Paolo Monti   Alessandro Salvato   Michele Simili

POLITECNICO DI TORINO

Computer Architectures

Defeating Hardware Trojan through Software Obfuscation

Professor: Ernesto Sanchez

Data: May 9, 2017

Contents

1 Introduction

Due to the increasing costs of manufacturing chips smaller and smaller transistors, the vast majority of companies is forced to employ and trust a third party to fabricate their design. This makes them very susceptible to malicious circuitry injected into their chip at fabrication time. We want to focus on digital hardware that can cause unintended behaviour (trojan). The project is divided into two parts: understanding how a digital hardware trojan works by developing an example, and how to mitigate or deny its malicious effects. This report will focus on the second part only. This final report aims at gathering the result of applying an evolutionary algorithm, run by uGP3, to some assembly programs (study cases) in a miniMips environment, in order to avoid the activation of an hardware trojan inside it. Our prototype trojan raises a payload when a certain sequence of instructions I1-I2-I3 is detected. Obviously, we don't know the exact sequence, so our aim is to mitigate the activation of a trojan triggered by an unknown sequence.
The starting population generated by uGP is a sequence of integers, between 0 and 99, one for each line of the starting code. Each one of there numbers represent a gene of the individual, effectively making up its DNA.
Then, the external evaluator takes control, with the final aim of defining the fitness values for the given individual. These parameters are 3, and uGP aims at maximizing them all in a multi-objective way.
First of all, the starting code is modified according to the DNA of the individual, with some rules defined in the codeShuffler.py script. The basic operations that can be applied to a line of code are 3. The first one simply adds a line with a NOP instruction, that where the processors does nothing relevant. The second one swaps the current line with the line above or below it. The last one, which is the most important, substitutes a line of code with one or more lines that are equivalent, i.e. they have the same output. The rules for substitution of an instruction can be found in Appendix A.
Applying all these rules, a new source file, modifiedCode.src, is generated. This file is now taken by the miniMips assembler and converted into a binary file.
The code is then simulated by the miniMips processor and some data is collected. Specifically, we are interested about the state of the data memory at the end of the simulation, the total elapsed time, and the number of accesses made to the memory by load and store instructions.
This information is then taken by another script, that finally computes the fitness based on their values. The most important thing is checking that the output of the program is the same as the one of the starting program. If they are different, the individual is discarded by setting its fitness values to 0. If their outputs are the same, the Jaccard Index between startingCode and modifiedCode is computed. The Jaccard Index is a parameter that represents how much different two text files are. In our case, it is very useful because it allows to check whether or not the new code has really changed.
Once it is computed, the final fitness indicators are the Jaccard Index, the reciprocal of the simulation time, and the reciprocal of the number of memory accesses. The last two are reciprocated because the evolutionary algorithm aims at maximizing fitness, while we want to minimize (or at least keep low) them.
As soon as uGP receives the fitness.out file for each individual of the population, it computes the individuals of the next population based on their fitness, and the cycle starts again.
The following image highlights the overall execution flow.

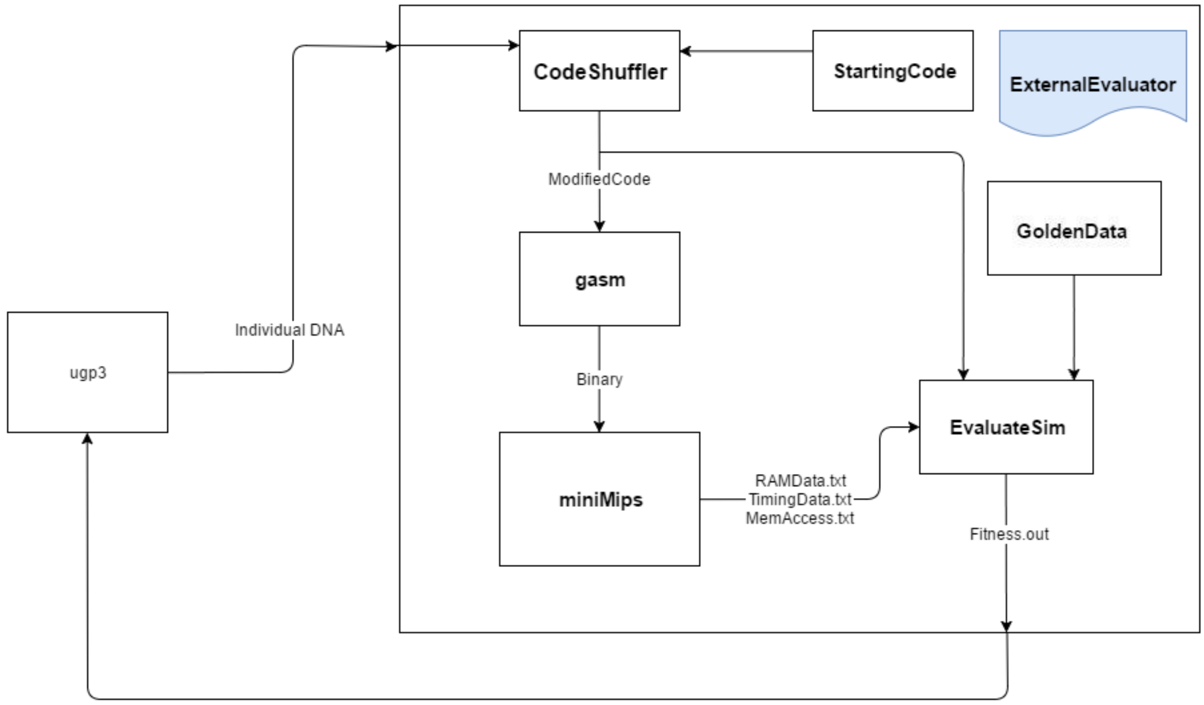

Figure 1: uGP execution flow

2 Study cases

2.1 Unrolled Fibonacci

```
ADDI $31, $0, 0
LW $1, 1024($31)        # $1 <- nth

LUI $31, 1
SRL $31, $31, 14        # = 4
LW $2, 1024($31)        # $2 <- nth+1 (it's the index in the Fibonacci's serie)

ADD $1, $2, $1          # 2
SLL $31, $31, 1         # index <- 4*2
SW $1, 1024($31)

ADD $2, $2, $1          # 3
ADDIU $31, $31, 4       # index <- 12
SW $2, 1024($31)        #

ADD $1, $2, $1          # 5
ADDI $31, $31, 4        # index <-16
SW $1, 1024($31)

ADD $2, $2, $1          # 8
ADDIU $31, $31, 4       # index <- 20
SW $2, 1024($31)        #

ADD $1, $2, $1          # 13
ADDI $31, $31, 4        # index <- 24
SW $1, 1024($31)

ADD $2, $2, $1          # 21
ADDIU $31, $31, 4       # index <- 28
SW $2, 1024($31)        #

ADD $1, $2, $1          # 44
ADDI $31, $31, 4        # index <- 32
SW $1, 1024($31)

ADD $2, $2, $1          # 65
ADDIU $31, $31, 4       # index <- 36
SW $2, 1024($31)        #

end: j end
```

This program takes as inputs 2 numbers (i-th and i-th+1) owning to Fibonacci's series and generates the next 8 values. They have to store in the first and second ram location, respectively. If the RAM starts at 1024, in terms of global addressing, we'll have

$$1024(\$0) <= ith$$
$$1028(\$0) <= ith + 1$$

The outputs are stored in RAM, as well. To see them you should read the content of the first tem RAM locations.

$$\begin{aligned} 1024(\$0) &<= ith \\ 1028(\$0) &<= ith + 1 \\ 1032(\$0) &<= ith + 2 \\ 1036(\$0) &<= ith + 3 \\ &\ldots \end{aligned}$$

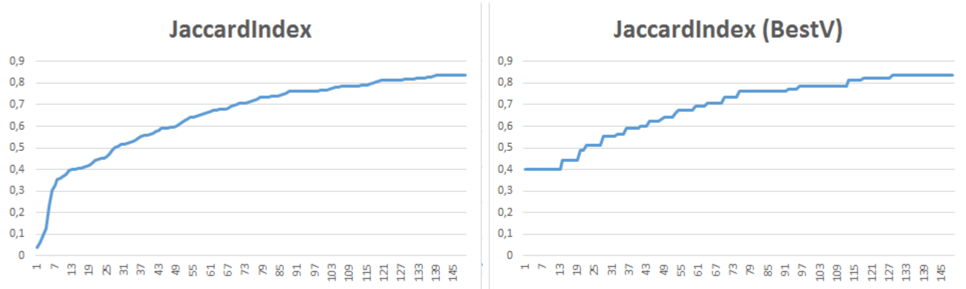

Figure 2: Jaccard Index

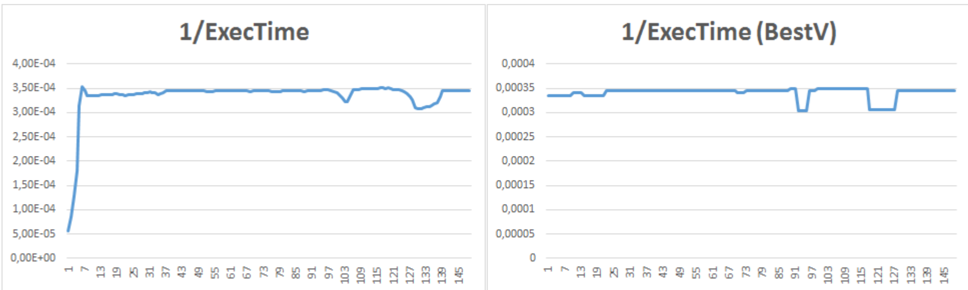

Figure 3: Execution Time

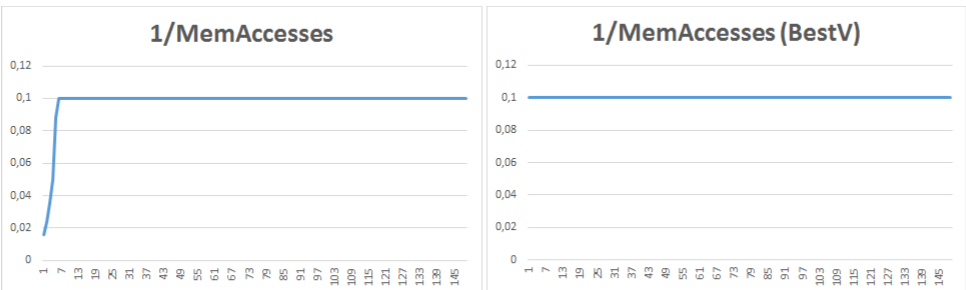

Figure 4: Memory Accesses

In the initial code, there were 28 tuples. In the final code, only 1 of those tuples are left. 96.42857% of possible trojans have been mitigated. In fact, as said before, we don't know the actual sequence of instruction that actives the trojan. Assuming that the initial code contains one raising tuple, at the end of the process we have obtained a best (final) code where the probability of activating the malicious behaviour is dropped to around 95%. We say it because we assumed that one of the 28 initial tuples was the offending one.

2.2 Average of 10 numbers

```
                addi $1, $0, 10        # initialize loop counter to 0
                addi $5, $0, 0         # initialize accumulator to 0
label:          addi $1, $1, -1
                lw $3, 1024($2)        # load from memory
                add $5, $3, $5         # accumulate
                addi $2, $2, 4
                bne $1, $0, label      # loop
                sll $10, $5, 7         # x * 128
                add $6, $10, $0        # 128 x
                sll $11, $5, 6         # x * 64
                add $6, $11, $6        # 192 x
                sll $12, $5, 3         # x * 8
                add $6, $12, $6        # 200 x
                sll $13, $5, 2         # x * 4
                add $6, $13, $6        # 204 x
                add $6, $5, $6         # 205 x
                srl $5, $6, 11         # 205x/2048 = x/10
                sw $5, 1024($0)
                sll $0, $0, 0
end:            j end
```

In the initial code, there were 20 tuples. In the final code, only 0 of those tuples are left. 100% of possible trojans have been mitigated. In fact, as said before, we don't know the actual sequence of instruction that actives the trojan. Assuming that the initial code contains one raising tuple, at the end of the process we have obtained a best (final) code that doesn't enable the malicious behaviour. We say it because we assumed that one of the 20 initial tuples was the offending one.

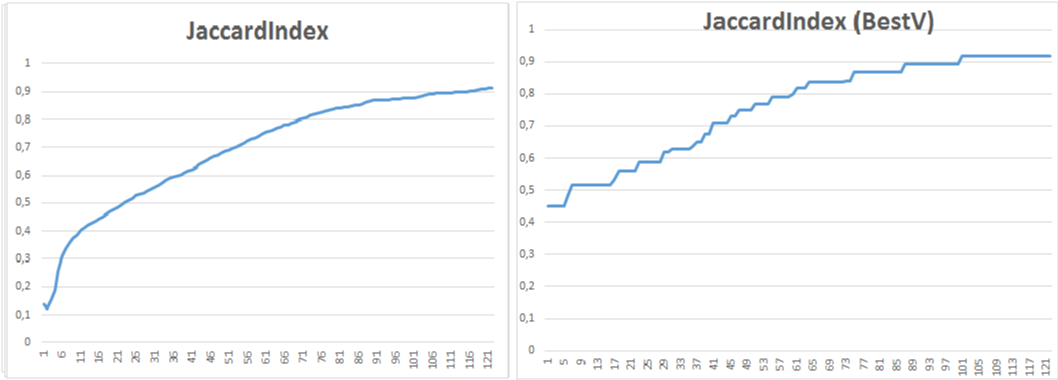

Figure 5: Jaccard Index

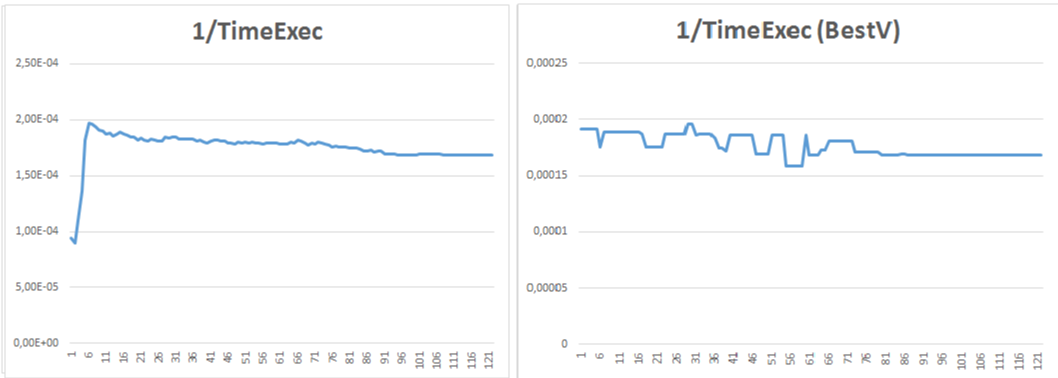

Figure 6: Execution Time

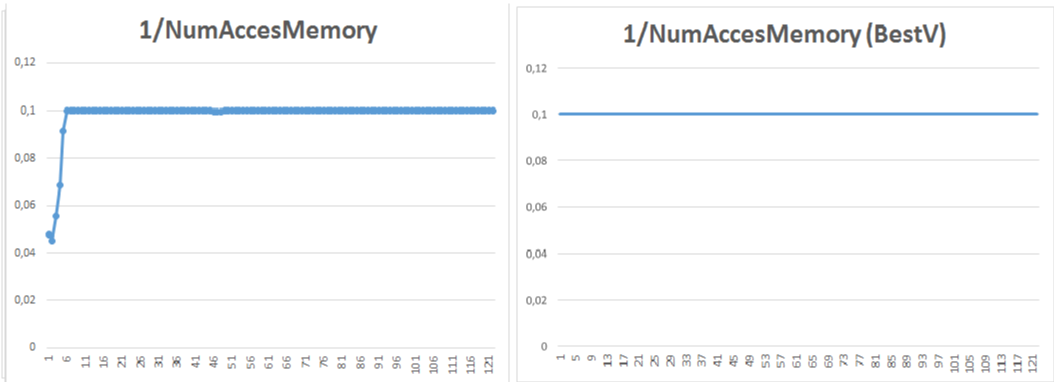

Figure 7: Memory Accesses

2.3 Factorials

```
                addi $1, $0, 10        #counter elements
                addi $2, $0, 0         #pointer elements

loopelem:       addi $1, $1, -1        #decrease counter fact
                lw $3, 1024($2)        #load next value at..
                                       #the address 1024 + $2(pointer)
                addi  $4, $0, 2        #initialize factorial counter for...
                                       # multiplications (starting from 2)
                addi $5, $0, 1         #acc of fact(accf)
                addi $3,$3,-1          #decrement the counter by one(starting...
                                       #from 2 we have to compute A-1 multiplications
                add $8,$0,$0           #initialize the multiplication counter...
                                       #(multiplication implemented as a loop of addition

loopfact:       addi $3,$3,-1          #decrement the factorial counter
                add  $6, $0, $5        #accumulator of factorial -> operand(that needs...
                                       #to be summed continuosly to implement mult)
                add  $7, $0, $4        #pointerf -> counterm
                add  $8, $0, $0        #set the multiplication accumulator to 0
                addi $4, $4, 1         #increment by one the factorial counter

loopmul:        addi $7, $7, -1        #decrease counter mul
                add  $8, $8, $6        #accm=accm+operand
                bne $7, $0, loopmul    #if not finish, keep adding
                add $5 , $0, $8 #update the accf with the result of mult
                bne $3, $0, loopfact   #if not finish, keep multiplying
                sw $5, 1024($2)        #store fact value to the same cell of memory
                addi $2, $2, 4         #increase pointer fact
                bne $1, $0, loopelem   #if not finish, keep taking elements from memory

end:            j end                  #conventional return instructional
```

In the initial code, there were 39 tuples. In the final code, only 10 of those tuples are left. 74.35897% of possible trojans have been mitigated. In according to written in the previous subsection, in this study case we have reduced the possibility of trojan enabling of around 74%, but the threat is not vanquished at all.

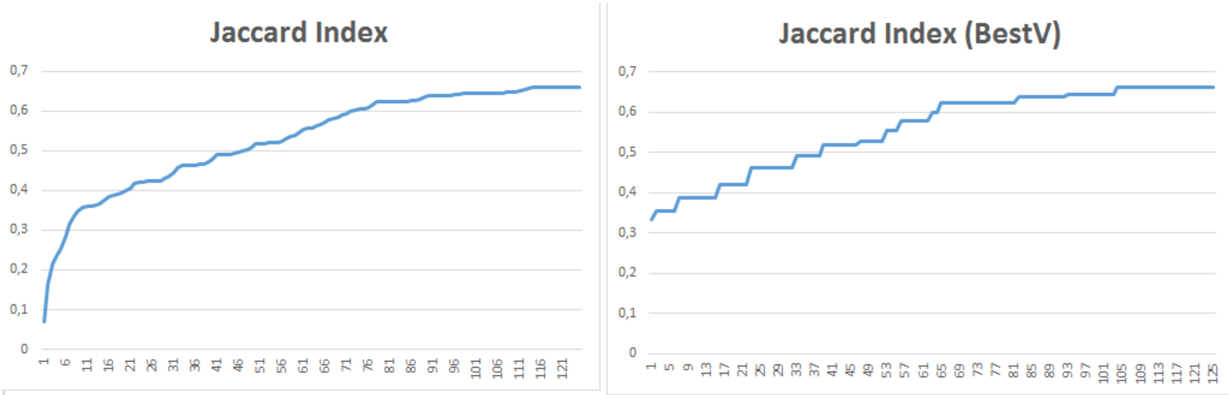

Figure 8: Jaccard Index

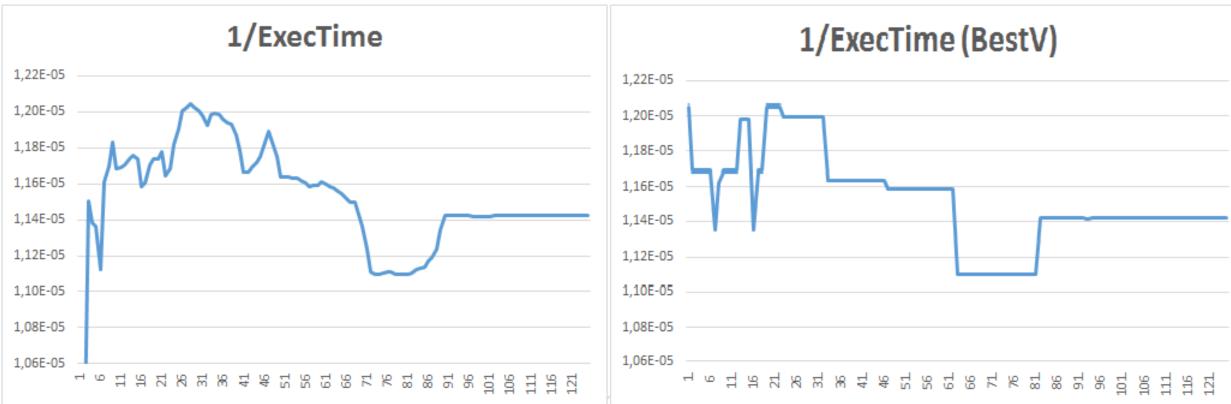

Figure 9: Execution Time

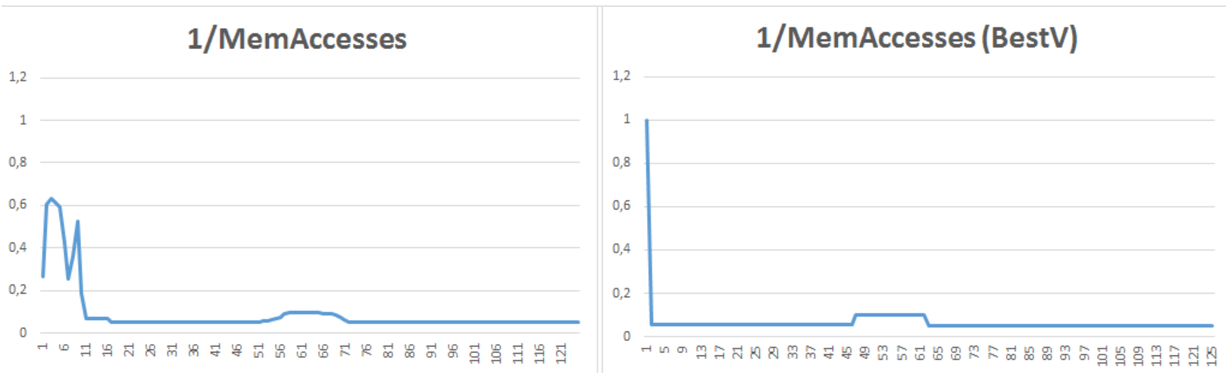

Figure 10: Memory Accesses

2.4 Matrix product

```
XOR $31, $31,$31         # i
XOR $30, $30, $30        # j
XOR $29, $29, $29        # k
ADDI $28, $0, 2          # dim matrixes
XOR $10, $10, $10        # counter
label:
           XOR $30, $30, $30

           label2:
                        XOR $29, $29, $29
                        label3:

                                     LW $3, 1056($0)
                                     SLL $31, $31, 3
                                     SLL $29, $29, 2
                                     ADD $27, $31, $29
                                     LW $1, 1024($27)        #mat1[i][k]
                                     SRL $31, $31, 3
                                     SRL $29, $29, 2
                                     SLL $29, $29, 3
                                     SLL $30, $30, 2
                                     ADD $26, $30, $29
                                     LW $2, 1040($26)        #mat2[k][j]
                                     SRL $29, $29, 3
                                     SRL $30, $30, 2
                                     MULT $1, $2
                                                             # MFHI $2
                                     MFLO $4                 # STRONG ipothesis:
                                     #multiplication generates a number max 32-bit lon

                                     ADD $3, $3, $4
                                     SW $3, 1056($0)
                                     ADDI $29, $29, 1
                                     SLL $0, $0, 0
                                     bne $29, $28, label3
                                     SLL $0, $0, 0
                        LW $3, 1056($0)
                        SW $3, 1060($10)
                        SW $0, 1056($0)
                        ADDI $10, $10, 4
                        ADDI $30, $30, 1
                        SLL $0, $0, 0
                        bne $30, $28, label2
                        SLL $0, $0, 0
           ADDI $31, $31, 1
           SLL $0, $0, 0
           bne $31, $28, label
           SLL $0, $0, 0
XOR $10, $10, $10
ADDI $2, $0, 16
label4: LW $1, 1060($10)        #this cycle is very useful only for debug purposes
        SW $1, 1024($10)
        ADDI $10, $10, 4
        SLL $0, $0, 0
        BNE $10, $2, label4

SLL $0, $0, 0
```

```
end: j end
```

This program performs the matrix product between two matrices 2×2. Let's imagine to have the following situation:

$$\begin{bmatrix} a_1 & a_2 \\ a_3 & a_4 \end{bmatrix} \times \begin{bmatrix} b_1 & b_2 \\ b_3 & b_4 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 \\ c_3 & c_4 \end{bmatrix}$$

To use that, it is necessary to provide inputs a_i and b_i, in according to the following implementative choice:

$$\begin{array}{ll} 1024(\$0) <= a1 & 1040(\$0) <= b1 \\ 1028(\$0) <= a2 & 1044(\$0) <= b2 \\ 1032(\$0) <= a3 & 1048(\$0) <= b3 \\ 1036(\$0) <= a4 & 1052(\$0) <= b4 \end{array}$$

At the end, A matrix locations are replaced by the elements of the result one.
The execution time is estimated between 25000 ns and 26000 ns. This program uses RAM addresses in range [1024, 1072].

In the initial code, there were 67 tuples. In the final code, only 15 of those tuples are left. 77.61194% of possible trojans have been mitigated.

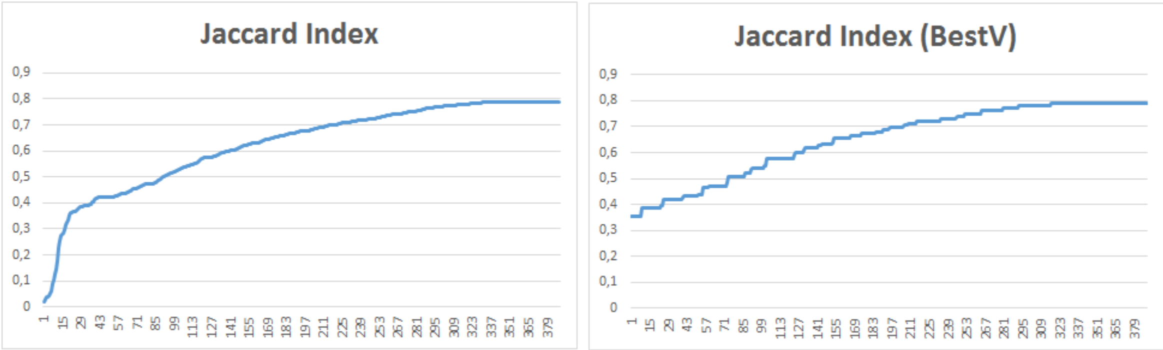

Figure 11: Jaccard Index

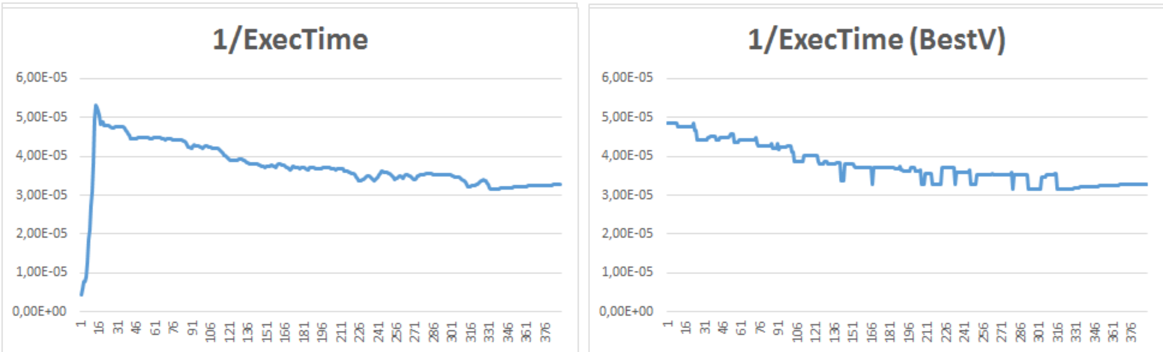

Figure 12: Execution Time

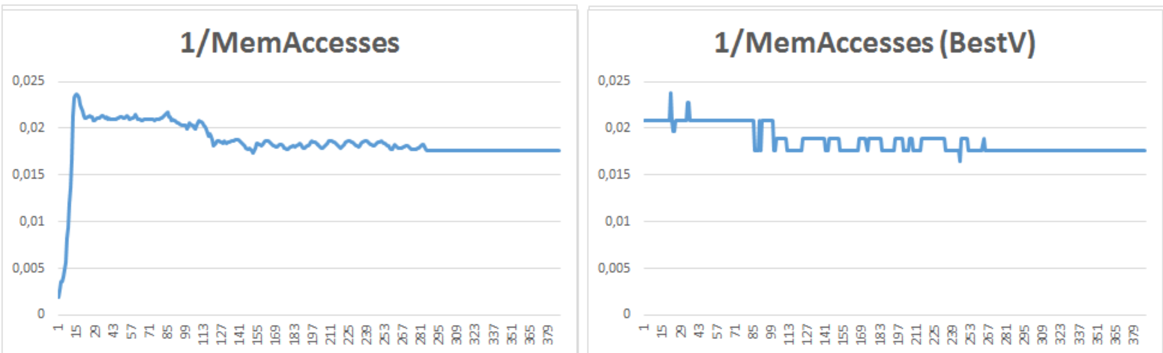

Figure 13: Memory Accesses

In the initial code, there were 67 tuples. In the final code, only 15 of those tuples are left. 77.61194% of possible trojans have been mitigated.

A Instruction substitution table

Instruction format	Substitution 1 code	Substitution 2	Substitution 3
ADD Rd, Rs, Rt	ADDU Rd, Rs, Rt	X	X
ADDI Rt, Rs, N	ADDIU Rt, Rs, N	X	X
ADDIU Rt, Rs, N	ADDI Rt, Rs, N	X	X
ADDU Rd, Rs, Rt	ADD Rd, Rs, Rt	X	X
AND Rd, Rs, Rt	ANDI Rs, Rs, -1 ANDI Rt, Rt, -1 AND Rd, Rs, Rt	XORI Rs, Rs, -1 XORI Rt, Rt, -1 OR Rd, Rs, Rt XORI Rd, Rd, -1 XORI Rs, Rs, -1 XORI Rt, Rt, -1	X
ANDI Rt, Rs, N	XORI Rs, Rs, -1 ADD Rt, 0, Rs $ADDIRs$,0, N XORI Rs, Rs, -1 NOR Rt, Rt, Rs	ADDI Rt, $0, N XORI Rt, Rt, -1 XORI Rs, Rs, -1 NOR Rt, Rt, Rs XORI Rs, Rs, -1	X

BEQ Rs, Rt, offset	XOR Rs, Rs, Rt
	BEQ Rs, $0, offset
		ADDI Rs, Rs, 1
		ADDI Rt, Rt, 1
		BEQ Rs, Rt, offset
			ANDI Rs, Rs, -1
			BEQ Rs, Rt, offset

BGEZ Rs, offset	BEQ Rs, $0, offset BGTZ Rs, offset	BEQ Rs, $0, offset BGEZ Rs, offset	X
BGEZAL Rs, offset	AND Rs, Rs, -1 BGEZAL Rs, offset	OR Rs, Rs, 0 BGEZAL Rs, offset	X
BGTZ Rs, offset	AND Rs, Rs, -1 BGTZ Rs, offset	OR Rs, Rs, 0 BGTZ Rs, offset	X
BLEZ Rs, offset	BEQ Rs, $0, offset BLTZ Rs, offset	AND Rs, Rs, -1 BLEZ Rs, offset	OR Rs, Rs, 0 BLEZ Rs, offset
BLTZ Rs, offset	AND Rs, Rs, -1 BLTZ Rs, offset	OR Rs, Rs, 0 BLTZ Rs, offset	X
BLTZAL Rs, offset	AND Rs, Rs, -1 BLTZAL Rs, offset	OR Rs, Rs, 0 BLTZAL Rs, offset	X
BNE Rs, Rt, offset	XOR Rs, Rs, Rt BNE Rs, $0, offset	X	X
BREAK	X	X	X
COP0 cop_func	X	X	X
J target	X	X	X
JAL target	X	X	X
JALR Rd, Rs	X	X	X
JR Rs	AND Rs, Rs, Rs JR Rs	X	X
LUI Rt, N	XOR Rs, Rs, Rs ADDI Rs, Rs, N SLL Rs, Rs, 16	X	X
LW Rt, offset	XOR Rt, Rt, Rt AND Rt, Rt, Rt LW Rt, offset	X	X
LWC0 Cs, offset	X	X	X
MFC0 Rt, Cs	X	X	X
MFHI Rd	XOR Rd, Rd, Rd MFHI Rd	X	X
MFLO Rd	XOR Rd, Rd, Rd MFLO Rd	X	X

Table 1: Instruction possible substitutions - I

Instruction format	Substitution 1 code	Substitution 2	Substitution 3
MTC0 Rt, Cs	X	X	X
MTHI Rs	X	X	X
MTLO Rs	X	X	X
MULT Rs, Rt	MULTU Rs, Rt	X	X
MULTU Rs, Rt	MULT Rs, Rt	X	X
NOR Rd, Rs, Rt	OR Rd, Rs, Rt XORI Rd, Rd, -1	X	X
OR Rd, Rs, Rt	NOR Rd, Rs, Rt XORI Rd, Rs, -1	X	X
ORI Rt, Rs, N	ADDI Rt, $0, N NOR Rt, Rt, Rs	X	X
SLL Rd, Rt, N	ADDI Rd, $0, N SLLV Rd, Rd, Rt	X	X
SLLV Rd, Rt, Rs	ADDI Rd, $0, Rs SLL Rs, Rs, 1 SLLV Rd, Rt, Rs	X	X
SLT Rd, Rs, Rt	SUB Rd, Rs, Rt SRL Rd, Rd, 31	X	X
SLTI Rt, Rs, N	ADDI Rt, Rs, -N SRL Rt, Rt, 31	X	X
SLTIU Rt, Rs, N	ADDIU Rt, $0, N SLTU Rt, Rs, Rt	X	X
SLTU Rd, Rs, Rt	X	X	X
SRA Rd, Rt, N	ADDI Rd, $0, N SRAV Rd, Rt, Rd	X	X
SRAV Rd, Rt, Rs	SRA Rt, Rt, 1 ADDI Rs, Rs, -1 SRAV Rd, Rt, Rs	X	X
SRL Rd, Rt, N	ADDI Rd, $0, N SRLV Rd, Rt, Rd	X	X
SRLV Rd Rt, Rs	X	X	X

SUB Rd, Rs, Rt	XORI Rt, Rt, -1
	ADD Rd, Rs, Rt
	ADDI Rd, Rd, 1
	XORI Rt, Rt, -1
		X	X

SUBU Rd, Rs, Rt	SUB Rd, Rs, Rt	X	X
SW Rt, offset	XOR Rd, Rd, $6 XOR $6, $6, Rt XOR Rt, Rt, $6 SW $6, offset XOR Rd, Rd, $6 XOR $6, $6, Rt XOR Rt, Rt, $6	X	X
SWC0 Cs, offset	X	X	X
SYSC	X	X	X
XOR Rd, Rs, Rt	XORI Rt, Rt, -1 AND Rd, Rs, Rt XORI Rt, Rt, -1 XORI Rs, Rs, -1 AND Rs, Rs, Rt OR Rd, Rd, Rs XORI Rs, Rs, -1	X	X
XORI Rt, Rs, N	ADDI Rt, $0, N XOR Rt, Rs, Rt	X	X

Table 2: Instruction possible substitutions - II