ROPER:
A Genetic ROP-Chain Compiler Targetting Embedded Devices

Olivia Lucca Fraser

NIMS Lab, Dalhousie University

June - August, 2016

0.0 There are Approximately 7 ARM Processors on the Market for Every Living Human

ARM is now the de facto standard architecture for embedded and mobile devices, including phones, routers, pacemakers, surveillance cameras, printers, tablets, cars, etc.

0.1 Return Oriented Programming (ROP-chain attacks)

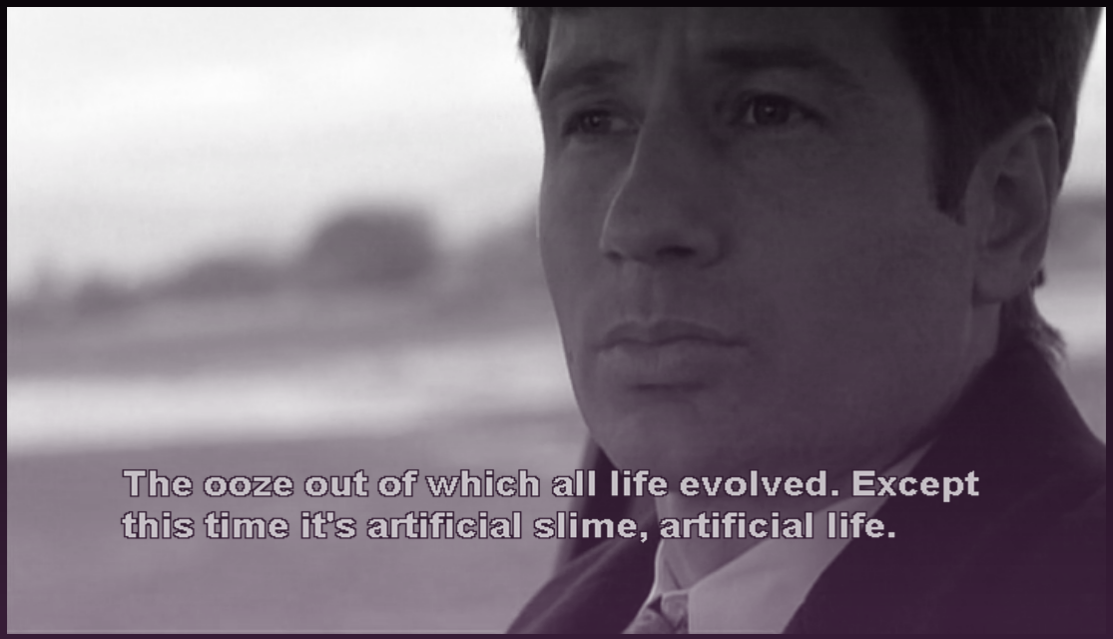
```
e50b3008        str     r3, [fp, #-8]
e3a0000c        mov     r0, #12
ebffff75        bl      8480 <_init+0x48>
e1a03000        mov     r3, r0
e50b3008        str     r3, [fp, #-8]
e3a0000a        mov     r0, #10
ebffff71        bl      8480 <_init+0x48>
e1a03000        mov     r3, r0
e1a02003        mov     r2, r3
e51b3008        ldr     r3, [fp, #-8]
e5832000        str     r2, [r3]
e51b3008        ldr     r3, [fp, #-8]
e5933000        ldr     r3, [r3]
e1a00003        mov     r0, r3
e3a01000        mov     r1, #0
e3a0200a        mov     r2, #10
ebffff76        bl      84bc <_init+0x84>
e3a0000a        mov     r0, #10
ebffff65        bl      8480 <_init+0x48>
e1a03000        mov     r3, r0
e1a02003        mov     r2, r3
e51b3008        ldr     r3, [fp, #-8]
e5832004        str     r2, [r3, #4]
e51b3008        ldr     r3, [fp, #-8]
e5933004        ldr     r3, [r3, #4]
e1a00003        mov     r0, r3
e3a01000        mov     r1, #0
e3a0200a        mov     r2, #10
ebffff6a        bl      84bc <_init+0x84>
e3a0000c        mov     r0, #12
ebffff59        bl      8480 <_init+0x48>
e1a03000        mov     r3, r0
e1a02003        mov     r2, r3
e51b3008        ldr     r3, [fp, #-8]
```

0.2 Return-Oriented Programming, cont.

- If we can't write shellcode to executable memory, then we'll just have to make use of memory that has *already* been marked executable.
- This means salvaging whatever 'gadgets' we can find there and using them to build our payload.

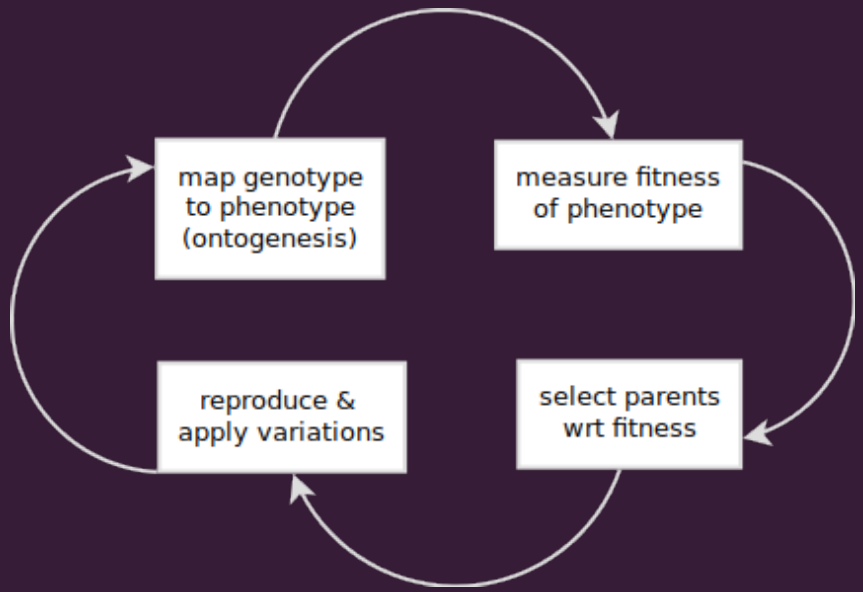
- any chunk of code sitting in executable memory can act as a gadget so long as we can regain control of programme after it executed
- typically this means that gadgets end with a "return" instruction
- hence the term "Return Oriented Programming"
- gadgets can be chained together to form arbitrarily complex programmes
- typically they are implemented as stacks of addresses, each pointing to a gadget that ends by hopping to the next address in the stack
- building these chains manually is difficult

0.3 Genetic Algorithms: Natural Selection in Code



Natural selection can be implemented in code.
We just need the space of possible solutions to exhibit:

- **variation** (sexual recombination or mutation, e.g.)
- **inheritance** (trivial, since we can copy code freely)
- **selection** (with respect to a fitness function)



0.4 Genetic Algorithms in Offensive Security

There has been surprisingly little usage of evolutionary methods in offensive security, as far as I'm aware. Some notable exceptions include:

- At DEFCON 21 (2013), Soen Vanned presented a tool that used GA to fuzz web forms over HTTP/HTTPS and test for vulnerabilities to SQL and shell command injection.
- Genetic algorithms have also been put to good use in code fuzzing (auditing a code base for bugs and vulnerabilities). The most prominent example is American Fuzzy Lop, developed by Michal Zalewski (AKA lcamtuf).
- 2006-2009 in the **NIMS** lab: Gunes Kayacik conducted a series of experiments, using genetic algorithms to develop stack-overflow shellcode attacks against Unix utilities, aiming to evade adaptive intrusion detection systems by training the attacks to mimic 'normal' behaviour.

0.5 Question

Can we utilize genetic algorithms to *evolve* ROP-chain payloads out of the “primordial ooze” of executable memory?

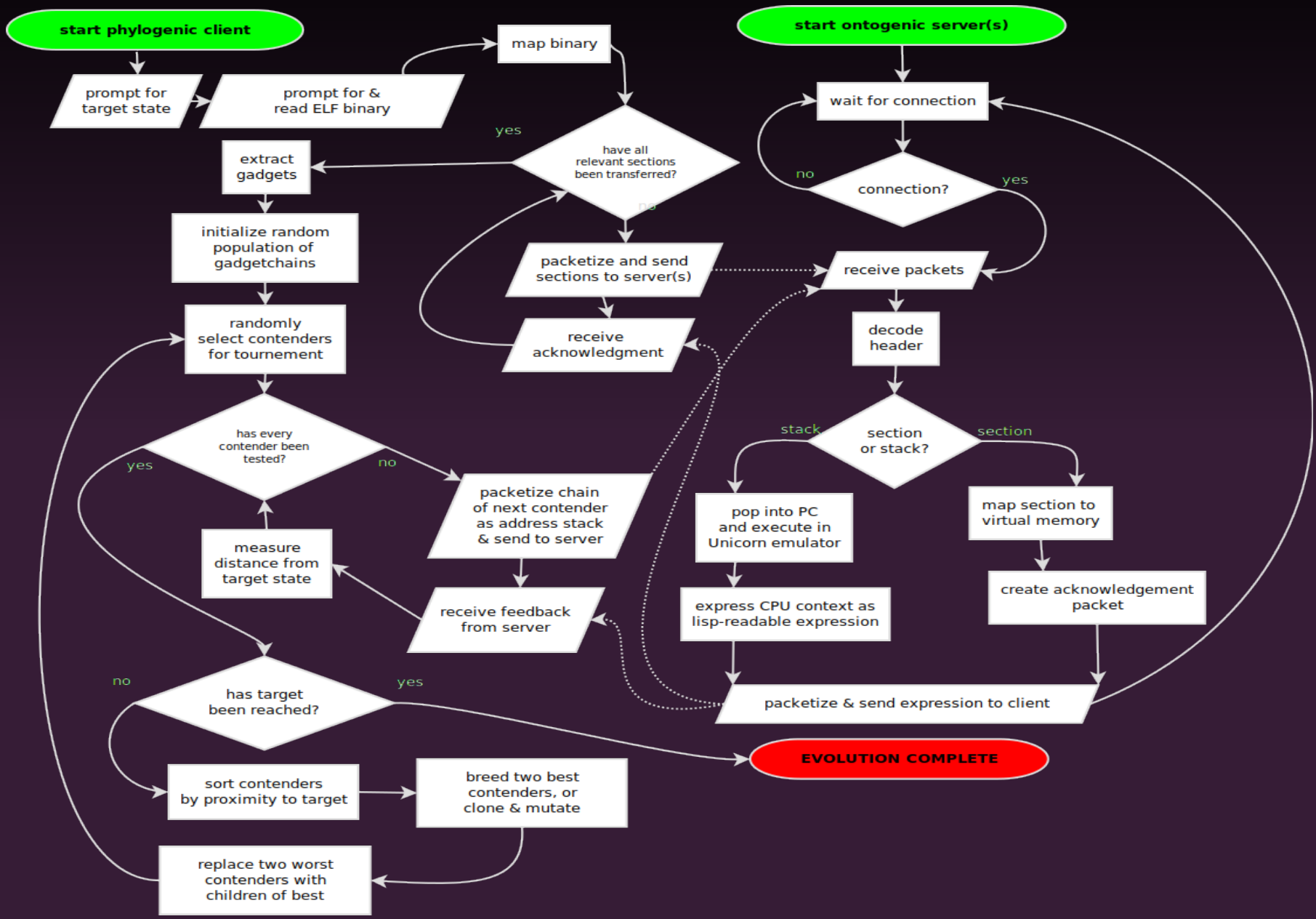
0.6 Introducing ROPER: A genetic engine for the evolution of ROP-chain payloads, targeting the ARM processor

Genotype: a stack of addresses that can be derefrenced into ROP-gadgets.

Phenotype: the behaviour of the CPU when this stack is executed (when it is popped into the program counter register).

- ROPER analyses target ELF binary file, and extracts ROP gadgets;
- these act as a **“gene pool”** out of which a random population of ROP-chains is initialized;
- each generation, a sample of chains are evaluated in a **virtual environment**,
- we isolate the **“fittest”** chains, the ones that come closest to bringing about the desired CPU context,
- and encourage them to **breed** and **mutate**,
- until they **converge** on a chain that accomplishes the task in question.

0.7 How ROPER works



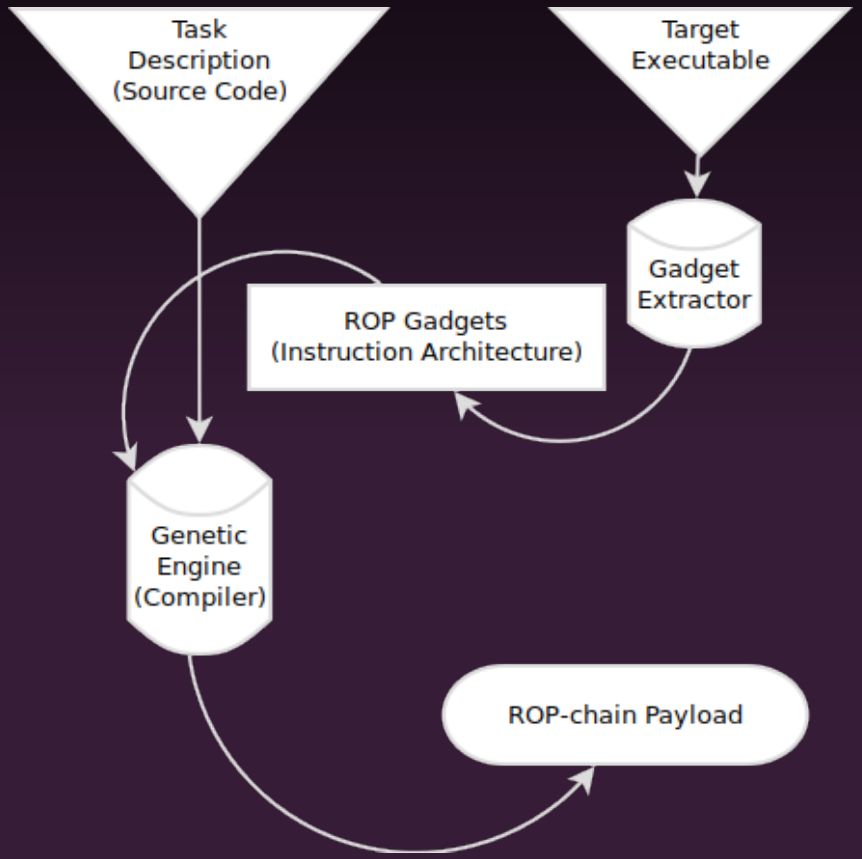
0.8 TODO

- ROPER is being designed so as to be easily extensible to other architectures besides 32-bit ARM (x86, x86_64, MIPS, ARM-64, etc.), so this can and should be actualized;
- the tool could be seen as something of a ‘compiler’ for a simple, declarative scripting language. At present this language consists only in simple register patterns, but it could easily be extended into something more ‘high-level’ and useful.
- the structure of the tool lends itself well to parallelization and distributed computing, which would increase its efficiency by a few orders of magnitude, if properly implemented.

Progress Report # 1

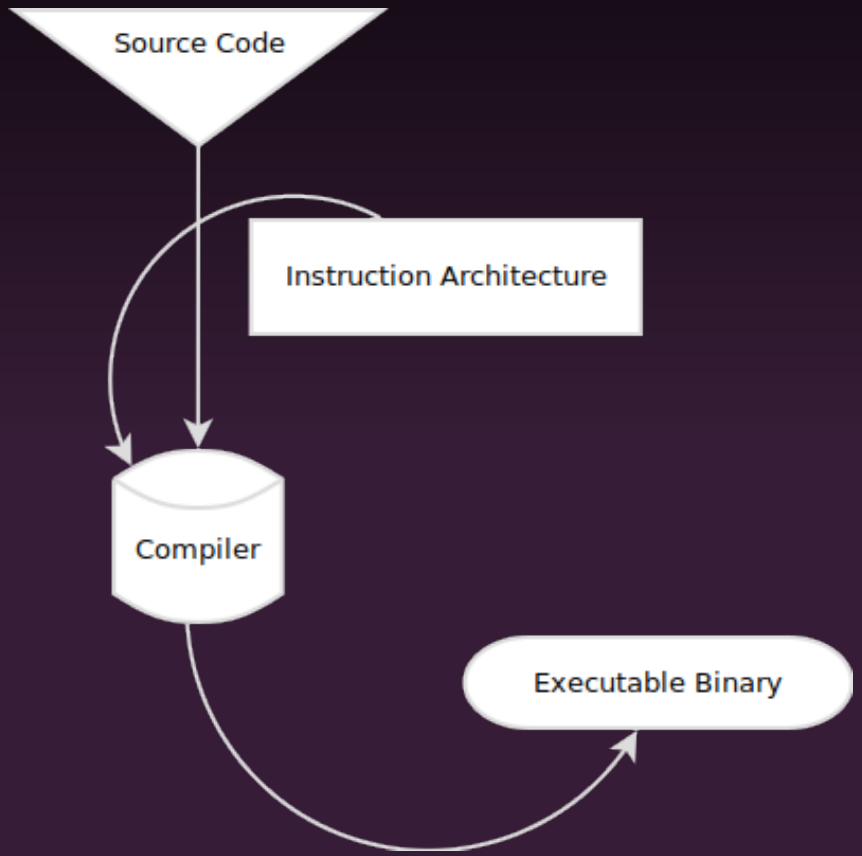
July, 2016

1.0: Conceptual Map



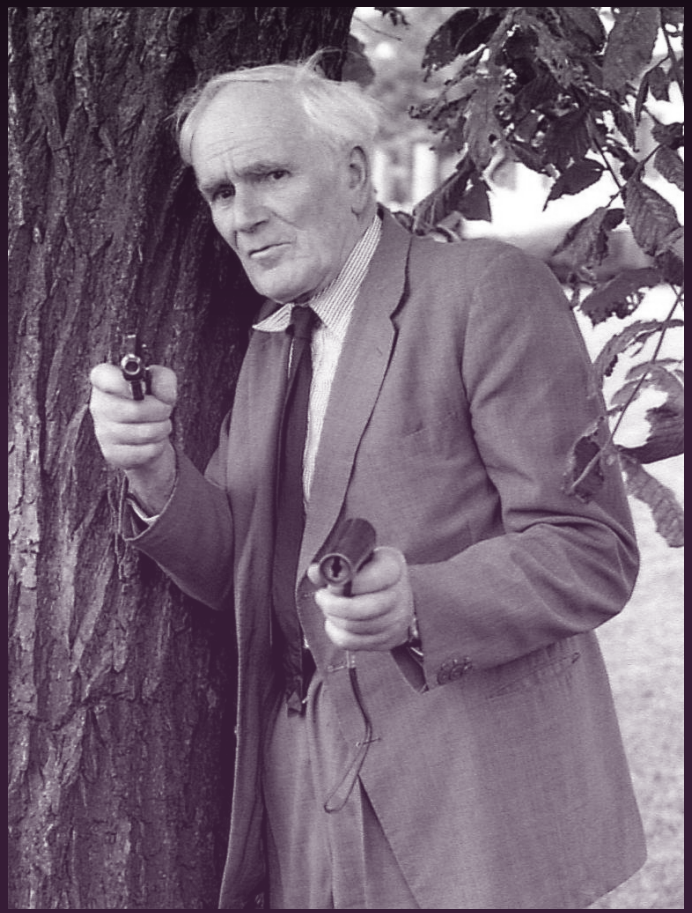
- At a certain level of abstraction, ROPER is just a *compiler*.
- It compiles a task description to a ROP-chain,
- using the “gadgets” extracted from the target as its instruction set
- and using a genetic algorithm as its instruction selection algorithm

1.1 Compiler Basics



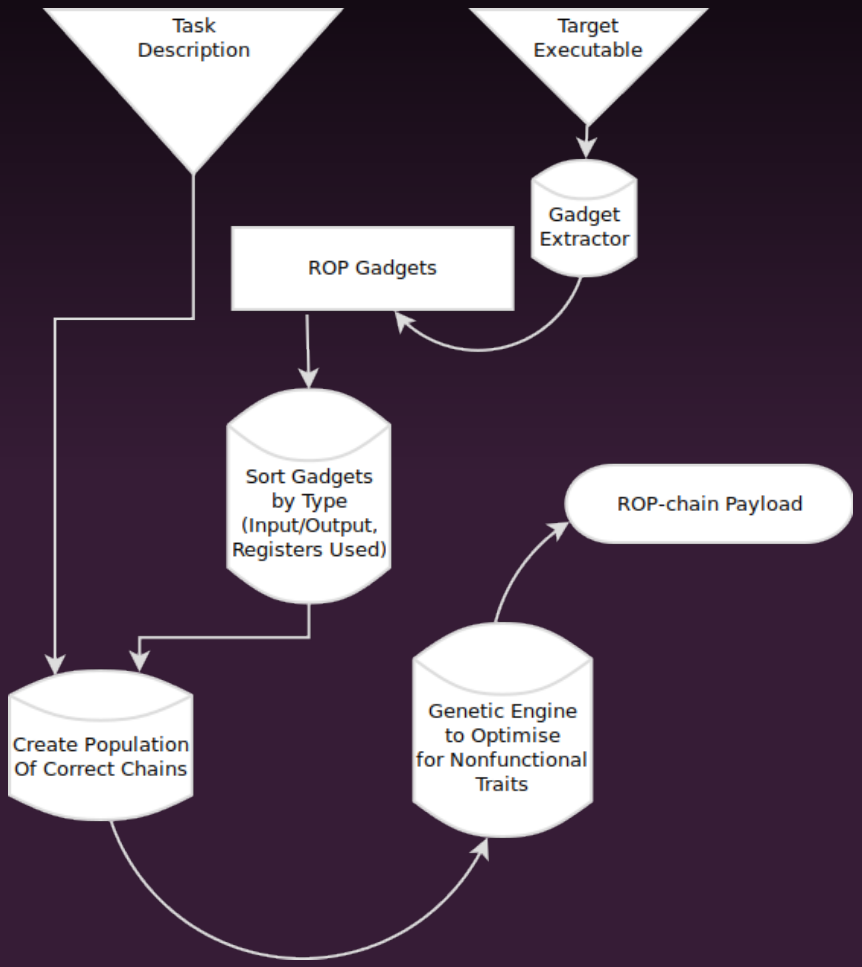
- a compiler translates source code to machine code
- the machine code consists of *instructions* that map onto the primitive actions of the CPU
- but for ROPER, the primitive “instructions” will be the gadgets extracted from the victim binary

1.2 Current State of the Art: Q



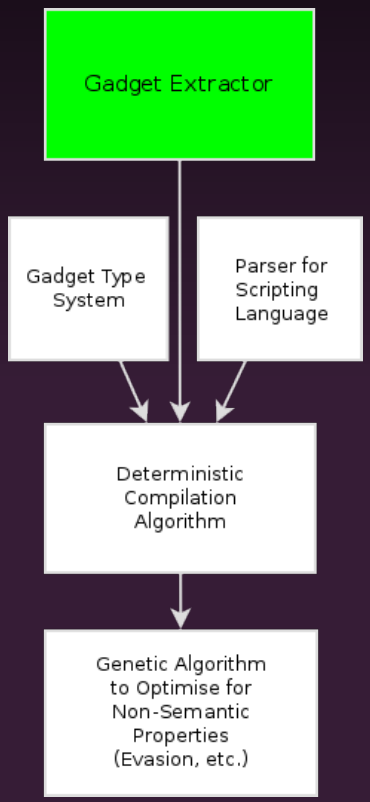
- ”Q: Exploit Hardening Made Easy”, Usenix Security Symposium, 2011: Schwartz, Avgerinos, & Brumley
- **Q** is a fully deterministic compiler, not driven by machine learning, that uses classical algorithms to compile user-written scripts into ROP-chains, using a given binary
- quite efficient, produces payloads from binaries $>=$ 20KB (tested on common Unix utilities)
- targets the x86 architecture, while our focus is the ARM and other embedded/mobile RISC architectures
- neither binary nor source seem to be publicly available
- we can build on their design, & incorporate genetic methods to optimize for **stealth**

1.3 Tactical Deployment of Genetic Methods



- use efficient deterministic methods wherever applicable
- we can deterministically compile semantically correct ROP-chains
- **Q** is proof that this can be done
- we will first design a skeletal, deterministic ROP compiler, & then exploit genetic methods to optimise for non-semantic properties, like stealth
- Kayacik's work (in our lab) showed that this, too, is feasible

1.4 Roadmap

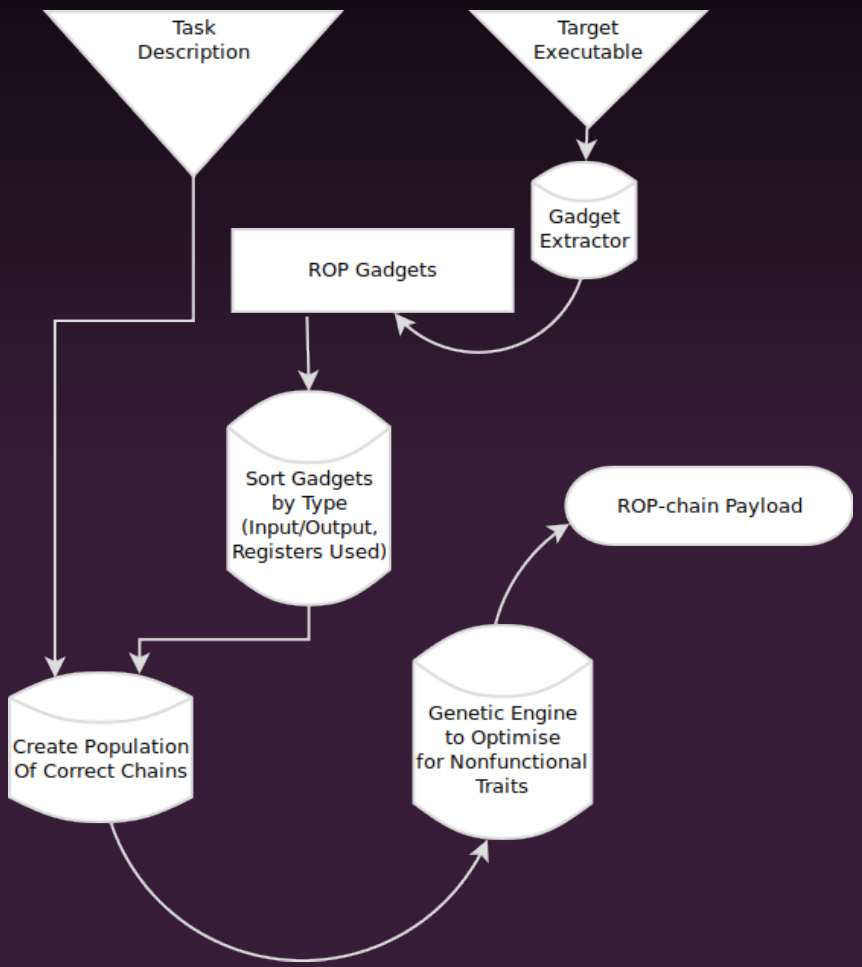


- I have ported the gadget-extraction algorithm, which I had earlier written in Lisp, to Haskell, for better integration with the rest of the compiler & emulator
- Now working on a type system for the gadgets – sorting them by input/output, and by registers used, so that they can be easily composed into chains by the deterministic compiling algorithms
- Planning the genetic components, so as to best take advantage of what they have to offer over and above deterministic compilation algorithms
- Planning a simple, scheme-like scripting language and parser

Progress Report # 2

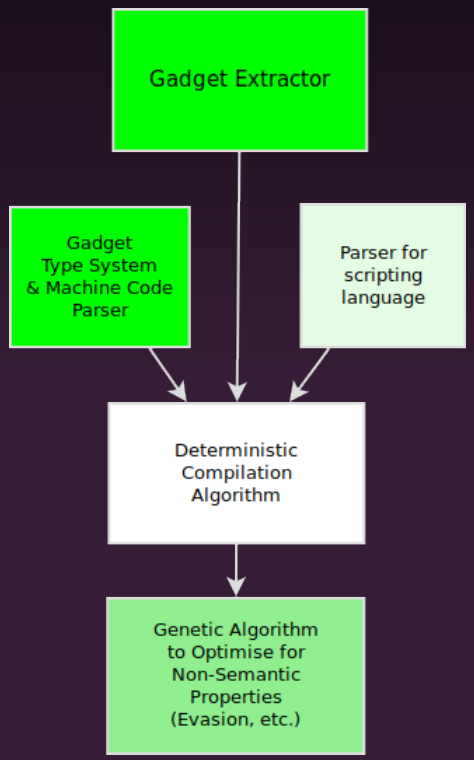
August, 2016

2.0 Conceptual Map (Recap)



- ROPER is a tool to automatically generate ROP-chains – exploit payloads that cannibalize a target process's own executable code segment rather than introducing code of their own
- it hybridizes deterministic compiling techniques with stochastic genetic algorithms
- the compiler generates an initial population of semantically correct or approximate solutions
- genetic algorithm optimises for both semantic correctness and non-semantic properties (stealth, brevity, obfuscation, etc.)

2.1 Roadmap



- I am developing a parser that reads compiled ARM machine code, and re-compiles it into series of Haskell functions and data structures.
- some of this code will be reused in the parser needed to compile the ROPER scripting language into an initial population and directions for the genetic algorithm
- I've also been experimenting with and prototyping the genetic component, independently of the rest, in LISP and C (with a purely random initial population)

2.2 Parsing and Compiling Machine Code into Haskell Functions

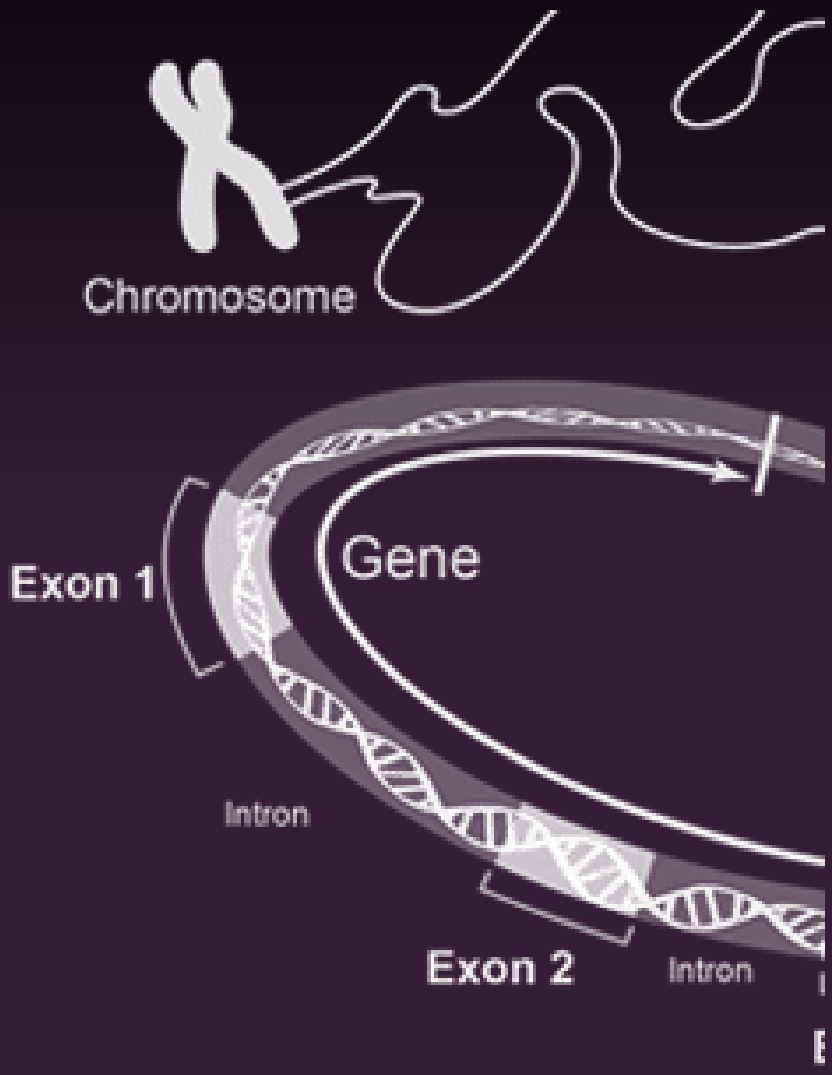
- ROPER finds its raw materials (its ‘instruction architecture’) by taking apart a target binary programme

- it order to process these materials intelligently, it needs to convert them into more tractable structures

- ROPER partially decompiles, or *recompiles*, the binary into a sequence of haskell functions and structures

- these can be analyzed and manipulated algebraically, so that they approximate or satisfy the objective given by the user, before being passed to the genetic algorithm

2.3 Application of the Type System: Detection of Syntactic Introns



'Introns' are pieces of code ('genes') in a sequence that have no semantic or functional effect on its output. Our type system will let us detect and manipulate them.

- The concept has a biological analogy in 'junk DNA'.
- Introns confer interesting and useful non-semantic features on the evolving algorithms:
  - » robustness to mutation – if a mutation affects intron segments, it will not semantically alter the output
  - » punctuated equilibrium – changes can accumulate for some time before being suddenly 'switched on'
  - » potential for obfuscation

2.4 Prototyping the Genetic Algorithms in Lisp

While developing the initial population compiler in Haskell, I experimented with a quick and dirty implementation of a few genetic algorithms in Lisp, and an ARM emulator server written in C.

The initial population, here, was *entirely randomized*, and the task was relatively simple:

- *bring the CPU context to a desired state, specified by a register pattern*

E.g. `#(0x01 _ 0x04 _ _ _ 0x05 )` means: *set R0 to 0x01, R2 to 0x03, and R6 to 0x05*

Candidate fitness functions:

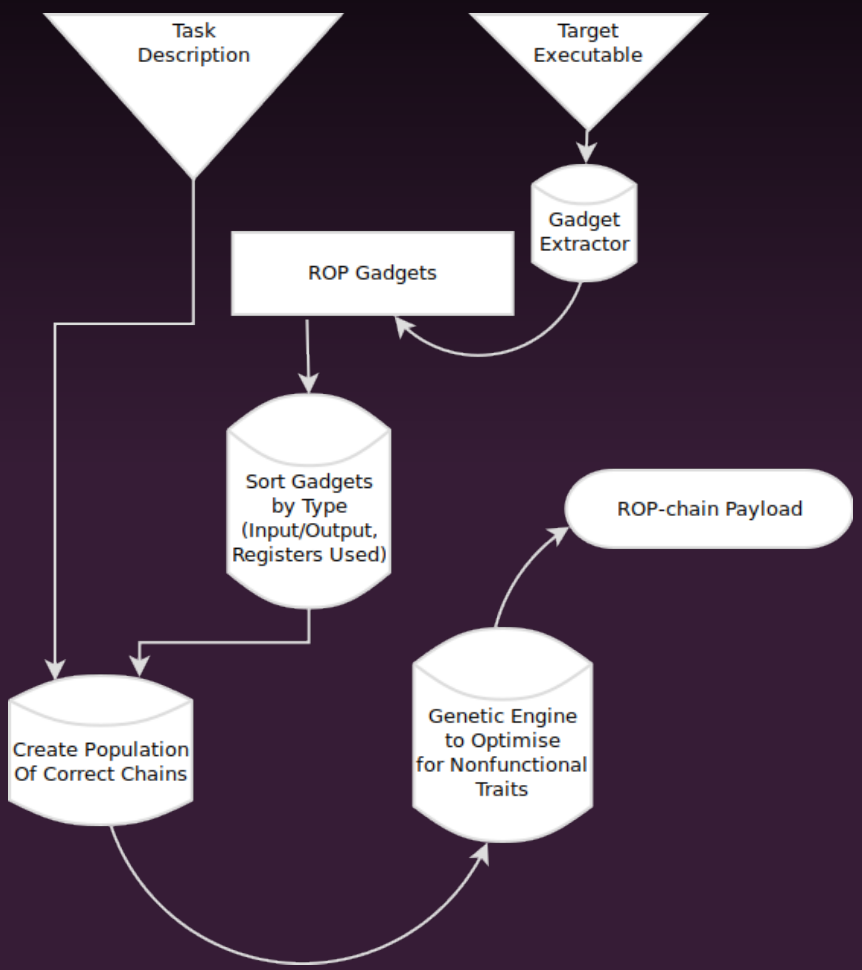
- An adaptation of Lexicase Selection: for each relevant register *R*, in random order, assess each creature in the population, and discard any that fail to correctly set *R*. When two remain, mate them, and replace the first culled with their child.
- A Euclidean distance function: treat the desired register pattern as an *n*-dimensional hyperplane in 15-dimensional space (for 15 registers). A creature's fitness is the distance between the register state it achieves and the target hyperplane.

The second met with greater success: after about 1000 generations, a 32 gadget-long chain emerged that satisfied the test pattern.

Progress Report # 3

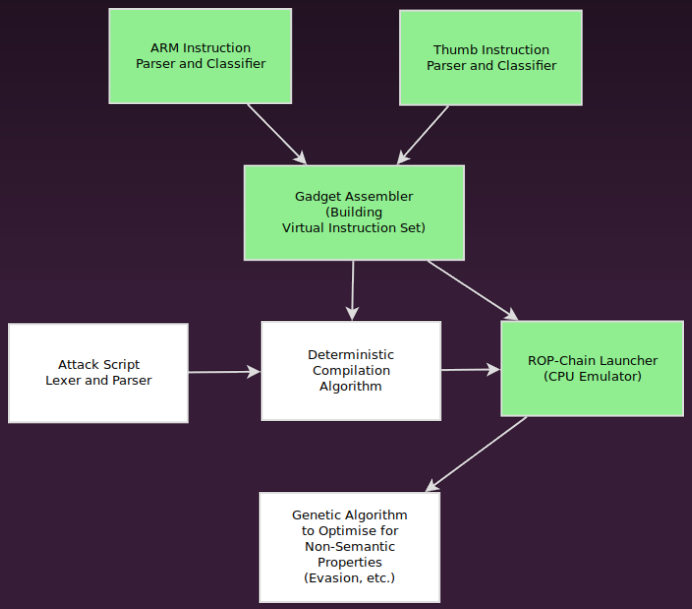
October, 2016

3.0 Conceptual Map (Recap)



- ROPER is a tool to automatically generate ROP-chains – exploit payloads that cannibalize a target process's own executable code segment rather than introducing code of their own
- it hybridizes deterministic compiling techniques with stochastic genetic algorithms
- the compiler generates an initial population of semantically correct or approximate solutions
- genetic algorithm optimises for both semantic correctness and non-semantic properties (stealth, brevity, obfuscation, etc.)

3.1 Roadmap



- Much of the work in the past two months has involved porting the code to Haskell (from Lisp and C) and optimizing its performance.
- The parsers for binary ARM and Thumb instructions are new, however, and are now complete, facilitating the analysis of compiled machine code.
- I have rebuilt the emulator responsible for launching ROP chains and reporting on their effects on the CPU context, which is essential for determining their fitness.
- I have completely redesigned the gadget extractor so that it now takes advantage of the type data synthesized by the instruction parser, better priming the initial population of chains.

3.2 Instruction Analysis & Gadget Synthesis

```
Ok, modules loaded: Gadget, ARMCommon, ARMParser, Aux, ElfHelper, ARM32, Thumb16.
*Gadget> testGadget "data/ldconfig.real" 2
------------------------------------------------------------
[00089018-00089038]: 8 instructions; SP moves 8
------------------------------------------------------------
00089018:  e0030590: ARM Mult --; r0 -> r3
0008901c:  e0854890: ARM MultLong --; r8 r0 -> r4 r5
00089020:  e0283198: ARM Mult --; r3 -> r8
00089024:  e0885005: ARM (DataProc ADD) --; r8 -> r5
00089028:  e0564004: ARM (DataProc SUB) --; r6 -> r4
0008902c:  e0c75005: ARM (DataProc SBC) --; r7 -> r5
00089030:  e1c940f0: ARM HalfWordDataR --; r0 r4 -> r4
00089034:  e8bd83f8: ARM BlockDataTrans --; r13 -> r3 r4 r5 r6 r7 r8 r9 r15
------------------------------------------------------------
------------------------------------------------------------
[00088760-00088768]: 2 instructions; SP moves 6
------------------------------------------------------------
00088760:  e1a00005: ARM (DataProc MOV) --; r0 -> r0
00088764:  e8bd80f8: ARM BlockDataTrans --; r13 -> r3 r4 r5 r6 r7 r15
------------------------------------------------------------

Number of gadgets: 453
*Gadget>
```

Progress Report # 4

November, 2016

4.1 Roadmap

- The emulator is now capable of evaluating not just shellcode, but full ROP-chains. Essential for mapping genotype to phenotype, and determining “fitness”.
- A baseline, ‘low-entropy’ compiler is ready – the worst of all possible proto-compilers – and generates random ROP-chains with minimal structure, just enough to preserve the integrity of the stack.
- A crude measure of fitness is now possible, enough to begin the natural-selective process.
- A simple mating algorithm has been defined, allowing the chains to sexually reproduce.

4.2 Random Chains with Stack Integrity

```
-------------------------------------------------------
[0001e818-0001e81b]: 3 instructions; SP moves 8
-------------------------------------------------------
0001e818:  e2844002: ARM (DataProc ADD) #&00000002; r4 -> r4
0001e81c:  e1a00004: ARM (DataProc MOV) --; r0 -> r0
0001e820:  e8bd83f8: ARM (BlockDataTrans LDMFD) --; r13 -> r3 r4 r5 r6 r7 r8 r9 r15
-------------------------------------------------------
,[Immediate: fb89de96]
,[Immediate: 6d3af1d6]
,[Immediate: a9c1796f]
,[Immediate: e2114840]
,[Immediate: 8f45f7aa]
,[Immediate: c21df982]
,[Immediate: 2567d5a6]
,-------------------------------------------------------
[000488f4-000488f6]: 2 instructions; SP moves 6
-------------------------------------------------------
000488f4:  e1a00005: ARM (DataProc MOV) --; r0 -> r0
000488f8:  e8bd81f0: ARM (BlockDataTrans LDMFD) --; r13 -> r4 r5 r6 r7 r8 r15
-------------------------------------------------------
,[Immediate: fb89de96]
,[Immediate: 6d3af1d6]
,[Immediate: a9c1796f]
,[Immediate: e2114840]
,[Immediate: 8f45f7aa]
,-------------------------------------------------------
```

4.3 ROP-chain Execution Emulation

The executable and data sections of the target binary are mapped into the emulator's virtual memory. A ROP-chain – a stack of addresses and immediate values – is pushed onto the target's stack, and then popped into its program counter, seizing the flow of control.

```
--| Tracing gadget at 0x0001e818, gadget size = 0x0000000c
    0001e818: e2844002
r0:  00000000  r1:  00000000  r2:  00000000  r3:  00000000  r4:  00000000  r5:  00000000  r6:  00000000  r7:  00000000
r8:  00000000  r9:  00000000  r10: 00000000  r11: 00000000  r12: 00000000  r13: 000b423c  r14: 00000000  r15: 0001e818
    0001e81c: e1a00004
r0:  00000000  r1:  00000000  r2:  00000000  r3:  00000000  r4:  00000002  r5:  00000000  r6:  00000000  r7:  00000000
r8:  00000000  r9:  00000000  r10: 00000000  r11: 00000000  r12: 00000000  r13: 000b423c  r14: 00000000  r15: 0001e81c
    0001e820: e8bd83f8
r0:  00000002  r1:  00000000  r2:  00000000  r3:  00000000  r4:  00000002  r5:  00000000  r6:  00000000  r7:  00000000
r8:  00000000  r9:  00000000  r10: 00000000  r11: 00000000  r12: 00000000  r13: 000b423c  r14: 00000000  r15: 0001e820

--| Tracing gadget at 0x000488f4, gadget size = 0x00000008
    000488f4: e1a00005
r0:  00000002  r1:  00000000  r2:  00000000  r3:  fb89de96  r4:  6d3af1d6  r5:  a9c1796f  r6:  e2114840  r7:  8f45f7aa
r8:  c21df982  r9:  2567d5a6  r10: 00000000  r11: 00000000  r12: 00000000  r13: 000b425c  r14: 00000000  r15: 000488f4
    000488f8: e8bd81f0
r0:  a9c1796f  r1:  00000000  r2:  00000000  r3:  fb89de96  r4:  6d3af1d6  r5:  a9c1796f  r6:  e2114840  r7:  8f45f7aa
r8:  c21df982  r9:  2567d5a6  r10: 00000000  r11: 00000000  r12: 00000000  r13: 000b425c  r14: 00000000  r15: 000488f8
```