



```
addiu $sp, -0x18
sw    $ra, 0x18+var_4($sp)
sw    $a0, 0x18+arg_0($sp)
lui   $t1, 3
jal   sub_2DAB8
lw    $a0, dword_35A6C
lui   $t1, 3
lw    $t7, dword_35A6C
lw    $t6, dword_35A70
subu  $t8, $t6, $t7
addiu $t9, $t8, 4
sltu  $t1, $v0, $t9
beqz  $t1, loc_2DA24
nop
```

Academia vs. Hackers

2nd International Workshop on Secure
Information Systems (SIS'07)

October 15-17, 2007, Wisla, Poland

```
move  $a0, $t7
lw    $a0, dword_35A6C
jal   sub_2DAD4
addiu $a1, $v0, 0x10
beqz  $v0, loc_2DA44
move  $v0, $0
la    $t1, dword_35A70
lw    $t1, dword_35A6C
lw    $t0, 0($t1)
subu  $t2, $t0, $t5
sra   $t3, $t2, 2
sll   $t4, $t3, 2
addu  $t5, $v0, $t4
sw    $t5, 0($t1)
sw    $v0, dword_35A6C
```

Invent & Verify

Agenda

- The common goal
- The conception of responsibilities
- The approaches
- Examples from the field
- Generalizing
- Proposal for future work

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
beqz $1, loc_2DA24
nop
sub 7, 11

```

```

move $a1, $v0
lw $a0, dword_35A70
jal sub_2DAD4
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA44
move $v0, $0
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($1)
sw $v0, dword_35A6C

```

Invent & Verify



Disclaimer

- We are a practitioners company
- We have (legal) hacker backgrounds
- We work with academics
- We highly respect them

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $1, $v0, $t8
beqz $1, loc_2DA24
nop
sub 7, 11
```

```
move $a0, $t7
lw $a0, dword_35A6C
jal sub_2DAD4
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA44
move $v0, $0
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($1)
sw $v0, dword_35A6C
```

Invent & Verify



The common goal

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
$1, 1or_2DA24
sub $t2, $t2, $t8
```

- Information security has three generally accepted goals:
 - Confidentiality**
 - Integrity**
 - Availability**
- Academic research and practitioners' approaches share these goals
 - In a more pragmatic way, we can describe information security as the work towards computer systems we can finally trust.

```
move $a0, dword_35A6C
lw $a0, dword_35A6C
jal sub_2DAB8
addiu $a1, $v0, 1
beqz $v0, 1or_2DA24
move $v0, $0
la $1, dword_35A6C
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t1, $t0
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



Conception of Responsibilities

- Academic research in the field of information security is conceived as responsible for defense mechanisms
 - Cryptography
 - Safe programming languages
 - Safe computing environments
- By identifying the root cause of an issue, a general solution is devised and shown to work

Invent & Verify



```
move $a0, $t7
lw $a1, dword_35A68
jal $a1, $t7
addiu $a1, $t7, 0x10
beqz $v0, loc_2DA44
move $v0, $t7
la $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $t1, 3
la $t2, dword_35A68
lw $t3, dword_35A6C
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $v0, $t8
beqz $t1, loc_2DA24
```

Conception of Responsibilities

- Hackers are the evil
- Hackers devise new ways of breaking into computer systems and make the world a less secure place
- When hackers solve a problem, it's a hack
 - Unreliable
 - Not solving the root cause

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
la $ra, 2DA68
lwi $ra, 2DA6C
lwi $t7, dword_35A6C
lwi $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $v0, $t8
beqz $t1, loc_2DA24
nop
sub $t1, $t1, $t2

```

```

move $a0, $t7
lwi $a0, dword_35A6C
jal sub_2DA40
addiu $a1, $v0
beqz $v0, loc_2DA44
move $v0, $0
la $t1, dword_35A70
lwi $t1, dword_35A6C
lwi $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

Invent & Verify



Conception of Responsibilities

```

addiu $sp, -0x18
sw    $ra, 0x18+var_4($sp)
sw    $a0, 0x18+arg_0($sp)
lui   $t1, 3
la    $a1, dword_35A68
lw    $t2, dword_35A6C
lw    $t7, dword_35A6C
lw    $t6, dword_35A70
subu  $t8, $t6, $t7
addiu $t3, $t6, 4
sllr  $t1, $v0, $t3
sub

```

- Programmers are responsible for writing secure code
 - Validation of input data
 - Failing securely
 - Appropriate and secure application of cryptography
- It's what we could call the "C programming approach": The programmer is responsible for allocating enough memory to store the data he is working with.

- What if that fails?
- Who is to blame for the consequences?

```

move  $a0, $t7
lw    $a0, dword_35A6C
subu  $t1, $a0, $t1
addiu $a1, $v0, 0x10
beqz  $v0, Top_35A74
move  $v0, $0
la    $t1, dword_35A70
lw    $t1, dword_35A6C
lw    $t0, dword_35A70
subu  $t2, $t0, $t1
sra   $t3, $t2, 2
sll  $t4, $t3, 2
addu  $t5, $v0, $t4
sw    $t5, 0($t1)
sw    $v0, dword_35A6C

```

Invent & Verify



Conception of Responsibilities

- The programmer's responsibility has been shown to just not work in practice:

No programmer can ever consider every possible way his code could fail.

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
la $a1, sub_2DAB8
lw $a2, dword_35A6C
lwi $t7, dword_35A6C
lwi $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sltu $t1, $v0, $t8
beqz $t1, loc_2DA24
```

```
move $a0, $t7
lwi $a0, dword_35A6C
jal sub_2DAD4
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA44
move $v0, $0
la $t1, dword_35A70
lwi $t1, dword_35A6C
lwi $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



The approaches: Academia

- You know a lot more about the academic approaches to computer security
 - We are not going to tell you how your world looks like
- From our point of view, they:
 - Aim at general solutions
 - Work excellent in theory
 - Have been instrumental in practice

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
srl $1, 2
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $v0, $t8
beqz $t1, loc_2DA24
```

```
move $a0, $t7
lw $a0, dword_35A6C
jal sub_35A6C
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA44
move $v0, $t0
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



The approaches: Hackers

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_20A68
sw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $t1, $v0, $t8
beqz $t1, loc_20A24
nop
sub $t1, $t1, $t1
```

- Hacker approaches are:
 - Pragmatic
 - Strictly result oriented
 - Rarely aim at a general solution
- Often, our goals are very vague
 - “Gain more privileges than we are supposed to have”
 - “Somehow prevent X from happening”
 - “Because we can?”

```
move $a0, $t7
lw $a0, dword_35A6C
jal sub_20A24
addiu $a1, $v0, 0x10
beqz $v0, loc_20A44
move $v0, $0
la $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t0
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



The approaches: Hackers

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
lwi $t0, dword_35A68
lwi $t7, dword_35A6C
lwi $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sltu $t1, $v0, $t8
beqz $t1, loc_2DA24
nop
sub $t1, $t1, 1

```

- The more aspects of a problem the hacker considers, the more successful he will be.
 - This holds true for defense and offense alike.
- Example questions:
 - “Can I make the defensive/offensive action too costly for the other side?”
 - “Can I trick the other side into incorrectly operating the defense/offense mechanisms used?”
 - “Can I appear to be on the other side?”
 - “Can I use a layer above/below the one attacked/defended?”

```

move $a0, $t7
lwi $a0, dword_35A70
jal sub_2DA04
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA04
move $v0, $0
la $t1, dword_35A70
lwi $t1, dword_35A70
lwi $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

Invent & Verify



Examples from the field

- Cryptography:
Hash performance, fuzzy fingerprints, certificates
- Decompilation:
Boomerang vs. IDA
- Defense mechanisms:
RISE vs. ASLR
- Imperfect solutions:
BinDiff

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $t1, 3
lui $t2, 0x2DAB8
lui $t3, 0x35A6C
lui $t4, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $t0, $t8
beqz $t1, loc_2DA24
pop $t1
sub $t1, $t1, 1
```

```
move $a0, $t7
lw $a1, dword_35A6C
jal $t1, $t2
addiu $a0, $t1, 0x10
beqz $t0, loc_2DA44
move $t0, $t0
la $t1, 0x35A6C
lw $t0, 0($t1)
lw $t0, 0($t1)
subu $t0, $t0, $t0
sra $t0, $t0, 2
sll $t5, $t0, 2
addu $t5, $t0, $t4
sw $t5, 0($t1)
sw $t0, dword_35A6C
```

Invent & Verify



Cryptography: Hash Performance

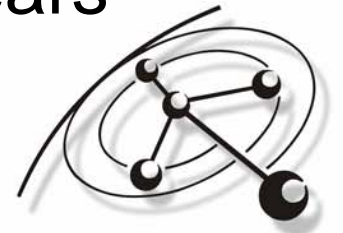
```
addiu $sp, -0x18  
sw $ra, 0x18+var_4($sp)  
sw $a0, 0x18+arg_0($sp)  
lui $1, 3  
jal sub_2DAB8  
lw $a0, dword_35A6C  
lui $1, 3  
lw $t7, dword_35A6C  
lw $t6, dword_35A70  
subu $t8, $t6, $t7  
addiu $t2, $t6, 4  
sltu $1, $v0, $t8  
bgez $1, loc_2DA24
```

- Cryptography is as important to hackers as it is to everyone else
- Hash algorithms are the tool of choice for integrity protection and fixed length data block representations
- We were tasked to select a hash algorithm:

```
move $a0, $t7  
lw $a0, dword_35A6C  
jal sub_2DAB8  
addiu $a1, $v0, 0  
beqz $v0, loc_2DA44  
move $v0, $0  
la $1, word_35A68  
lw $t1, dword_35A6C  
lw $t0, 0($t1)  
subu $t2, $t0, $t1  
sra $t3, $t2, 2  
sll $t4, $t3, 2  
addu $t5, $v0, $t4  
sw $t5, 0($t1)  
sw $v0, dword_35A6C
```

- It had to be secure
- It had to promise security for a few more years
- It was to be used in a network protocol

Invent & Verify



Cryptography: Hash Performance

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
beqz $1, loc_2DA24
nop
sub 7, 11

```

- Selection:
 - MD5 is dead
 - SHA1 appears to die shortly
 - SHA256 – SHA512 are “just” longer versions
- Our favorite cryptographer named **Whirlpool** as the algorithm of choice.

- So we considered it. But...

```

move $a0, $t7
lw $a0, dword_35A6C
jal sub_2DAB8
addiu $a1, $v0, 4
beqz $v0, loc_2DA44
move $v0, $0
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

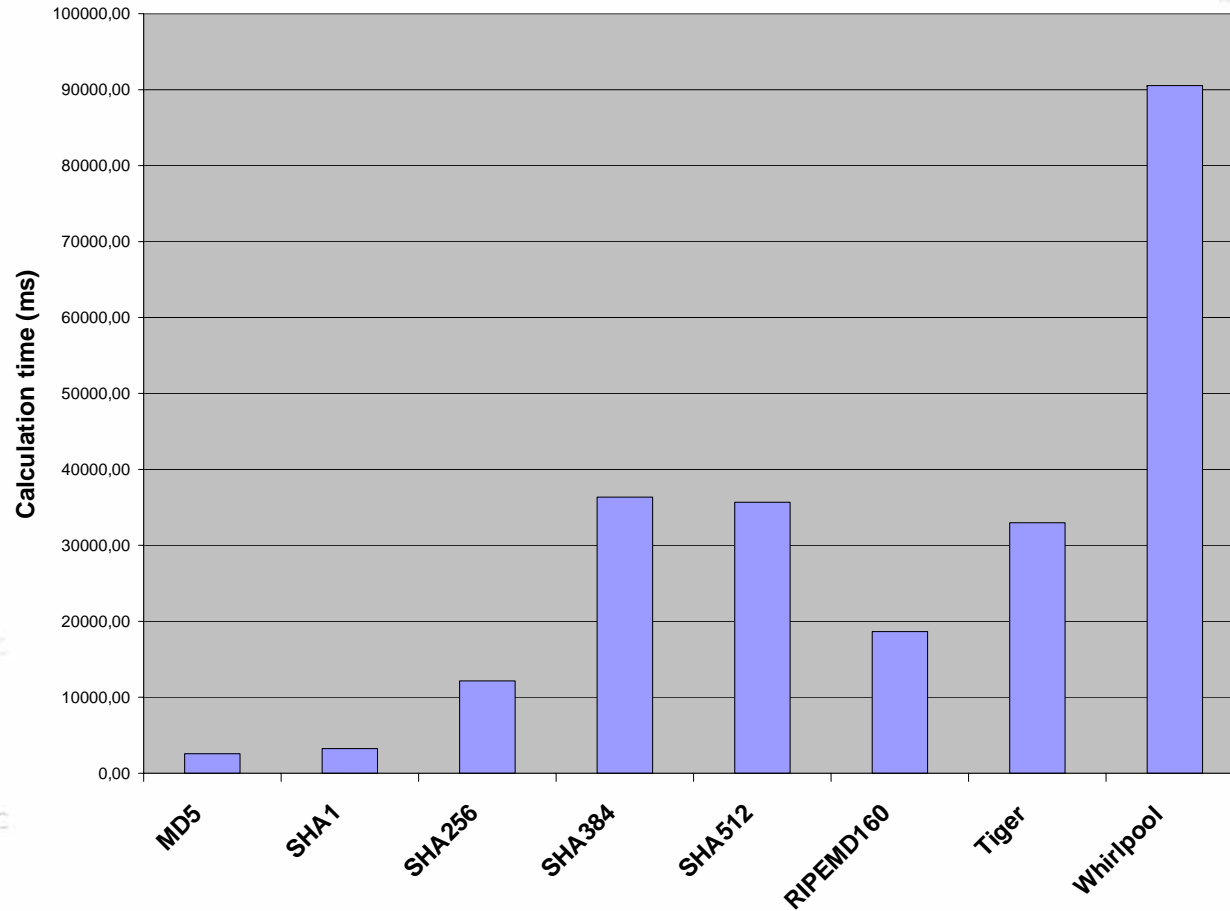
Invent & Verify



Cryptography: Hash Performance

```
addiu $sp, -0x18  
sw $ra, 0x18+var_4($sp)  
sw $a0, 0x18+arg_0($sp)  
lui $1, 3  
jal sub_2DAB8  
lw $a0, dword_35A6C  
lui $1, 3  
lw $t7, dword_35A6C  
lw $t6, dword_35A70  
subu $t8, $t6, $t7  
addiu $t2, $t6, 4  
sllr $1, $v0, $t8  
beqz $1, loc_2DA24  
P  
sub 2...
```

Hash of 639936KB File



```
move $a0, $t7  
lw $a0, dword_35A6C  
jal sub_2DAD4  
addiu $a1, $v0, 0x10  
beqzl $v0, loc_2DA44  
move $v0, $0  
la $1, dword_35A70  
lw $t1, dword_35A6C  
lw $t0, 0($t1)  
subu $t2, $t0, $t1  
sra $t3, $t2, 2  
sll $t4, $t3, 2  
addu $t5, $v0, $t4  
sw $t5, 0($t1)  
sw $v0, dword_35A6C
```

Invent & Verify



Cryptography: Hash Performance

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $1, $v0, $t8
beqz $1, loc_2DA24
nop
sub 7

```

- The insecure algorithms were fast
- The secure algorithms were too slow
- Security (collision resilience) is an important design goal when developing hash algorithms
- Speed is an optional design goal

→ In practice, people will use insecure algorithms because they cannot use the secure ones.

- The collision resilience requirement is overrated while the speed requirement is underrated.

```

move $a1, $v7
lw $a0, 0x18+var_4($sp)
jal sub_2DAB8
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA24
move $v0, 10
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

Invent & Verify



Cryptography: Fuzzy Fingerprints

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
beqz $1, loc_2DA24
sub $t2, $t2, $t8
```

- In cryptography, fingerprints are the most common way to check valid signatures
 - Unfortunately, some times by humans
 - Example: verify the host identity with SSH
- So, instead of breaking the crypto, a hacker [1] came up with a method of generating look-alike fingerprints.

```
move $a0, $t7
lw $a0, dword_35A6C
jal sub_2DAB8
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA44
move $v0, $t0
la $1, dword_35A6C
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

- In crypto, they are not the same
- For the human brain, they may well be

Invent & Verify



Cryptography: Fuzzy Fingerprints

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
beqz $1, loc_2DA24
nop
sub $t2, $t2, $t8

```

```

---[Current State]-----
0d 00h 02m 00s | Total: 2216k hashes | Speed: 18469 hashes/s
-----
Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
Hash Algorithm: Message Digest 5 (MD5)
Digest Size: 16 Bytes / 128 Bits
Message Digest: d1: bc: df: 32: a2: 45: 2e: e0: 96: d6: a1: 7c: f5: b8: 70: 8f
Target Digest: d6: b7: df: 31: aa: 55: d2: 56: 9b: 32: 71: 61: 24: 08: 44: 87
Fuzzy Quality: 47.570274%

```

```

move $v0, $0
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($1)
sw $v0, dword_35A6C

```

Invent & Verify



Cryptography: Fuzzy Fingerprints

```
addiu $sp, -0x18  
sw $ra, 0x18+var_4($sp)  
sw $a0, 0x18+arg_0($sp)  
lui $1, 3  
jal sub_2DAB8  
lw $a0, dword_35A6C  
lui $1, 3  
lw $t7, dword_35A6C  
lw $t6, dword_35A70  
subu $t8, $t6, $t7  
addiu $t2, $t6, 4  
sltu $1, $v0, $t8  
beqz $1, loc_2DA24  
sub 7...
```

- An example of attacking a different layer:
 - Ignore the crypto
 - Fool the crypto user

- Side note:

A lot of academic work went into this attack for generating the fuzzy fingerprints

```
move $a0, $v0  
lw $a0, dword_35A6C  
jal sub_2DAB8  
addiu $a0, $a0, 4  
beqz $v0, loc_2DA44  
move $v0, $0  
la $1, dword_35A70  
lw $t1, dword_35A6C  
lw $t0, 0($t1)  
subu $t2, $t0, $t1  
sra $t3, $t2, 2  
sll $t4, $t3, 2  
addu $t5, $v0, $t4  
sw $t5, 0($t1)  
sw $v0, dword_35A6C
```

Invent & Verify



Cryptography: Certificates

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
ror $1, $v0, $t8
sub $t2, $t2, $t8

```

- A lot of research went into authentication and authorization using certificates and PKI
 - They all depend on the secrecy of the secret key
 - Therefore, they assume that the secret key is rarely lost to a third party
- Current certificate / PKI systems are unmanageable, since in reality, people are loosing their secret keys to attackers ***all the time***

→ Spend more time researching fast and scalable revocation ☺

```

move $a0, $t7
lw $a1, $a0
jal sub_2DAB8
addiu $a1, $v0, 0x10
beqz $v0, $t2
move $v0, $t2
la $1, dword_35A70
lw $t1, dword_35A70
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

Invent & Verify



Program code recovery

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $t1, 3
sll $t1, $t1, 2
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sll $t1, $t1, 4
beqz $t1, $t2, 2
```

- In reality, there are many legitimate reasons to recover the code of a program from its binary representation:
 - The source code might be lost
 - The compiler might no longer exist
 - The platform might no longer be available

But the most important one is:

- You simply don't have the source

```
move $a0, $t7
lw $a0, 0x18+var_4($sp)
jal sub_35A44
addiu $a1, $v0, 0x10
beqz $v0, $a1, 25A44
move $v0, $a0
la $t1, dword_35A6C
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



Boomerang [2]

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $1, $v0, $t8
beqz $1, loc_2DA24
```

- Since 2002, the Boomerang decompiler is a successful project to recover source code from binary code
 - Developed by a team at the University of Queensland, Australia
 - Widely recognized for significant advancements in the field of decompilation
 - Team: 6 developers

```
move $a0, $t0
lw $a0, dword_35A6C
jal sub_2DAD4
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA24
move $v0, $0
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```



IDA Pro [3]

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
beqz $1, loc_2DA24
nop
sub 7, 11
```

- A disassembler, not a decompiler
- The tool of choice for reverse engineering
 - Interactive: accepts the fact that experience is the best pattern recognition
 - Imperfect: Fails often, but allows the user to correct the failure
 - Helps the reverse engineer to recover the source code manually
 - Team: 1 developer

```
move $a0, $t2
lw $a0, dword_35A6C
jal sub_2DAB8
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA24
move $v0, $0
la $1, dword_35A70
lw $t1, dword_35A6C
lw $t0, $t1
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



Enter: OOP

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $t1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $t1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $v0, $t8
beqz $t1, loc_2DA24
nop
sub $t1, $t1, 1

```

- Both tools (should we say approaches)
 - Have been around for a while
 - Have been used and adapted
 - Fail with code generated from C++
- For people in the field, this means:
 - Adjusting IDA took a few days (plugin)
 - Adjusting Boomerang would take ???

➔ While the decompiler attempts to tackle the root cause of the problem (lost source), reality gives the advantage to the quick-and-dirty solution (the disassembler).

```

move $v0, $v0
lw $t0, 0($v0)
jal sub_2DAD4
addiu $a0, $v0, 4
beqz $t0, loc_2DA24
move $v0, $v0
la $t1, 0($v0)
lw $t1, dword_35A6C
lw $t1, dword_35A6C
subu $t2, $t1, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

Invent & Verify



Exploitation Prevention: RISE [4]

- “Randomized Instruction Set Emulation”
- Encrypts binary code at load time
- Decrypts binary code during execution (instruction fetch) in an emulator environment
 - Since attacker provided code is not encrypted “correctly”, it will produce gibberish when decrypted and fail
 - The process will crash, or, in supposedly rare circumstances, loop infinitely
 - Execution of attacker code is prevented

Invent & Verify



```
move $a0, $t7
lw $a0, dword_35A6C
jal sub_27A4A
addiu $a1, $v0, 4
beqz $v0, loc_27A44
move $v0, $0
la $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, 2
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

```
addiu $ep, -0x18
sw $ra, 0x18+var_4($ep)
sw $a0, 0x18+arg_0($ep)
lwf $t1, 3($t0)
subu $t1, $t1, 3
lw $a0, dword_35A6C
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sll $t1, $v0, $t8
lwf $t1, loc_27A24
subu $t1, $t1, 3
```

Exploitation Prevention: RISE

- Obvious performance impact
- Binary programs can no longer be debugged if anything fails
- And by the way:
Killing sparrows with nuclear warheads, including AWACS support, was never a wise move.

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
lwi $t0, dword_35A6C
lwi $t7, dword_35A6C
lwi $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $t0, $t8
beqz $t1, loc_2DA24
nop
sub $t1, $t1, $t1

```

```

move $a0, $t7
lwi $a0, dword_35A6C
jal sub_35A6C
addiu $a0, $sp, 4
beqz $v0, loc_2DA44
move $v0, $0
la $t1, dword_35A70
lwi $t1, dword_35A6C
lwi $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

Invent & Verify



Exploitation Prevention: ASLR [5]

- Executing attacker provided code needs an important piece of information: the **address**
- Address Space Layout Randomization just loads the program and its segments into different virtual memory regions every time
 - Performance impact: none (regular operation)
 - Result: better than RISE
- Concept developed by hackers who did not want to get compromised by their peers
 - Linux GRSEC/PAX patch
 - OpenBSD
 - Now also in Windows Vista

```

move $a1, 0($a0)
lw $a1, 0($a1)
jal sub_20A24
addiu $a1, $v0, 1
beqz $v0, 1($v0)
move $v0, 0
la $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, 1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C

```

```

addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
lwi $a1, 2
lw $a0, dword_35A6C
lwi $t7, dword_35A6C
lwi $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sll $t1, $v0, $t2
beqz $t1, 1($v0)
nop
sub $t1, $t1, 1

```

Invent & Verify



RISE vs. ASLR

- The inventors of RISE only considered successful exploits in their research
- The inventors of ASLR considered why exploits are often unsuccessful and how to force this situation

```

addiu $sp, -0x18
sw    $ra, 0x18+var_4($sp)
sw    $a0, 0x18+arg_0($sp)
lui   $t1, 3
jal   sub_2DAB8
lw    $a0, dword_35A6C
lui   $t1, 3
lw    $t7, dword_35A6C
lw    $t6, dword_35A70
subu  $t8, $t6, $t7
addiu $t2, $t6, 4
sllw  $t1, $v0, $t8
bgez  $t1, loc_2DA24
nop
sub   $t1, $t1, $t1

```

```

move  $a0, $t7
lw    $a0, dword_35A6C
jal   sub_2DAD4
addiu $a1, $v0, 0x10
beqz  $v0, loc_2DA44
move  $v0, $0
la    $t1, dword_35A70
lw    $t1, dword_35A6C
lw    $t0, 0($t1)
subu  $t2, $t0, $t1
sra   $t3, $t2, 2
sll   $t4, $t3, 2
addu  $t5, $v0, $t4
sw    $t5, 0($t1)
sw    $v0, dword_35A6C

```

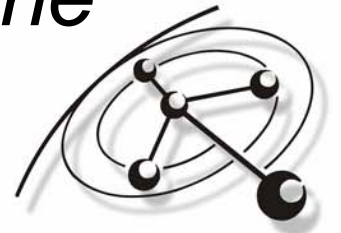
Invent & Verify



Imperfect Solutions: BinDiff

- Comparing two binary programs to determine if they share functionality is hard
 - From an academic point of view, it's impossible
- Comparing two binary programs that are known to be very much alike has advantages for a hacker
 - Think before and after a security patch
- So, someone [6] came up with a structural comparison method that works *most of the time*.

Invent & Verify



```
move $a0, $t7
lw $a0, dword_35A6C
jal sub_35A64
addiu $a0, 0x18
beqz $v0, $PA
move $v0, $0
la $t0, 0
lw $t0, 0($t0)
lw $t0, 0($t0)
subu $t0, $t0, $t0
sra $t0, $t0, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t0)
sw $v0, dword_35A6C
```

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwf $t1, 3
jcf $t1, 2
lwf $a0, dword_35A6C
lwf $t1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sltu $t1, $t0, $t8
```

Imperfect Solutions: BinDiff

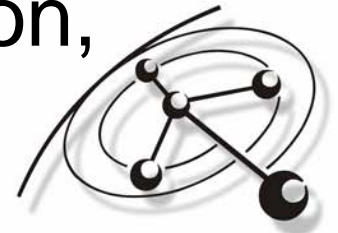
```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
lwi $t2, 2
lwi $a0, dword_35A6C
lwi $t1, 3
lwi $t2, 3
lwi $t7, dword_35A6C
lwi $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $t1, $t0, $t8
beqz $t1, loc_2DA24
nop
sub $t1, $t1, 1
```

- Turns out:
 - The method works well with any two binaries!
 - Even when both are for different CPU platforms.
 - Even when only a few functions are common
- The method also allows to give similarity measures for any two binaries.

```
move $a0, $t7
lwi $a0, dword_35A6C
jal sub_2DAD4
addiu $t1, $t0, 10
beqz $t1, loc_2DA24
move $v0, 10
la $t1, dword_35A70
lwi $t2, $t1, 0($t1)
lwi $t3, $t1, 4($t1)
subu $t4, $t3, $t2
sra $t1, $t4, 2
sll $t1, $t1, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

→ Although the problem is still provably unsolvable, for any practical consideration, it is solved.

Invent & Verify



Generalizing

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $1, $t0, $t8
jal sub_2DA24
```

- From our point of view, the scientific way of defining requirements is too strict for real world use.
 - It seems to be common to consider the initial goal of the research a hard requirement while neglecting the environmental aspects
 - What is “given” (the premise) does not always hold true
 - The better approach might be to take several possible combinations of “given” as base for the research

```
move $a0, $t0
lw $a0, dword_35A70
jal sub_2DAD4
addiu $a1, $v0, 0x10
beqz $v0, 0x0
move $v0, 10
la $1, dword_35A70
lw $t1, dword_35A70
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 1
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



Generalizing

- Assume that things go wrong in the worst possible way
- Never start reasoning with: “Nobody would ever...”
 - Someone will
- Usability in production environments should play a bigger role in the development of defense technologies

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lui $1, 3
jal sub_2DAB8
lw $a0, dword_35A6C
lui $1, 3
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllr $1, $v0, $t8
beqz $1, loc_2DA24
```

```
move $a0, $v0
lw $a0, dword_35A6C
jal sub_2DAB8
addiu $a0, $v0, 10
beqz $v0, loc_2DA44
move $v0, $a0
la $t1, loc_2DA44
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

Invent & Verify



Proposal for future work

- The discussion has been too arrogant from both sides in the past
 - Academia is per se right
 - Hackers are per se successful
 - Both is wrong
- At the end of the day, we are all in the same boat here.

```
move $a0, $t7
lw $a1, dword_35A70
jal $a0
addiu $a1, $v0, 0x10
beqz $v0, loc_2DA44
move $v0, $0
la $t1, dword_35A70
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
lwi $t2, dword_35A68
lw $t3, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $t1, $v0, $t8
beqz $t1, loc_2DA24
```

Invent & Verify



Proposal for future work

- Most serious hackers will be honored and delighted to review research concepts (before they are done)
 - Yes, most of them do that for free, even if they are professionals
- Most would also be delighted to show their ideas to academic researchers for feedback and discussions
 - If they don't find their concepts on ACM with someone else's name on it

Invent & Verify



```
move $a0, $v0
lw $a0, dword_35A44
jal $a0
addiu $a0, $v0, -0x10
beqz $v0, loc_2DA44
move $v0, $0
la $t1, dword_35A6C
lw $t1, dword_35A6C
lw $t0, 0($t1)
subu $t2, $t0, $t1
sra $t3, $t2, 2
sll $t4, $t3, 2
addu $t5, $v0, $t4
sw $t5, 0($t1)
sw $v0, dword_35A6C
```

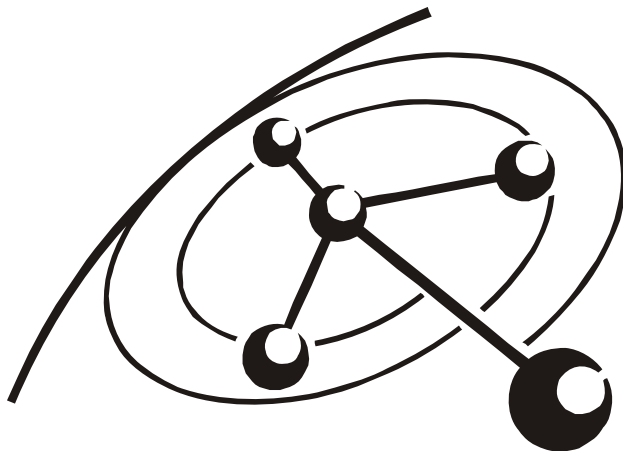
```
addiu $sp, -0x18
sw $ra, 0x18+var_4($sp)
sw $a0, 0x18+arg_0($sp)
lwi $t1, 3
fcti $t1, 2DAB8
lw $t2, dword_35A6C
lw $t7, dword_35A6C
lw $t6, dword_35A70
subu $t8, $t6, $t7
addiu $t2, $t6, 4
sllw $t3, $v0, $t8
lwi $t1, 1, loc_2DA24
subu $t1, $t1, $t2
```


Recurity Labs

```

addiu $sp, -0x18
sw    $ra, 0x18+var_4($sp)
sw    $a0, 0x18+arg_0($sp)
lui   $1, 3
jal   sub_2DAB8
lw    $a0, dword_35A6C
lui   $1, 3
lw    $t7, dword_35A6C
lw    $t6, dword_35A70
subu  $t8, $t6, $t7
addiu $t2, $t6, 4
sllw  $1, $t6, $t8
beqz  $1, loc_2DA24

```



Recurity Labs

Jörn Bratzke
Vulnerability Researcher

Phone: +49 30 69539993-0
 FAX: +49 30 69539993-8
 E-Mail: joern@recurity-labs.com

Recurity Labs GmbH, Berlin, Germany
<http://www.recurity-labs.com/>

```

move  $a0, $a0
lw    $a0, $a0
jal   sub_2DAB8
addiu $a1, $a1
beqz  $v0, $v0
move  $v0, $v0
la    $1, 0
lw    $t1, dword_35A6C
lw    $t0, 0($t1)
subu  $t2, $t0, $t1
sra   $t3, $t2, 2
sll  $t4, $t3, 2
addu  $t5, $v0, $t4
sw    $t5, 0($t1)
sw    $v0, dword_35A6C

```

Invent & Verify

