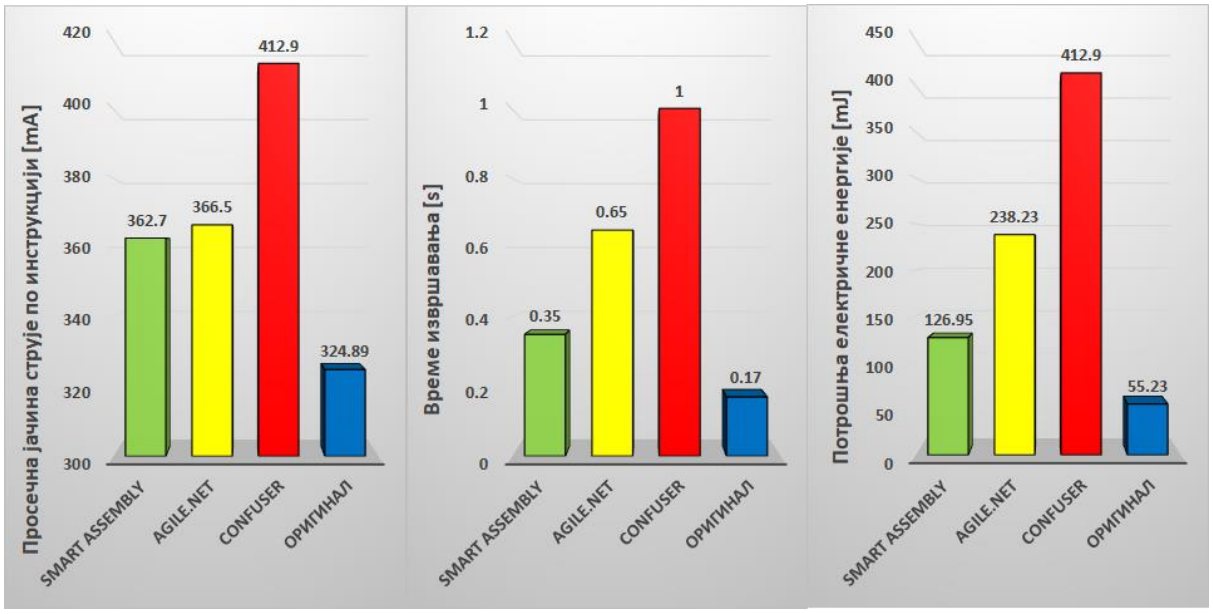
Упоредни приказ профила потрошње за тестиране алате у случају заштите података, за сценарио *quick sort* алгоритма за 10 000 елемената приказан је на сликама 5.32, 5.33 и 5.34, док је оригинални профил потрошње дат на слици 5.31. На основу добијених резултата, обједињених у табели 5.8, може се закључити следеће:

- *Smart Assembly* – дуплиран број инструкција, као и величина датотеке. Потрошња електричне енергије је увећана за 129% док је време извршавања дуплирнао.
- *Agile.NET & Confuser* – генеришу изуетно захтеван код са аспекта енергетске ефикасности. Потрошња електричне енергије је увећа вишеструко у оба случаја, као и време извршавања.

Илустровани приказ измерених вредности из табеле 5.8, дат је на сликама 5.35 и 5.36.



Слика 5.35 Графички приказ: број инструкција (лево), величина датотеке (десно)



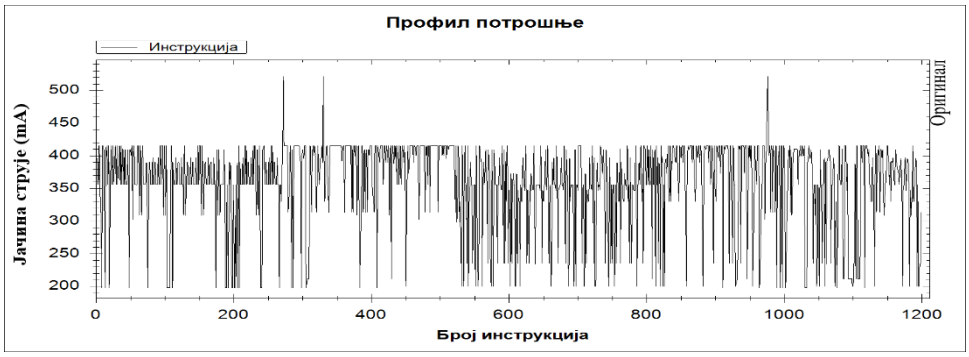
Слика 5.36 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

5.4 Комбинација лексичке и заштите тока извршавања

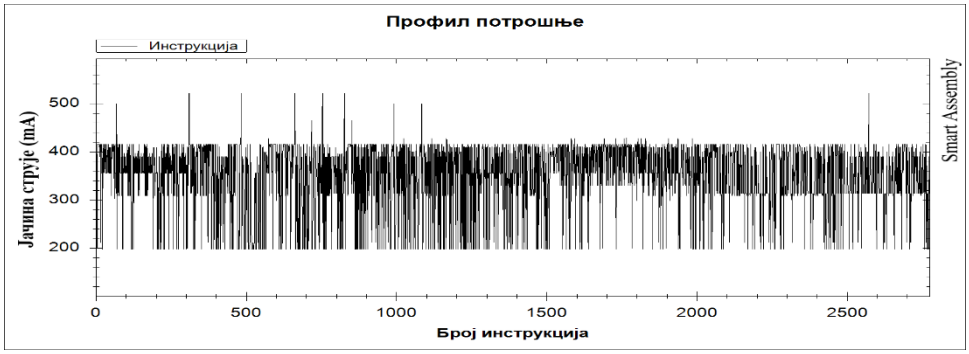
У овој глави, приказани су резултати за двоструки ниво заштите који укључује следеће типове обфускације:

- лексичку,
- обфускацију тока извршавања.

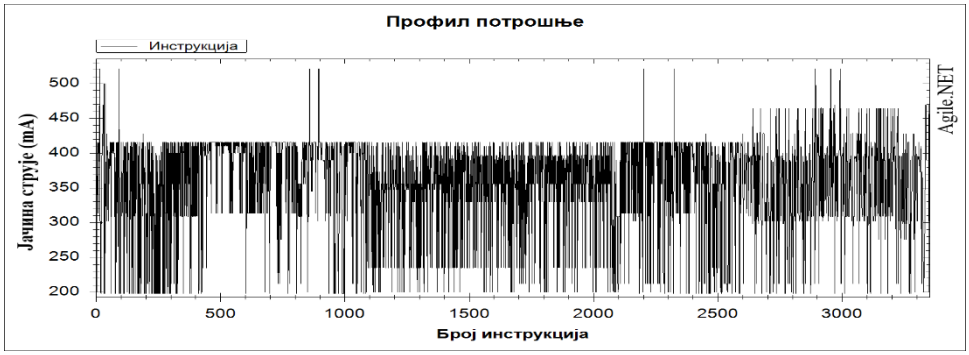
5.4.1 Множење матрица



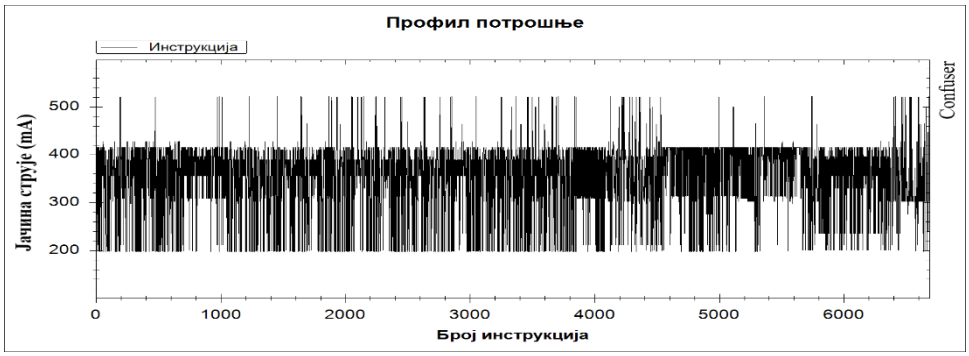
Слика 5.37 Профил потрошње немодификоване датотеке - оригинал



Слика 5.38 Два нивоа заштите - множење матрица - Smart Assembly



Слика 5.39 Два нивоа заштите - множење матрица - Agile.NET



Слика 5.40 Два нивоа заштите - множење матрица - Confuser

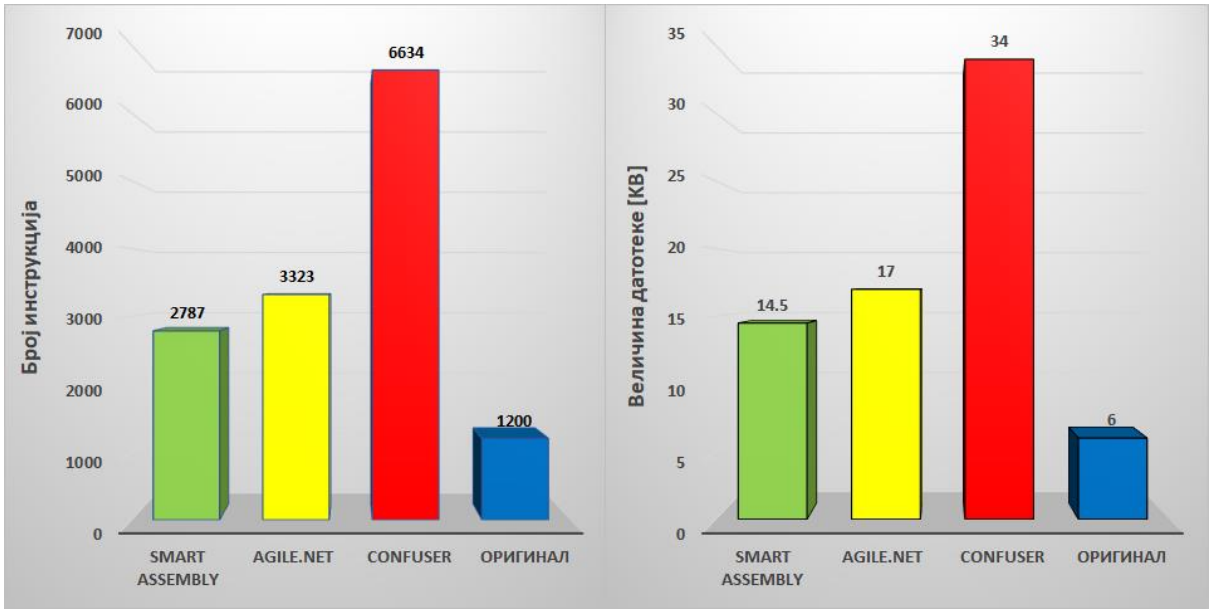
Табела 5.9 Два нивоа заштите - множење матрица

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Множење матрица	Број асемблерских инструкција	1200	Smart Assembly	2787	⇑ 132
			Agile.NET	3323	⇑ 176
			Confuser	6634	⇑ 452
	Величина датотеке (KB)	6	Smart Assembly	14.5	⇑ 141
			Agile.NET	17	⇑ 183
			Confuser	34	⇑ 466
	Јачина просечне струје по инструкцији (mA)	357.34	Smart Assembly	357.33	⇔
			Agile.NET	358.13	⇔
			Confuser	353.7	⇓ 2
	Време извршавања (s)	0.0137	Smart Assembly	0.029	⇑ 116
			Agile.NET	0.036	⇑ 162
			Confuser	0.071	⇑ 446
	Енергија [mJ]	4.65	Smart Assembly	10.36	⇑ 123
			Agile.NET	12.89	⇑ 177
			Confuser	25.11	⇑ 440

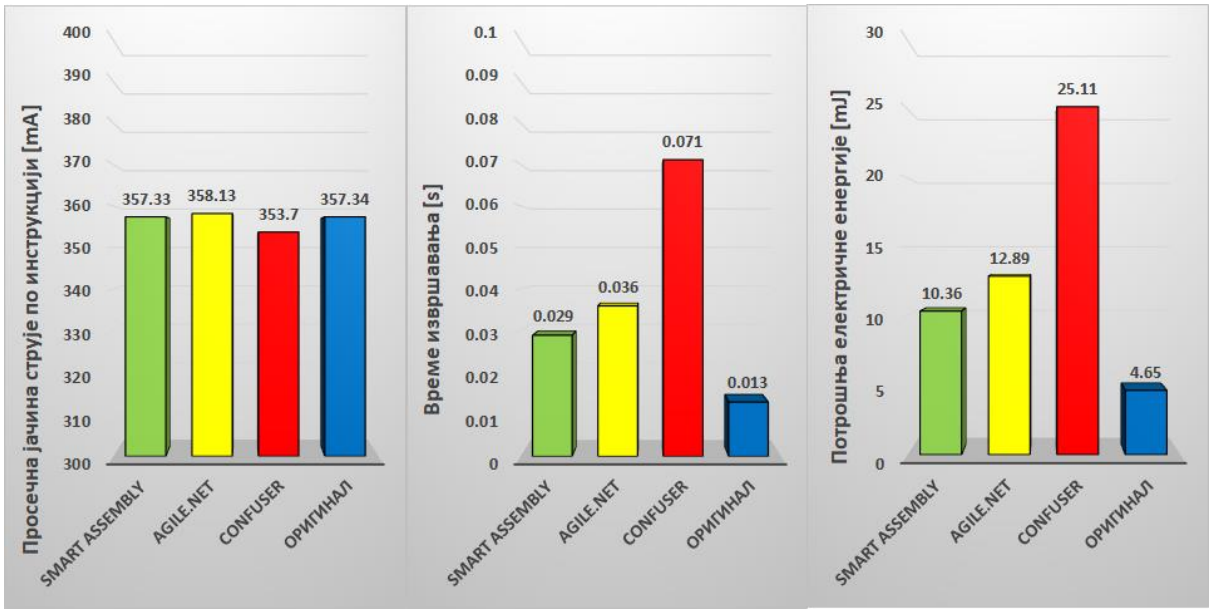
Упоредни приказ профила потрошње за тестиране алате у случају комбиноване лексичке и заштите тока извршавања, за сценарио множења матрица приказан је на сликама 5.38, 5.39 и 5.40, док је оригинални профил потрошње дат на слици 5.37. На основу добијених резултата, обједињених у табели 5.7, може се закључити следеће:

- *Smart Assembly* – показује знатно лошије резултате у односу на појединаче примене наведених заштита. Број инструкција је увећан за 132%, потрошња електричне енергије за 123%. Просечна струја по инструкцији је остала идентична оригиналној, док је време извршавања увећано за 116%.
- *Agile.NET* – показује знатно лошије резултате од *Smart Assembly*. Инструкциони сет је увећан за 176%, потрошена електрична енергија за 177%, док је јачина просечне струја остала приближно иста као и оригинал.
- *Confuser* – за разлику од претходна два алата, овде се да приметити да су резултати вишеструко лошији у свим категоријама, нарочито у времену извршавања и количини потрошене електричне енергије чије повећање износи 440% у односу на оригинал.

Илустровани приказ измерених вредности из табеле 5.9, дат је на сликама 5.41 и 5.42.

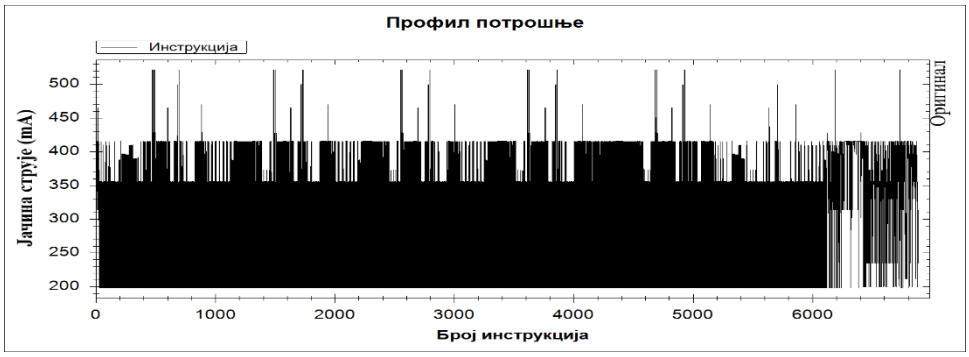


Слика 5.41 Графички приказ: број инструкција (лево), величина датотеке (десно)

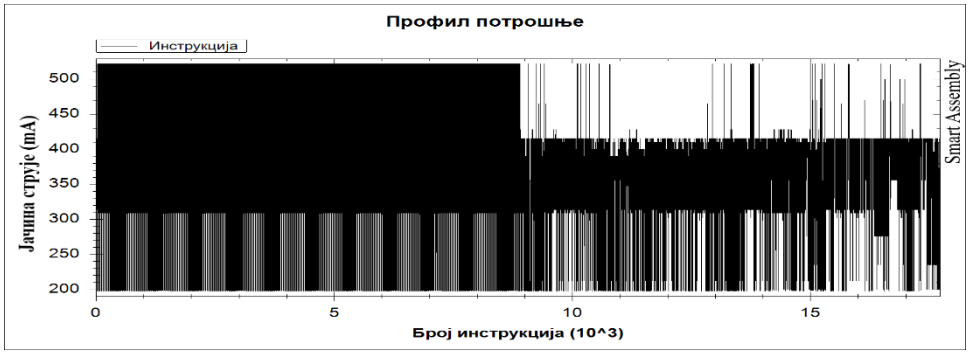


Слика 5.42 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

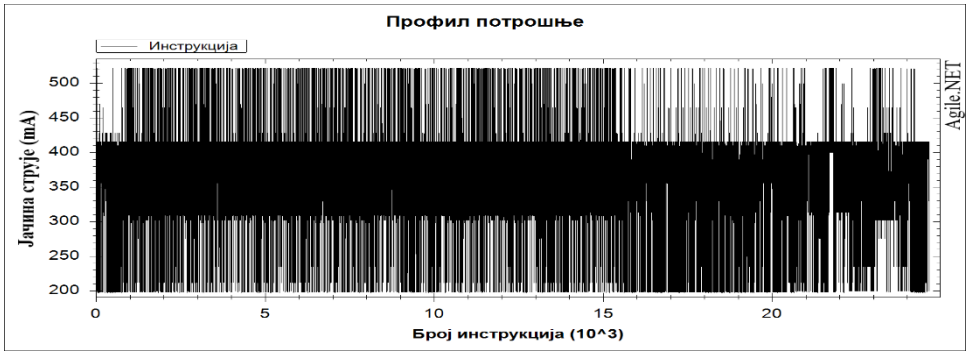
5.4.2 Quick sort алгоритам



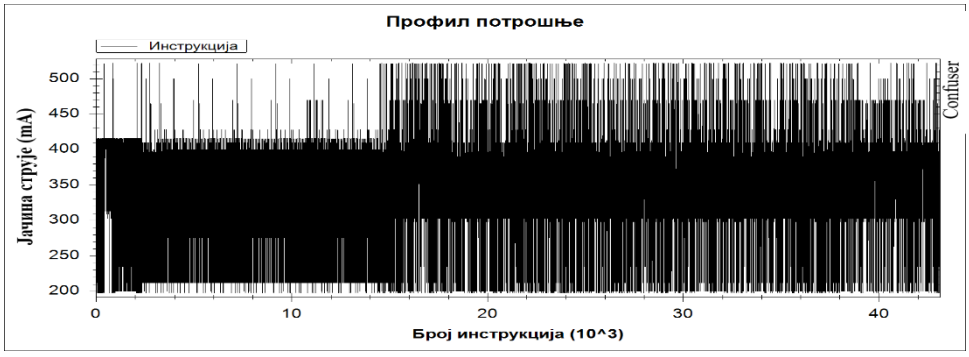
Слика 5.43 Профил потрошње немодификоване датотеке - оригинал



Слика 5.44 Два нивоа заштите - сортирање - Smart Assembly



Слика 5.45 Два нивоа заштите - сортирање - Agile.NET



Слика 5.46 Два нивоа заштите - сортирање - Confuser

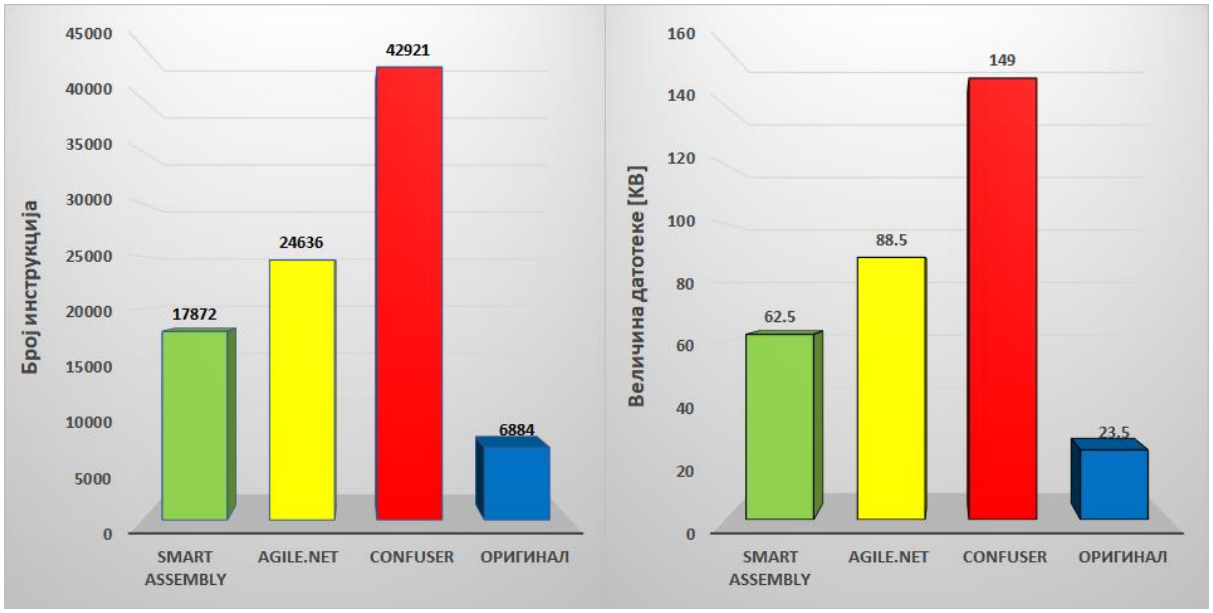
Табела 5.10 Два нивоа заштите - сортирање

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Quick Sort	Број асемблерских инструкција	6884	Smart Assembly	17872	⇑ 159
			Agile.NET	24636	⇑ 257
			Confuser	42921	⇑ 523
	Величина датотеке (KB)	23.5	Smart Assembly	62.5	⇑ 166
			Agile.NET	88.5	⇑ 276
			Confuser	149	⇑ 534
	Јачина просечне струје по инструкцији (mA)	324.89	Smart Assembly	355.2	⇑ 9
			Agile.NET	392.6	⇑ 21
			Confuser	363.1	⇑ 11
	Време извршавања (s)	0.178	Smart Assembly	0.48	⇑ 169
			Agile.NET	0.66	⇑ 270
			Confuser	1.2	⇑ 574
	Енергија [mJ]	55.23	Smart Assembly	170.5	⇑ 208
			Agile.NET	259.12	⇑ 369
			Confuser	435.72	⇑ 688

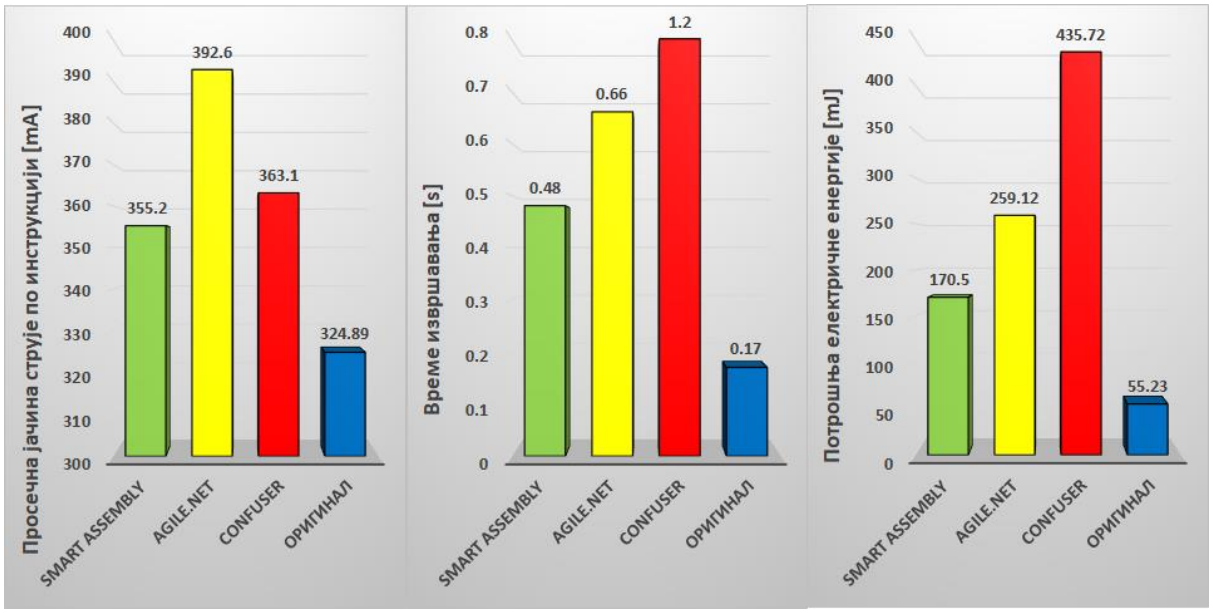
Упоредни приказ профила потрошње за тестиране алате у случају комбиноване лексичке и заштите тока извршавања, за *quick sort* сценарио приказан је на сликама 5.44, 5.45 и 5.46, док је оригинални профил потрошње дат на слици 5.43. На основу добијених резултата, обједињених у табели 5.10, може се закључити следеће:

- *Smart Assembly* – потрошња електричне енергије је увећана за преко 200%, просечна струја по инструкцији је увећана за 9%, док је брзина извршавања увећана за 145%.
- *Agile.NET* – показује знатно лошије резултате од *Smart Assembly*. Списак инструкција је увећан за 257%, потрошња електричне енергије за 369%, док је јачина просечне струја порасла за 21%.
- *Confuser* – за разлику од претходна два алата, *Confuser* приказује далеко најлошије перформансе у свим категоријама. Потрошња електричне енергије је повећана за скоро 700% у односу на оригинал.

Илустровани приказ измерених вредности из табеле 5.10, дат је на сликама 5.47 и 5.48.



Слика 5.47 Графички приказ: број инструкција (лево), величина датотеке (десно)



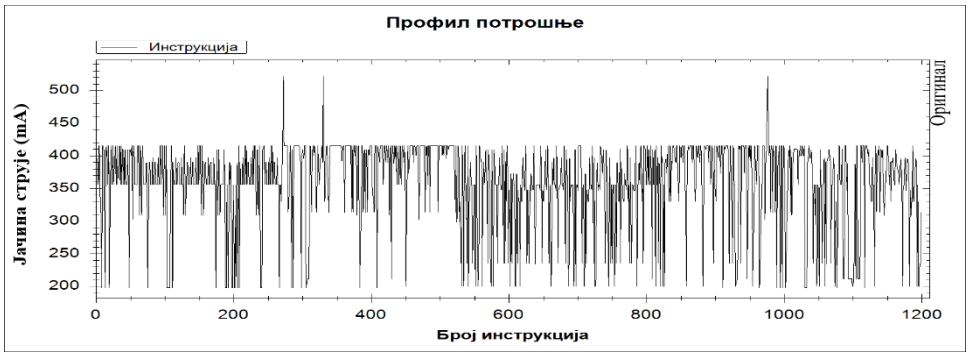
Слика 5.48 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

5.5 Комбинација лексичке, тока извршавања и заштите података

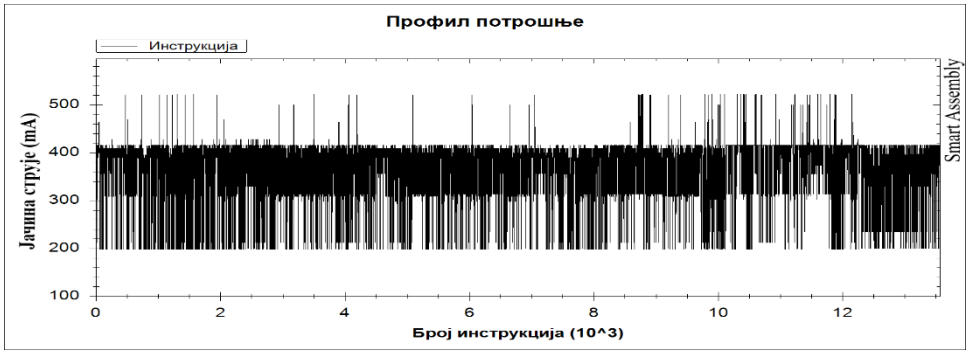
У овој глави, приказани су резултати за троструки ниво заштите који укључује следеће типове обфускације:

- лексичку,
- обфускацију тока извршавања,
- обфускацију података.

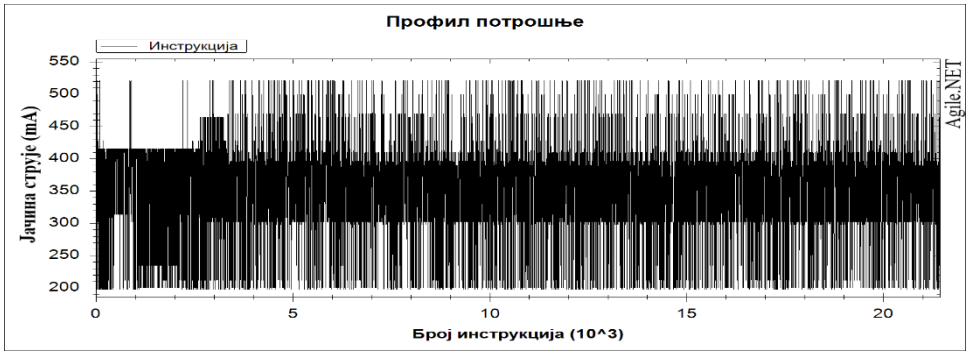
5.5.1 Множење матрица



Слика 5.49 Профил потрошње немодификоване датотеке - оригинал



Слика 5.50 Три нивоа заштите - множење матрица - Smart Assembly



Слика 5.51 Три нивоа заштите - множење матрица - Agile.NET

Слика 5.52 Три нивоа заштите - множење матрица - Confuser

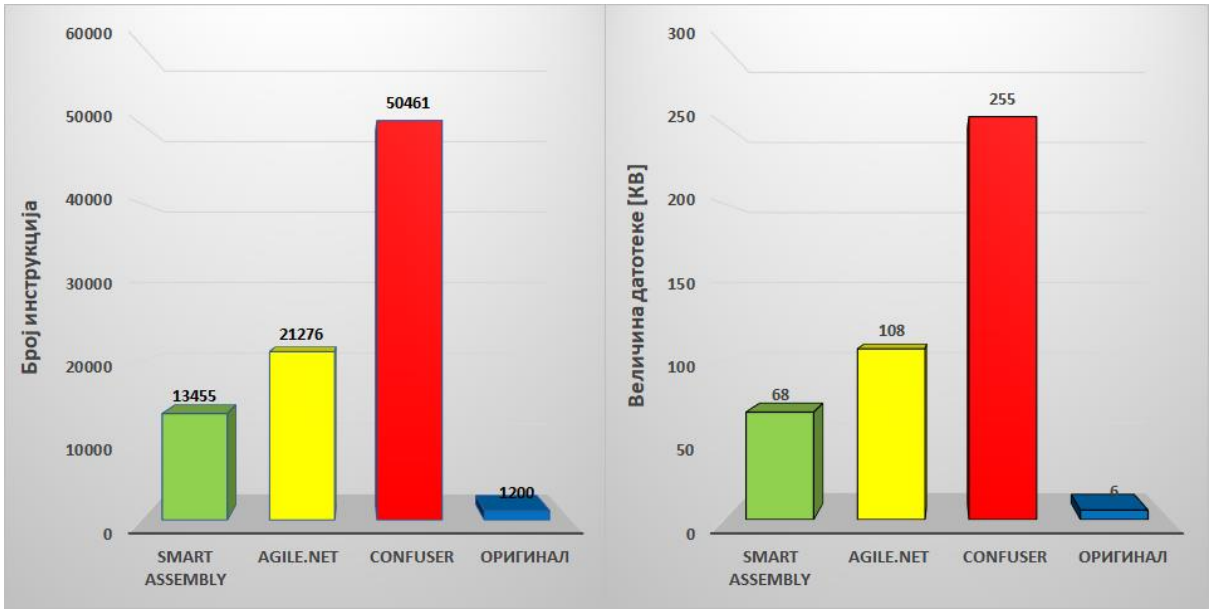
Табела 5.11 Три нивоа заштите - множење матрица

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Множење матрица	Број асемблерских инструкција	1200	Smart Assembly	13455	⇑ 1021
			Agile.NET	21276	⇑ 1673
			Confuser	50461	⇑ 4105
	Величина датотеке (KB)	6	Smart Assembly	68	⇑ 1033
			Agile.NET	108	⇑ 1700
			Confuser	255	⇑ 4151
	Јачина просечне струје по инструкцији (mA)	357.34	Smart Assembly	359.4	⇔
			Agile.NET	361.1	⇑ 1
			Confuser	356.6	⇔
	Време извршавања (s)	0.0137	Smart Assembly	0.16	⇑ 1067
			Agile.NET	0.24	⇑ 1651
			Confuser	0.57	⇑ 4062
	Енергија [mJ]	4.65	Smart Assembly	57.5	⇑ 1136
			Agile.NET	86.66	⇑ 1763
			Confuser	203.26	⇑ 4271

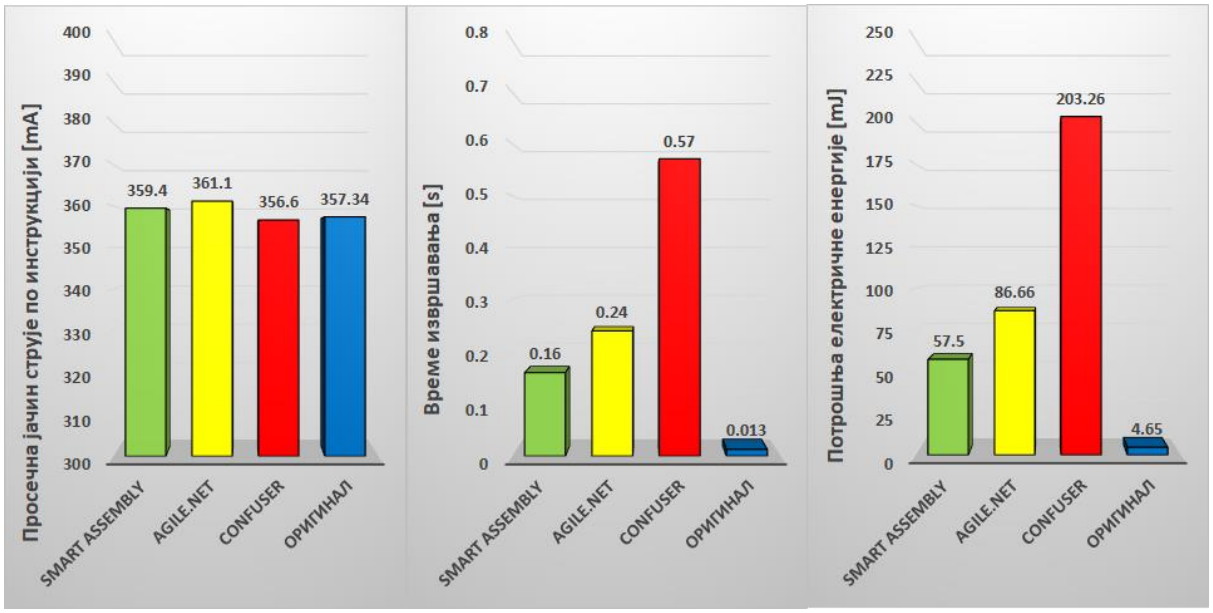
Упоредни приказ профила потрошње за тестиране алате у случају комбиноване лексичке, заштите тока извршавања и заштите података, за сценарио множења матрица приказан је на сликама 5.44, 5.45 и 5.46, док је оригинални профил потрошње дат на слици 5.43. На основу добијених резултата, обједињених у табели 5.10, може се закључити следеће:

- *Smart Assembly* – потрошња електричне енергије је увећана за преко 1136%, јачина просечне струје по инструкцији је занемарљиво увећана, док је брзина извршавања скочила за 1067%.
- *Agile.NET* – показује знатно лошије резултате од *Smart Assembly*. Списак инструкција је увећан за 1673%, потрошња електричне енергије за 1763%, док је величина датотеке порасла за 1700%.
- *Confuser* – за разлику од претходна два алата, *Confuser* приказује далеко најлошије перформансе у свим категоријама. Потрошња електричне енергије је повећана за абнормалних 4271% у односу на оригинал, време извршавања за 4062% док је величина датотеке порасла за 4151%.

Илустровани приказ измерених вредности из табеле 5.10, дат је на сликама 5.47 и 5.48.

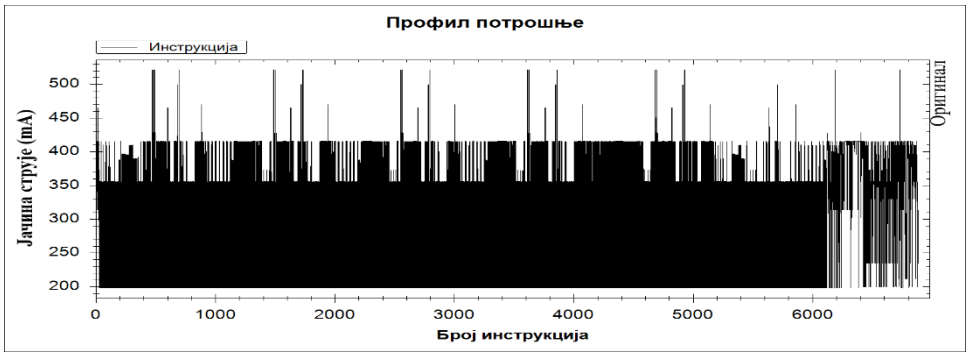


Слика 5.53 Графички приказ: број инструкција (лево), величина датотеке (десно)

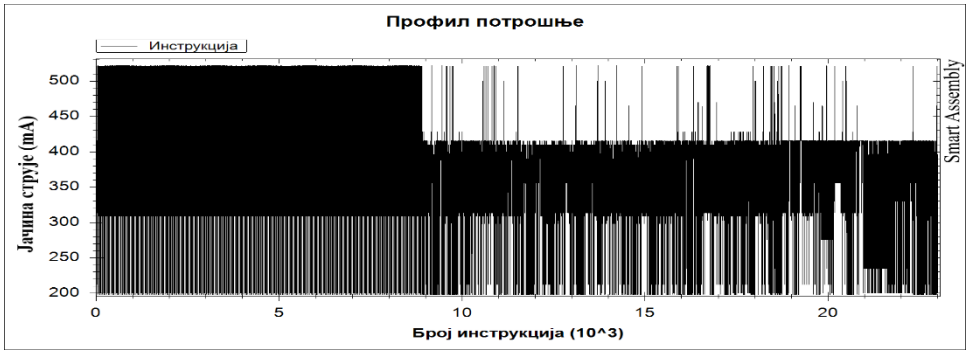


Слика 5.54 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

5.5.2 Quick sort алгоритам

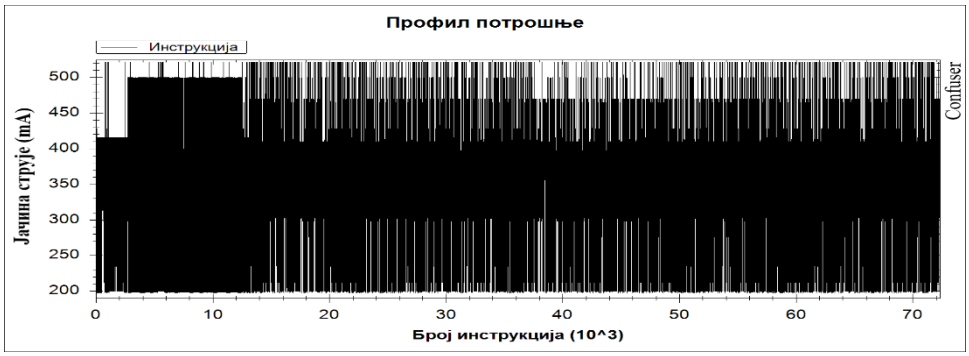


Слика 5.55 Профил потрошње немодификоване датотеке - оригинал



Слика 5.56 Три нивоа заштите - сортирање - Smart Assembly

Слика 5.57 Три нивоа заштите - сортирање - Agile.NET



Слика 5.58 Три нивоа заштите - сортирање - Confuser

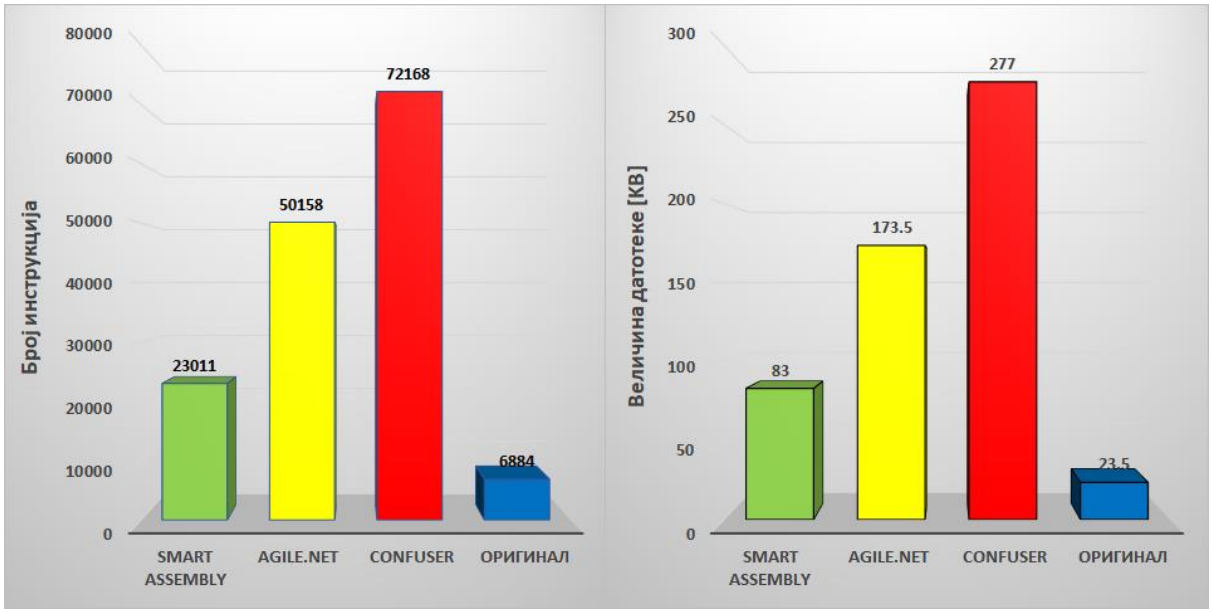
Табела 5.12 Три нивоа заштите - сортирање

Датотека	Мерени параметри	Пре обфускације	После обфускације		Промена [%]
Quick Sort	Број асемблерских инструкција	6884	Smart Assembly	23011	⇑ 234
			Agile.NET	50158	⇑ 628
			Confuser	72168	⇑ 948
	Величина датотеке (KB)	23.5	Smart Assembly	83	⇑ 253
			Agile.NET	173.5	⇑ 638
			Confuser	277	⇑ 1078
	Јачина просечне струје по инструкцији (mA)	324.89	Smart Assembly	354.1	⇑ 9
			Agile.NET	402.5	⇑ 24
			Confuser	378.4	⇑ 16
	Време извршавања (s)	0.178	Smart Assembly	0.56	⇑ 214
			Agile.NET	1.4	⇑ 686
			Confuser	2	⇑ 1023
	Енергија [mJ]	55.23	Smart Assembly	198.3	⇑ 259
			Agile.NET	563.5	⇑ 920
			Confuser	756.8	⇑ 1270

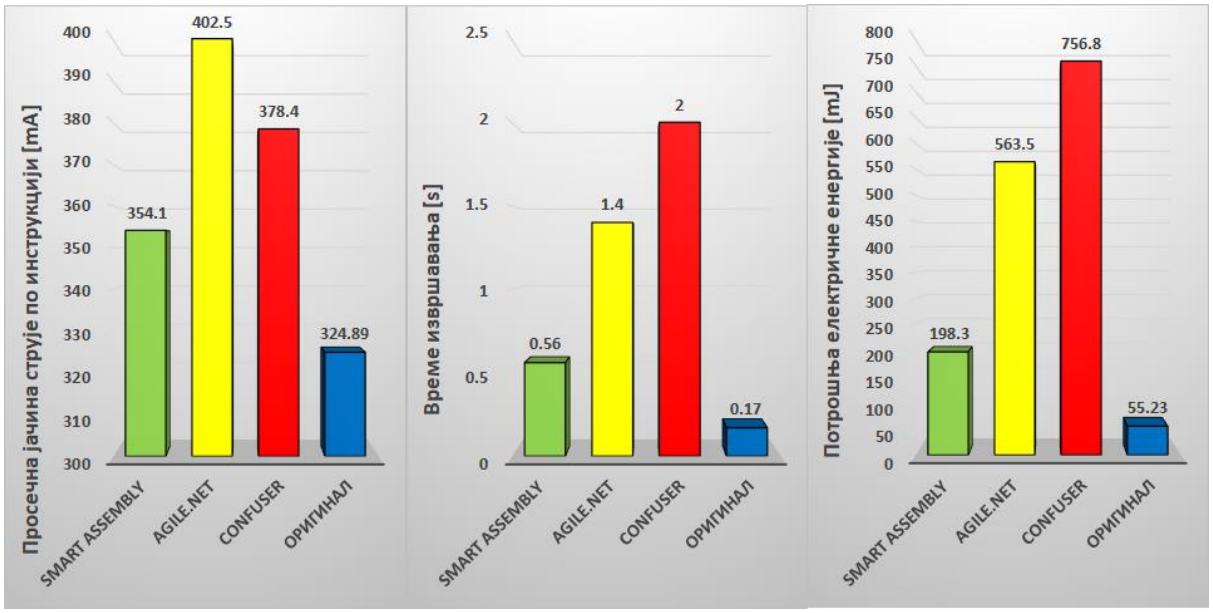
Упоредни приказ профила потрошње за тестиране алате у случају комбиноване лексичке, заштите тока извршавања и заштите података, за сценарио множења матрица приказан је на сликама 5.56, 5.57 и 5.58, док је оригинални профил потрошње дат на слици 5.55. На основу добијених резултата, обједињених у табели 5.11, може се закључити следеће:

- *Smart Assembly* – потрошња електричне енергије је увећана за преко 230%, јачина просечне струје по инструкцији увећана је за 9%, време извршавања увећано је за 214%, док је потрошња електричне енергије повећана за 259%.
- *Agile.NET* – показује изузетно лоше резултате у поређењу како са оригиналним тако и са резултатима *Smart Assembly*-а. Списак извршених инструкција увећан је за 628%, величина датотеке за 638% док је потрошња електричне енергије је порасла за чак 920%.
- *Confuser* – слично као и *Agile.NET, Confuser* опет генерише најлошије резултате у свим категоријама. Потрошња електричне енергије повећана је за 1270% у односу на оригинал. Ова промена је повезана са великим временом извршавања као и просечном струјом по инструкцији.

Илустровани приказ измерених вредности из табеле 5.10, дат је на сликама 5.59 и 5.60.



Слика 5.59 Графички приказ: број инструкција (лево), величина датотеке (десно)



Слика 5.60 Графички приказ: јачина струја, време извршавања, потрошња електричне енергије

Дискусија резултата

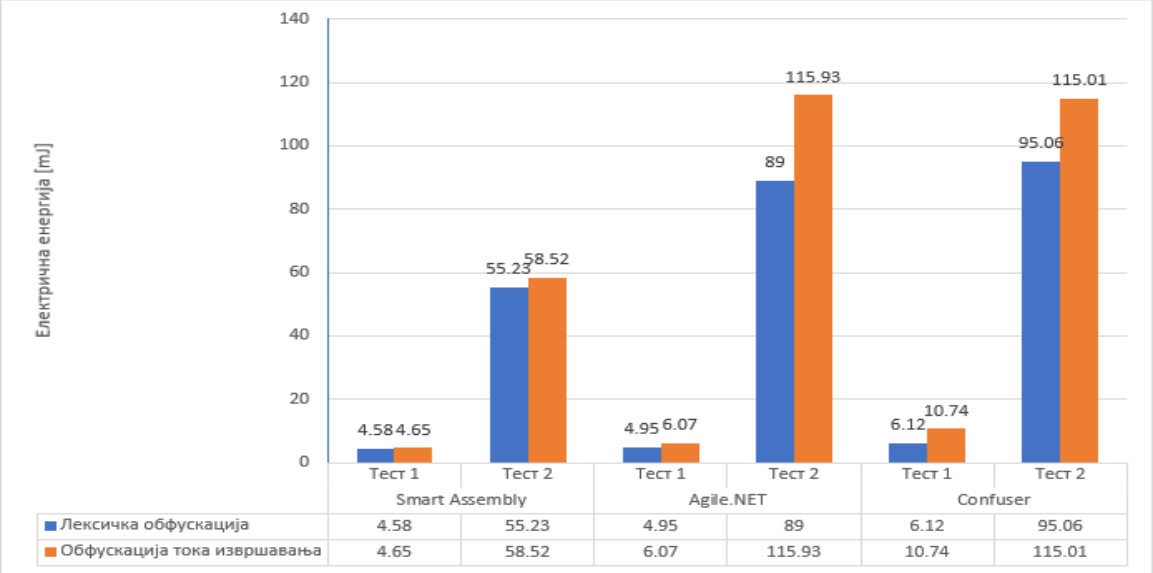
У циљу свеобухватнијег поређења резулата анализираних обфускационих трансформација и потврде хипотеза, табеларно ће бити представљни упоредни резултати појединачних нивоа заштите разврстани по тестираним сценацијима и коришћеним алатима.

Тестирани сценаији:

- Множење матрица – Тест 1,

- *Quicksort* – Тест 2.

На слици 5.61 приказане су упоредне вредности потрошње електричне енергије анализираних алата (*Smart Assembly, Agile.NET, Confuser*) за лексичку и обфускацију тока извршавања.



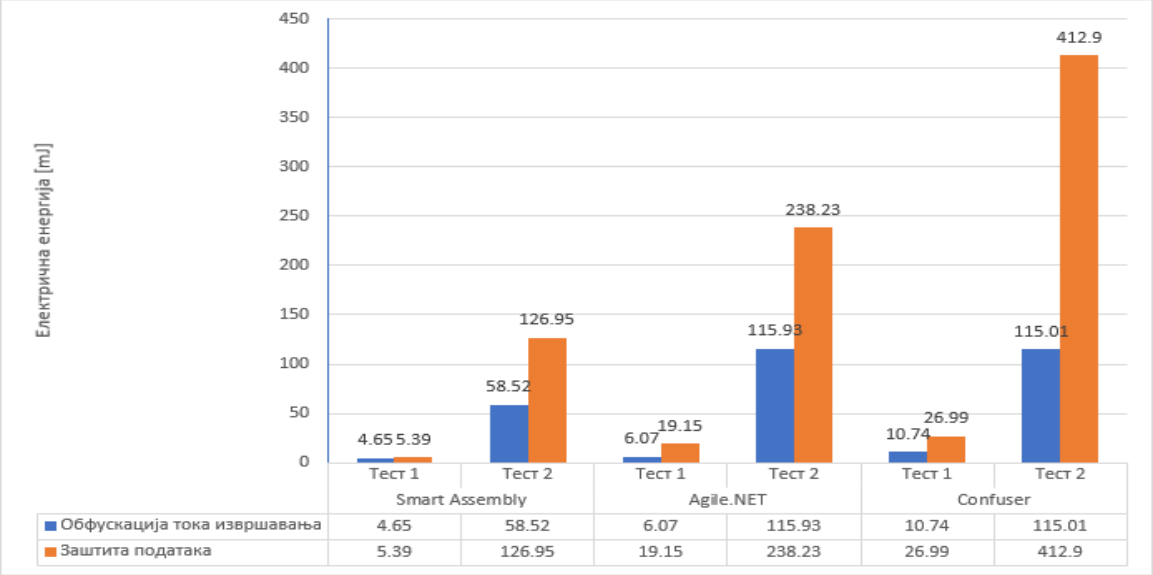
	Smart Assembly		Agile.NET		Confuser	
	Тест 1	Тест 2	Тест 1	Тест 2	Тест 1	Тест 2
Лексичка обфускација	4.58	55.23	4.95	89	6.12	95.06
Обфускација тока извршавања	4.65	58.52	6.07	115.93	10.74	115.01

Слика 5.61 Потрошња електричне енергије анализираних алата за лексичку и заштиту тока извршавања

Резултати тестирања, за оба сценарија, показују да заштита кода трансформацијама тока извршавања производи енергетски захтевнији профил него након примене лексичке обфускације. Односно, тестирани сценарији над којима је примењена обфускација тока извршавања троше више електричне енергије.

Добијени резултати потврђују хипотезу Х2 која каже: ”***Показати да обфускација тока извршавања генерише енергетски захтевнији профил потрошње од лексичке обфускације, односно код који троши више електричне енергије приликом свог извршавања.***”

На слици 5.62, приказане су упоредне вредности потрошње електричне енергије анализираних алата (*Smart Assembly, Agile.NET, Confuser*) заштиту тока извршавања и заштиту података.



Слика 5.62 Потрошња електричне енергије анализираних алата за заштиту тока извршавања и заштиту података

Резултати тестирања, за оба сценарија, показују да заштита кода трансформацијама података производи енергетски захтевнији профил него након примене обфускације тока извршавања. Односно, тестирани сценарији над којима је примењена обфускација података троше више електричне енергије.

Добијени резултат потврђују хипотезу Х3 која каже: ”***Показати да обфускација података генерише енергетски захтевнији профил потрошње од профила насталог применом обфускације тока извршавања.***”

Добијени резултати, приказани профилима потрошње током главе 5 недвосмилено показују утицај обфускационих техника на профиле потрошње. Додатно, у главама 5.4 и 5.5 приказани су резултати комбинованих примена анализираних обфускационих типова.

Приказани резултати потврђују хипотезу Х1 која каже: “***Показати да обфускација кода мења профил потрошње у зависности од типа обфускације као и тестираног узорка.***”

6 Закључак

Захваљујући профилима потрошње, у овој докторској дисертацији приказан је утицај различитих обфускационих техника пре свега на енергетску ефикасност која је обрађена коришћењем профила потрошње, као и потрошња електричне енергије. Додатно, показан је утицај на величину извршне датотеке, број асемблерских инструкција и време извршавања. Јединственост коју нам споменути приступ омогућава, уједно отвара врата дефинисању потенцијалних метрика за процену енергетске ефикасности великих софтверских решења. Поред сигурности и перформанси, аспект енергетске ефикасности данас добија велику популарност, посебно увођењем нових квалитативних атрибута као што је одрживост *IT* система. Разматрање енергије као важног фактора мора бити доведено у први план, нарочито у будућности где ова тематика добија све више простора у истраживачким сферама. Са друге стране, заштита софтверских решења представља посебан изазов, где обфускација представља најкоришћенији, али не и једни механизам заштите.

Прегледом актуелног стања у разматраној области, закључено је да је не постоје ефикасне методе за детекцију утицаја алата за промену разумљивости кода на енергетску ефикасност.

У овој докторској дисертацији, анализиран је утицај обфускационих трансформација на енергетску ефикасност програма и предложена је техника која се базира на профилима потрошње. Тестирање је реализовано за три различите програмске подршке (*Confuser, Agile.NET, Smart Assembly*) применом следећих трансформационих нивоа заштите:

- лексичка,
- заштита тока извршавања,
- заштита података.

Да би се добили експериментални резултати, у *.NET* окружењу, имплементирана је прилагођена архитектура мерења заснована на статичкој анализи кода, као и сам генератор профила потрошње. Предложена техника, верификована је на два тестна сценарија (алгоритам за множење матрица димензије 100×100, алгоритам за брзо сортирање 10000 елемената (*quick sort*)). Анализом резултата, показано је да одређене обфускационе трансформације генеришу веће, и изузетно захтевније профиле са аспекта енергетске ефикасности, као што је обфускација заштите података. Овај рад показује да се профили потрошње могу користити као метрика за мерење ефикасности и класификацију самих обфускатора.

Профили потрошње, као механизам идентификације злонамерних софтвера су врло искористива, обећавајућа [27, 116] и тренутно слабо истражена област. Злоћудни програми користе обфускацију као технику да би сакрили своју присутност и да би се са лакоћом дистрибуирали по систему. Техником динамичког дисасемблинга би се могло

вршити препознавање и класификација коришћених обфускатора при њиховој детекцији с обзиром да сваки обфускатор има свој препознатљив отисак (енг. *thumbprint*).

Поред детекције неуобичајених промена на софтверу, профиле потрошње је могуће користити и за откривање кварова на хардверу што је посебно интересантно у високо-дистрибуираним системима.

Даља истраживања

Даља истраживања у области утицаја обфускације на енергетску ефикасност могу се одвијати у праву проширења типова анализираних апликација (нпр. *web* апликације). Такође, интересантно би било показати утицај *McCabe*-ове цикломатске комплексности на енергетску ефикасност.

Могућност даљег истраживања и усавршавања предложеног приступа огледа се пре свега у унапређењу модела асемблерских инструкција, односно конструкцији хардверске инструментације за мерење пикова јачине струје асемблерских инструкција за процесоре нових генерација.

7 Литература

[1] T. Guelzim, M. Obaidat, “Chapter 8 – Green Computing and Communication Architecture", *Handbook of Green Information and Communication Systems*, Academic Press, January 2013.

[2] H.T. Mouftah, B. Kantarci, “Energy Efficient Cloud Computing – A Green Migration of Traditional IT”, *Handbook of Green Information and Communication Systems*, Academic Press, pp. 295-330, December 2013.

[3] National Academy of Technologies of France, “Impact des TIC sur la consommation d’énergie à travers le monde”, EDP Sciences, https://www.academie-technologies.fr/en/blog/posts/energy-and-carbon-prints-effects-of-icts-on-energy-consumption-round-the-world (29.5.2019.), 2015.

[4] Anne-Cécile Orgerie, Marcos Dias de Assunção, and Laurent Lefèvre, “A Survey on Techniques for Improving the Energy Efficiency of Large Scale Distributed Systems”, *ACM Computing Surveys*, vol.46, no.4, 2014.

[5] Fabrice Demarthon, Denis Delbecq, and Grégory Fléchet, “The Big Data Revolution”, CNRS International Magazine, http://www.cnrs.fr/fr/pdf/cim/CIM28.pdf (29.5.2019.), December 2012.

[6] Jonathan Koomey, “Growth in Data Center Electricity Use 2005 to 2010”, *Analytics Press*, August 2011.

[7] James Glanz, “Power, Pollution and the Internet”, *The New York Times*, September 2012.

[8] Marko Đuković, Ervin Varga, “Load profile-based efficiency metrics for code obfuscators”, *Acta Polytechnica Hungarica*, vol. 15, no. 5, pp. 191-212, 2015, *DOI: 10.12700/APH.12.5.2015.5.11*

[9] Cagri Sahin, Lori Pollock, James Clause, “How do code refactorings affect energy usage?”, *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement*, IEEE/ACM, September 2014.

[10] Peréz-Castillo Ricardo, Piattini Mario, “Analyzing the harmful effect of god class refactoring on power consumption”, *IEEE Software*, vol. 31, pp. 48-54, January 2014.

[11] Johann Timo, Dick Markus, Naumann Stefan, Kern Eva, “How to measure energy-efficiency of software: Metrics and measurement results”, *Proceeding of*

the First International Workshop on Green and Sustainable Software, IEEE, pp. 51-54, June 2012.

[12] Nicholas Hunt, Paramjit Singh Sandhu, Luis Ceze, "Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures", *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures*, IEEE, pp. 63-70, June 2011.

[13] Cagri Sahin, Furkan Cayci, Irene Lizeth Gutierrez, James Clause, Fouad Kiamilev, Lori Pollock, Kristina Winbladh, "Initial Explorations on Design Pattern Energy Usage", *Proceeding of the First International Workshop on Green and Sustainable Software*, pp. 55-61, June 2012.

[14] Christian Bunse and Sebastian Stiemer, "On the energy consumption of design patterns", *Proceedings of the 2nd Workshop EASED@BUIS Energy Aware Software-Engineering and Development*, pp. 7–8, March 2013.

[15] Irene Manotas, Cagri Sahin, James Clause, Lori Pollock, Kristina Winbladh, "Investigating the impact of Web Servers on Web Application Energy Usage", *Proceeding of the Second International Workshop on Green and Sustainable Software*, September 2013.

[16] Pinto G., Castor F., Liu Yd., "Understanding energy behaviors of thread management constructs", *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ACM, pp. 345-360, 2014.

[17] Liu Yd., "Energy-efficient synchronization through program patterns", *Proceedings of the First International Workshop on Green and Sustainable Software*, IEEE, pp. 35-40, 2012.

[18] Noureddine Adel, Bourdon Aurelion, Rouvoy Romain, Seinturier Lionel, "A preliminary study of the impact of software engineering on GreenIT", *Proceedings of the First International Workshop on Green and Sustainable Software (GREENS)*, IEEE, pp. 21–27, June 2012.

[19] Noureddine Adel, Rouvoy Romain, Seinturier Lionel, "Monitoring energy hotspots in software", *Journal of Automated Software Engineering*, SPRINGER, 22(3): 291–332, 2015.

[20] Noureddine Adel, Bourdon Aurelion, Rouvoy Romain, Seinturier Lionel, "Runtime monitoring of software energy hotspots", *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pp. 160-169, September 2012.

[21] Jain Ravi, Molnar David, Ramzan Zulfikar, “Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments”, *Proceedings of the 2005 Joint Workshop on Foundations of Mobile Computing*, ACM, pp. 70-79, September 2005.

[22] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software”, *Addison-Wesley*, October 1994.

[23] Sherif Yacoub, Hany Ammar, “Pattern-Oriented Analysis and Desing: Composing Pattern to Design Software Systems”, Addison Wesley, August 2003.

[24] Feitosa Daniel, Alders Rutger, Ampatzoglou Apostolos, Avgeriou Paris, Nakagawa Elisa Yumi, “Investigating the effect of design patterns on energy consumption”, *Journal of Software: Evolution and Process*, vol. 29, no. 2, January 2017.

[25] Khomh Foutse, Gueheneuc Yann Gael, Antoniol Giuliano, “Playing roles in design patterns: An empirical descriptive and analytic study”, *Proceedings of the IEEE International Conference on Software Maintenance*, IEEE, pp. 83–92, September 2009.

[26] Ampatzoglou Apostolos, Chatzigeorgiou Alexander, Charalampidou Sofia, Avgeriou Paris, “The effect of GoF design patterns on stability: A case study”, *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 781–802, March 2015.

[27] Mark Anderson, “Rooting Out Malware With a Side Channel Chip Defense System”, *IEEE Spectrum*, January 2015.

[28] V. Tiwari, S. Malik, A. Wolfe, “Instruction Level Power Analysis and Optimization of Software”, *IEEE Transactions Very Large Scale Integration (VLSI Systems)*, pp. 326-328, 1996.

[29] S. Nikolaidis, Th. Laopoluos, “Instruction Level Power Consumption Estimation of Embedded Processors for Low Power Application”, *Proceedings of the International Workshop on Intelligent Data Acquisition and Advanced Computing* Systems, IEEE, vol. 24, pp. 133-137, August 2002.

[30] J.T.Russel, M.F. Jacome, “Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors”, *Proceedings International Conference on Computer Desing*, pp. 328-333, October 1998.

[31] V. Tiwari, S. Malik, A. Wolfe, “Power Analysis of Embedded Software: A First Step Towards Software Power Minimization”, *IEEE Transactions Very Large Scale Integration (VLSI)*, pp. 437-445, 1994.

[32] Cagri Sahin, Philip Tornquist, Ryan McKenna, Zachary Pearson, James Clause, “How Does Code Obfuscation Impact Energy Usage?”, *IEEE International Conference on Software Maintenance and Evolution*, IEEE, pp. 131-140, October 2014.

[33] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti and Maurizio Morisio, “Definition, Implementation and Validation of Energy Code Smells: an Exploratory Study on an Embedded System”, *in ENERGY 2013 : The Third International Conference on Smart Grids, Green Communications and IT Energy aware Technologies*, pp. 34–39, March 2013.

[34] Marion Gottschalk, Jan Jelschen, and Andreas Winter, “Saving Energy on Mobile Devices by Refactoring”, *EnviroInfo Conference*, pp. 437– 444, September 2014.

[35] RWE Power AG, “Kernkraftwerk Emsland”, 2013.

[36] Mirco Josefiok, Marcel Schröder and Anreas Winter, “An Energy Abstraction Layer for Mobile Computing Devices”, *Softwaretechnik - Trends*, vol. 5, 2013.

[37] Marcel Schröder, “Erfassung des Energieverbrauchs von Android Apps”, Carl von Ossietzky University, Diploma Thesis, 2013.

[38] A. Rodriguez, M. Longo, and A. Zunino, “Using bad smell-driven code refactorings in mobile applications to reduce battery usage”, *Simposio de Argentino*, 2015.

[39] Abram Hindle, “Green mining: a methodology of relating software change and configuration to power consumption”, *Empirical Software Engineering*, SPRINGER, vol. 20, no. 2, pp. 374-409, April 2015.

[40] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos, and L. B. de Brisolara, “Analysis and evaluation of the android best practices impact on the efficiency of mobile applications”, *Brazilian Symposium on Computing Systems Engineering (SBESC)*, IEEE Computer Society, pp. 157–158, November 2013.

[41] Mario Linares-Vasquez, Gabriele Bavota, Carlos Bernal-Cardenas, Rocco Oliveto, Massimiliano Di Penta and Denys Poshyvanyk, “Mining energy-greedy api usage patterns in android apps: An empirical study”, *Proceedings of the 11th Working Conference on Mining Software Repositories*, ACM, pp. 2–11, 2014.

[42] Christian Bunse, Hagen Hopfner, Suman Roychoudhury, and Essam Mansour, “Choosing the ’best’ sorting algorithm for optimal energy consumption”, *Proceedings of the 4th International Conference on Software and Data Technologies (ICSOFT)*, vol. 2, pp. 199–206, January 2009.

[43] Potlapaly R. Nachiketh, Ravi Srivaths, Raghunathan Anand, Jha K. Niraj, “A Study of the Energy Consumption Characteristics of Cryptographic Algorithms and Security Protocols”, *IEEE Transactions on Mobile Computing*, vol. 5, no. 2, 128-143, December 2005.

[44] Alireza Hodjat and Ingrid Verbauwhede, “The Energy Cost of Secrets in Ad-Hoc Networks”, *Proceedings IEEE CAS Workshop Wireless Communications And Networking*, September 2002.

[45] R. Karri and P. Mishra, “Minimizing Energy Consumption of Secure Wireless Session with QoS Constraints”, *Proceedings International Conference Communication*, pp. 2053-2057, May 2002.

[46] D.S. Wong and A.H. Chan, “Mutual Authentication and Key Exchange for Low Power Wireless Communications”, *Proceedings IEEE Military Communication Conferences*, pp. 39-43, October 2001.

[47] Raj Athul, Sriram Sankaran, J. Jithish, “Modelling the Impact of Code Obfuscation on Energy Usage”, *Proceedings CEUR Workshop*, vol. 1819, 2017.

[48] Sebastian Banescu, Martin Ochoa, Alexander Pretschner, “A framework for measuring software obfuscation resilience against automated attacks”, *IEEE/ACM 1st International Workshop on Software Protection*, pp. 45-51, May 2015.

[49] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee, “Unassisted and automatic generation of high-coverage tests for complex systems programs”, *Proceedings 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI) Conference*, pp. 209-224, January 2008.

[50] Y. Wu, H. Fang, S. Wang, and Z. Qi., “A framework for measuring the security of obfuscated software”, *Proceedings of the 2010 International Conference on Test and Measurement*, ICTM, 2010.

[51] T. J. McCabe. “A complexity measure”, *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320, December 1976.

[52] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel, “Program obfuscation: a quantitative approach”, *Proceedings of the 2007 ACM Workshop on Quality of Protection*, pp. 15-20, October 2007.

[53] Timothy M. Meyers and David Binkley, “Slice-based cohesion metrics and software intervention”, *Working Conference on Reverse Engineering*, pp.256-265, 2004.

[54] Hsin-Yi Tsai, Yu-Lun Huang, D. Wagner, “A Graph Approach to Quantitative Analysis of Control-Flow Obfuscating Transformations”, *IEEE Transactions on Information Forensics and Security*, vol. 4, no. 2, pp. 257-267, 2009.

[55] A. Majumdar, S. Drape, C. Thomborson, “Metrics-based Evaluation of Slicing Obfuscations”, *International Symposium on Information Assurance and Security*, pp.472-477, 2007.

[56] M. Kainth, L. Krishnan, C. Narayana, S. G. Virupaksha, R. Tessier. “Hardware-assisted code obfuscation for fpga soft microprocessors”, *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 127-132, 2015.

[57] Dimitris Economou, Suzanne Rivoire, Partha Ranganathan and Christos Kozyrakis, “Full-system power analysis and modeling for server environments”, *Workshop on Modeling Benchmarking and Simulation*, MOBS, June 2006.

[58] C. Isci and M. Martonosi, “Runtime power monitoring in high-end processors: Methodology and empirical data”, *Proceedings 36th Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[59] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra, “Using predictive modeling for cross-program design space exploration in multicore systems”, *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT, pp. 327-338, 2007.

[60] Sriram Sankaran, “Predictive modeling based power estimation for embedded multicore systems”, *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 370-375, 2016.

[61] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, “What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps”, *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pp. 267–280, 2012.

[62] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker, “eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones”, *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pp. 57–70, 2013.

[63] Y. Liu, C. Xu, and S. Cheung, “Where has my battery gone? Finding sensor related energy black holes in smartphone applications”, *Proceedings of the 2013 IEEE International Conference on Pervasive Computing and Communications*, pp. 2–10, 2013.

[64] B. Priyantha, D. Lymberopoulos, and J. Liu, “LittleRock: Enabling energy-efficient continuous sensing on mobile phones", *IEEE Pervasive Computing*, pp. 12–15, 2011.

[65] Digvijay Singh, Peter A. H. Peterson, Peter L. Reiher, and Wiliam J. Kaiser, “The Atom LEAP platform for energy-efficient embedded computing: Architecture, operation, and system implementation", 2010.

[66] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li and L. K. John, “Using complete machine simulation for software power estimation: The SoftWatt approach”, *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pp. 141–151, 2002.

[67] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha, “Hybrid simulation for embedded software energy estimation”, *Proceedings of the 42nd annual Design Automation Conference*, pp. 23–26, 2005

[68] C. Seo, S. Malek, and N. Medvidovic, “Component-level energy consumption estimation for distributed Java-based software systems”, *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pp. 97–113, 2008.

[69] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, “Estimating mobile application energy consumption using program analysis”, *Proceedings of the 2013 International Conference on Software Engineering*, pp. 92–101, 2013.

[70] A. Litke, K. Zotos, A. Chatzigeorgiou, and G. Stephanides, “Energy consumption analysis of design patterns”, *Proceedings of the International Conference on Machine Learning and Software Engineering*, pp. 86–90, 2005.

[71] G. Procaccianti, H. Fernandez, and P. Lago, “Empirical Evaluation of Two Best-Practices for Energy-Efficient Software Development”, *Journal of System and Software*, vol. 117, pp. 185–198, July 2016.

[72] F. Alizadeh Moghaddam, G. Procaccianti, G. A. Lewis and P. Lago, “Empirical validation of cyber-foraging architectural tactics for surrogate provisioning”, *The Journal of Systems and Software*, vol. 138, no. 4, pp. 37–51, April 2018.

[73] W. Diffie, M. Hellman, “New Directions in Cryptography”, *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp.644-654, November 1976.

[74] L. Benini, G.D. Micheli, “System-Level Power Optimization: Techniques and Tools”, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 5, no. 2, pp. 115-192, April 2000.

[75] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, Amin Vahdat, “ECO systems: Managing Energy as a First Class Operating System Resource”, *Proceeding of the 10th International Conference on Architectural Support for Programming Languages and Operating System*, pp. 123-132, October 2002.

[76] Business Software Alliance, Eighth Annual BSA and IDC Global Software Piracy Study, May 2011.

[77] Neil MacDonald, Amrit Williams, “Hype Cycle for Cyberthreats”, Gartner Inc., September 2006.

[78] A. Main and P. van Oorschot, “Software protection and application security: Understanding the battleground”, 2004.

[79] Snežana Šarboh, Jovan Perić, Miljana Perić, “ZAŠTITA SOFTVERA - IZMEĈU AUTORSKOG PRAVA I PATENTA”, *18. Telekomunikacioni forum TELFOR*, Novembar 2010.

[80] ЗАКОН О АУТОРСКОМ И СРОДНИМ ПРАВИМА, "Сл. гласник РС", бр. 104/2009, 99/2011 и 119/2012.

[81] "Zakon o patentima”, Službeni list SCG, br.32/2004.

[82] "Zaštita softvera”, CCERT-PUBDOC-2004-04-71.

[83] William Feng Zhu, “Concepts and Techniques in Software Watermarking and Obfuscation”, University of Auckland, The Department of Computer Sciences, August 2007.

[84] G. Hachez, “A comparative study of software protection tools suited for e-commerce with contributions to software watermarking and smart cards”, Ph.D. dissertation, Universite Catholique de Louvain, March 2003.

[85] Christian Collberg, Clark Thomborson and Douglas Low, “On the limits of software watermarking”, *Technical Report #164*, Department of Computer Science, The University of Auckland, 1998.

[86] Jasvir Nagra, Christian Thomborson and Clark Collberg, “Software watermarking: Protective terminology”, *Proceedings of the ACSC*, 2002.

[87] Christian Collberg and Clark Thomborson, “Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection”, IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 735-746, August 2002.

[88] Stephen Drape, “Generalizing the array split obfuscation”, Information Sciences, vol. 177, no. 1, pp. 202–219, January 2007.

[89] Stephen Drape, Oege de Moor and Ganesh Sittampalam, “Transforming the .NET Intermediate Language using Path Logic Programming”, *Principles and Practice of Declarative Programming*, ACM, pp. 133–144, 2002.

[90] Y. He, “Tamperproofing a software watermark by encoding constants”, University of Auckland, March 2002.

[91] Phillipe Biondi and Fabrice Desclaux, “Silver needle in the skype”, Presentation at BlackHat Europe, March 2006.

[92] Nuno Santos, Pedro Pereira, and Luis Moura e Silva, “A Generic DRM Framework for J2ME Applications”, *First International Mobile IPR Workshop: Rights Management of Information (MobileIPR),* pp. 53–66. Helsinki Institute for Information Technology, August 2003.

[93] Christian Collberg, Clark Thomborson and Douglas Low, “Manufacturing cheap, resilient, and stealthy opaque constructs”, *Proceeding of the 25th ACM Symposium on Principles of Programming Languages (POPL 1998)*, pp. 184–196, January 1998.

[94] Christian Collberg, Clark Thomborson and Douglas Low, “A taxonomy of obfuscating transformation”, Technical report 148, Department of computer science, the University of Auckland, Auckland, New Zealand, 1997.

[95] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang, “On the (Im)possibility of obfuscating programs”, *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pp. 1–18. Springer-Verlag, 2001.

[96] Warren A. Harrison and Kenneth I. Magel, “A complexity measure based on nesting level”, SIGPLAN Notices, vol. 16, no. 3, pp. 63–74, 1981.

[97] D. Low, “Java control flow obfuscation”, Master’s thesis, University of Auckland, New Zealand, 1998.

[98] IOCCC. The International Obfuscated C Code Contest. URL: www.ioccc.org (24.5.2019.)

[99] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Flippo Ricca, Marco Torchiano and Paolo Tonella, “The effectiveness of source

code obfuscation: an experimental assessment", *IEEE International Conference on Program Comprehension (ICPC 2009)*, IEEE CS Press, June 2009.

[100] Jien Tsai Chan and Wuu Yang, "Advanced obfuscation techniques for java bytecode", *Journal of Systems and Software*, ELSEVIER vol. 71, no. 1, pp. 1-10, April 2004.

[101] Tyma P. "Method for renaming identifiers of a computer program", US patent 6,102,966 (2000).

[102] http://www.babelfor.net/products/obfuscator (27.5.2019.)

[103] Anirban Majumdar, Clark D. Thomborson, and Stephen Drape, "A survey of control-flow obfuscations", *2nd International Conference on Information Systems Security*, pp. 353–356, December 2006.

[104] Madou Matias, Anckaert Bertrand, Bruno De Bus, Koen De Bosschere, Jan Cappaert and Bart Preneel, "On the Effectiveness of Source Code Transformations for Binary Obfuscation", *International Conference on Software Engineering Research and Practice*, June 2006.

[105] Chow Stanley, Gu Yuan, Johnson Harold and Zakharov A. Vladimir, "An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs", *In the proceedings of 4th International Conference on Information Security*, SPRINGER, vol. 2200, pp. 144-155, 2001.

[106] Sebastian Schrittwieser and Stefan Katzenbeisser, "Code Obfuscation against Static and Dynamic Reverse Engineering", Vienna University of Technology, Austria, Darmstadt University of Technology, Germany.

[107] Stephen Drape, "Intelectual Property Protection Using Obfuscation", Research Project sponsored by Siemens AG, Munich, Collaboration between Siemens, Munich and the University of Oxford, CS-RR-10-02, March 2009.

[108] Anirban Majumdar and Clark Thomborson, "Manufacturing opaque predicates in distributed systems for code obfuscation", *Proceedings of the 29th Australasian Computer Science Conference*, Australian Computer Society , pp. 187–196, 2006.

[109] J. Palsberg, S. Krishnaswamy, K. Minseok, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking", *Proceedings of the 16th Annual Computer Security Applications Conference*, IEEE, pp. 308–316, 2000.

[110] Stephen Drape, "Obfuscation of Abstract Data-Types", PhD thesis, Oxford University Computing Laboratory, 2004.

[111] Anirban Majumdar, Stephen J. Drape, and Clark D. Thomborson, "Slicing obfuscations: design, correctness, and evaluation", *Proceedings of the 2007 ACM workshop on Digital Rights Management*, ACM, pp. 70–81, 2007.

[112] Chenxi Wang, Jonathan Hill, John C. Knight and Jack W. Davidson. "Protection of software-based survivability mechanisms", *Proceedings of the 2001 International Conference on Dependable Systems and Networks*, IEEE Computer Society , pp. 193–202, 2001.

[113] Matias Madou, Ludo Van Put, and Koen De Bosschere. "Understanding obfuscated code", *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pp. 268–271, IEEE, 2006.

[114] Stephen Drape and Anirban Majumdar, "Design and evaluation of slicing obfuscations", Technical Report 311, CDMTCS, The University of Auckland, June 2007.

[115] Stephen Drape, Clark Thomborson and Anirban Majumdar, "Specifying imperative data obfuscations", *Proceedings of the 10th Information Security Conference (ISC '07), volume 4779 of Lecture Notes in Computer Science*, SPRINGER, pp. 299–314, 2007.

[116] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious pat- terns", *In 12th USENIX Security Symposium*, pp. 169–186, August 2003.

[117] Ilus You and Kangbin Yim, "Malware Obfuscation Techniques - A Brief Survey", *IEEE International Conference on Broadband, Wireless Computing, Communication and Application*, IEEE, pp. 297-300, November 2010.

[118] Wing Wong and Mark Stamp, "Hunting for Metamorphic Engines", *Journal in Computer Virology*, vol. 2, no. 3, pp. 211-229, December 2006.

[119] Arini Balakrishnan and Chloe Schulze, "Code Obfuscation Literature Survey", 2005.

[120] Evgenios Konstantinou, "Metamorphic Virus: Analysis and Detection", RHUL-MA-2008-02, Technical Report of University of London, January 2008.

[121] Marius Popa, "Characteristics of Program Code Obfuscation for Reverse Engineering of Software", P*roceedings of the 4th International Conference on Security for Information Technology and Communications*, Academy of Economic Studies and Military Technical Academy, pp. 103-112, January 2011.

[122] Flemming Nielson, Hanne R. Nielson and Chris Hankin, "Principles of Program Analysis", Springer-Verlag, 1999.

[123] Yuschuk, OllyDbg [Online]. Available: http://www.ollydbg.de (20.9.2015.), 2015.

[124] Inc. Immunity. Immunity debugger. http://debugger.immunityinc.com

[125] Microsoft. Windbg, https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools (27.5.2019.)

[126] Hex-Rays SA, The IDA Pro disassembler and debugger, www.hex-rays.com/idapro (24.5.2019.), 2019.

[127] J. R. Levine, "Linkers and Loaders", Morgan Kaufman Publishers, 2000.

[128] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell and Mark Horowitz, "Architectural support for copy and tamper resistant software", *Proceeding of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM, vol. 28, no. 5, pp. 168–177, December 2000.

[129] Christian Collberg and Clark Thomborson, "Software watermarking: Models and dynamic embeddings", *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 1999)*, pp. 311-324, January 1999.

[130] Christian Collberg and Clark Thomborson, "Watermarking, tamper-proofing, and obfuscation – tools for software protection", *Technical Report TR00-03*, The Department of Computer Science, University of Arizona, February 2000.

[131] Christian Collberg, Clark Thomborson, and Douglas Low, "Breaking abstractions and unstructuring data structures", *Proceedings of the IEEE International Conference on Computer Languages*, pp. 28-38, 1998.

[132] G. Wroblewski, "General Method of Program Code Obfuscation", *PhD thesis*, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[133] W. Cho, I. Lee, and S. Park, "Against intelligent tampering: Software tamper resistance by extended control flow obfuscation", *Proceedings World Multiconference on Systems, Cybernetics, and Informatics*. International Institute of Informatics and Systematics, 2001.

[134] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. "Software obfuscation on a theoretical basis and its implementation", *IEEE Transactions Fundamentals*, E86-A(1), January 2003.

[135] C. Wang, J. Hill, J. Knight, and J. Davidson, “Software tamper resistance: Obstructing static analysis of programs”, *Technical Report CS-2000-12*, December 2000.

[136] Eldad Eilam, “Reversing: Secrets of reverse engineering”, WILEY, 2005.

[137] Mihai Doinea, Sorin Pavel, “Security Optimization for Distributed Applications Oriented on Very Large Data Sets”, *Informatica Economică*, vol. 14, no. 2, pp. 72-85, 2010.

[138] Paul Pocatilu, “Android Applications Security”, *Informatica Economică*, vol. 15, no. 3, pp. 163-171, 2011.

[139] Cagri Sahin, “Empirical Investigating Energy Impacts Of Software Engineering Decisions”, *PhD Thesis*, University of Delaware, 2017.

Прилози

Прилог 1: Исечак модела инструкција из [31]

Number	Instruction	Base Cost (mA)
1	NOP	275.7
2	MOV DX,BX	302.4
3	MOV DX,[BX]	428.3
4	MOV DX,[BX][DI]	409.0
5	MOV [BX],DX	521.7
6	MOV [BX][DI],DX	451.7
7	ADD DX,BX	313.6
8	ADD DX,[BX]	400.1
9	ADD [BX],DX	415.7
10	SAL BX,1	300.8
11	SAL BX,CL	306.5
12	LEA DX,[BX]	364.4
13	LEA DX,[BX][DI]	345.2
14	JMP label	373.0
15	JZ label	375.7
16	JZ label	355.9
17	CMP BX,DX	298.2
18	CMP [BX],DX	388.0

У прилогу 1, приказана је листа анализираних инструкција за процесор Intel 486DX2, са просечном јачином струје за сваку понаособ. Мерење је реализовано захваљујући посебно развијеном харверском инструментацијом детаљније описаном у раду [31].

Прилог 2: Мерење потрошње електричне енергије

У овој секцији, дате су дефиниције електричне снаге и електричне енергије и њихове формуле.

Снага [139] **:** Представља брзину промене потрошње електричне енергије с временом, или брзину којом се енергија преноси путем електричног кола. Мери се у ватима [W], и израчунава се као производ електричног потенцијала (напона) и електричне струје, по формули (П2.1):

$$P = V*I \qquad (П2.1)$$

где су:

- P - електрична снага, мери се у ватима [W],
- V - напон, мери се у волтима [V],
- I - јачина струја, мери се у амперима [A].

Енергија [139]: Представља производ електричне снаге у времену. Мери се у џулима [J] који су еквивалентни ват-секундама. Потрошња електричне енергије се израчунава по формули (П2.2):

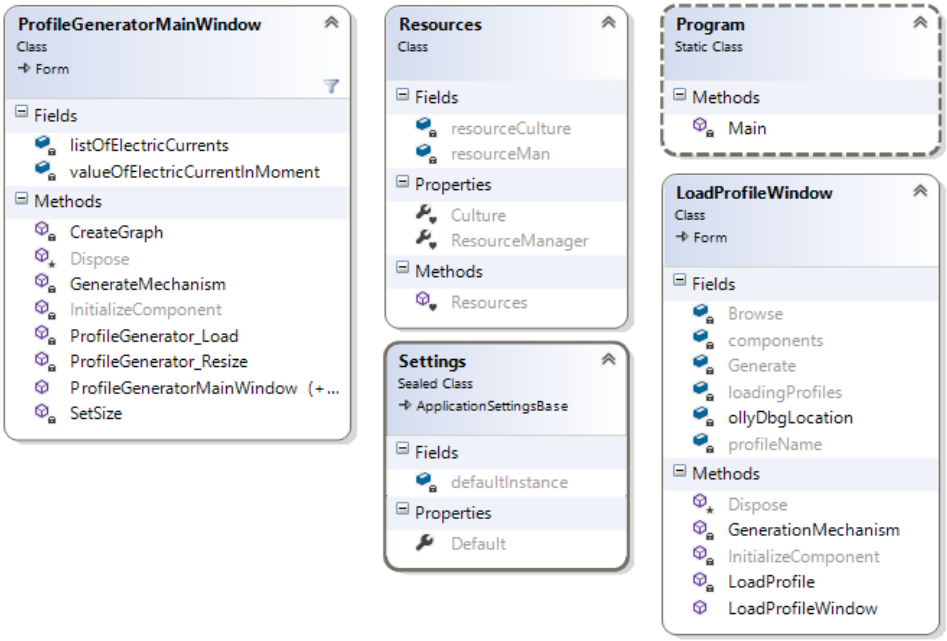
$$W = P*T \qquad (П2.2)$$

где су:

- W - електрична енергија, мери се у џулима [J],
- P - електрична снага, мери се у ватима [W],
- I - јачина струја, мери се у амперима [A].

Прилог 3: Класни дијаграм генератора профила

У прилогу 3, дат је приказ дијаграма класа апликације за генерисање профила потрошње, као и објашњења најбитнијих.



- *ProfileGeneratorMainWindow* - представља класу главног прозора која је задужена за учитавање асемблерских инструкција, као и генерисање самог профила потрошње.
  - *listOfElectricCurrents* - листа вредности јачина струје из модела дефинисаног у раду [31],
  - *valueOfElectricCurrentInMoment* - тренутна вредност јачине струје која се обрађује,
  - *CreateGraph* - метода за креирање графика профила потрошње,
  - *GenerateMechanism* - метода за позивање механизма за генерисање профила,
  - *ProfileGenerator_Load, SetSize ProfileGenerator_Resize, ProfileGeneratorMainWindow* - методе за рад са графичким окружењем (прозором).

- *LoadProfileWindow* - представља класу за учитавање датотека које садрже оригиналне и обфусковане асемблерске кодове, на основу којих се генеришу профили потрошње.
    - *GenerationMechanism* - метода задужена за позивање класе заслужне за генерисање профила, односно *ProfileGeneratorMainWindow,*
    - *profileName* - име профила који се обрађује,
    - *ollyDbgLocation* - веза са диссемблираним вредностима, тј. асемблерским кодом,
    - *LoadProfileWindow* - метода за манипулацију са прозором,
    - *LoadProfile* - метода за учитавање профила

Биографија

Марко М. Ђуковић, рођен је 27.септембра 1985. године. Дипломски-мастер рад из области електротехнике и рачунарства а на тему "*Windows Mobile* решење програмске подршке за управљање аукцијским процесом трговине електричне енергије", одбранио је 2009. године на Факултету техничких наука у Новом Саду. Добитник је награде "Доситеја", фонда за младе таленте Републике Србије 2008. године.

Од 2009-2012, радио је у компанији "*Telvent DMS*" као развојни инжењер из домена управљања и трговине електричне енергије. У периоду од 2012. године до данас, запослен је у компанији "*Schneider Electric DMS NS*". Тренутно се налази у сектору за истраживање и развој, на позицији руководиоца тима за базне прорачуне преносне мреже.

Године 2010. уписао је докторске студије на студијском програму Енергетика, електроника и телекомуникације, на Факултету техничких наука у Новом Саду.

Коаутор је два научна рада.