Generating Optimised and Formally Checked Packet Parsing Code

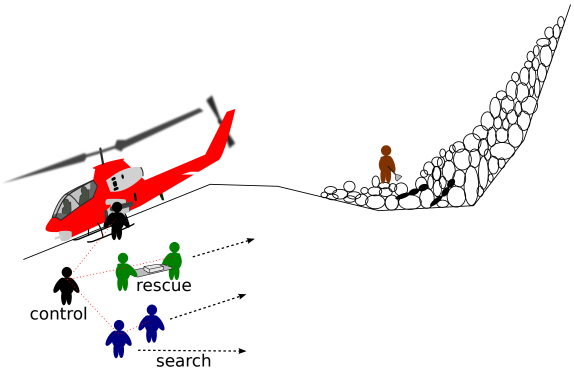
Sebastien Mondet, University of Oslo
Ion Alberdi,
Thomas Plagemann, University of Oslo

IFIP Sec'11

Plan

The Context

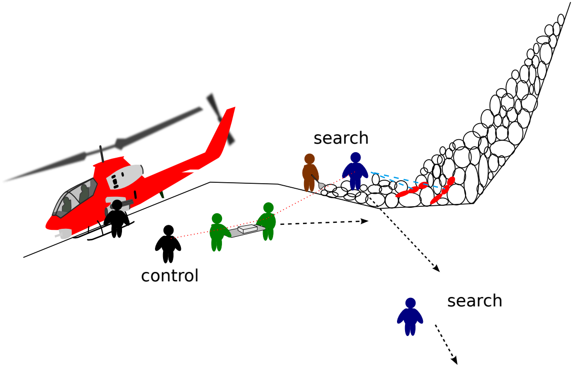
Protection Middleware for Mobile Ad-hoc Networks



Many papers: distributed authentication, firewalls, intrusion detection, etc.

The Context

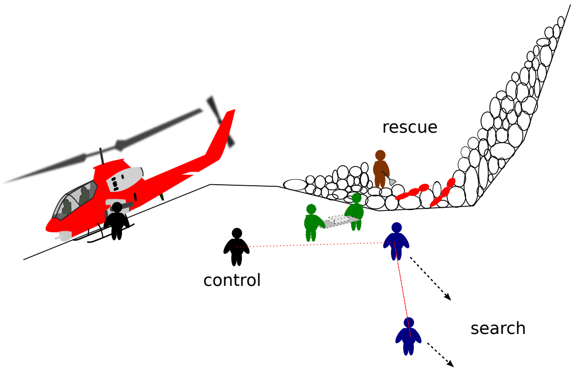
Protection Middleware for Mobile Ad-hoc Networks



Many papers: distributed authentication, firewalls, intrusion detection, etc.

The Context

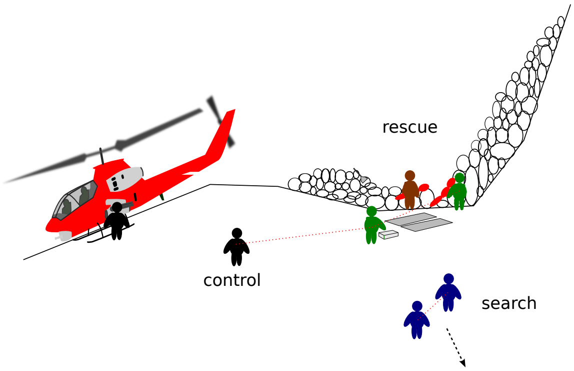
Protection Middleware for Mobile Ad-hoc Networks



Many papers: distributed authentication, firewalls, intrusion detection, etc.

The Context

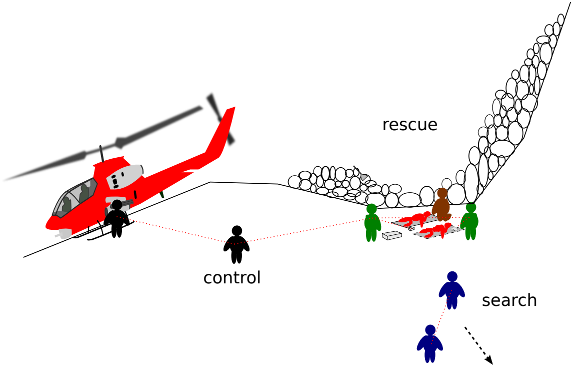
Protection Middleware for Mobile Ad-hoc Networks



Many papers: distributed authentication, firewalls, intrusion detection, etc.

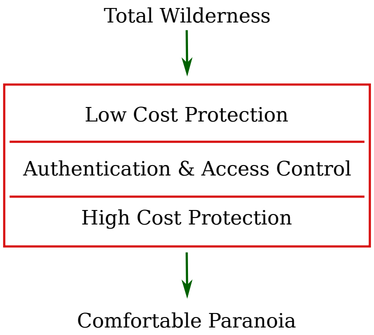
The Context

Protection Middleware for Mobile Ad-hoc Networks

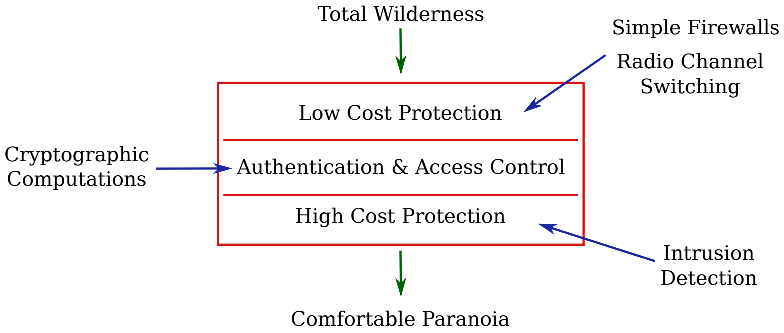


Many papers: distributed authentication, firewalls, intrusion detection, etc.

The Expected Middleware



The Expected Middleware



The Problem

Heterogeneity:

- From Sensors to Clusters …
- No common platform

Performance:

- Especially for mobile devices
- Protection is not the main purpose of most computing devices

Safety and Security:

- Cannot trust human beings writing code
- Protection software (e.g. in C++): often more flaws than benefits
- Responsibility …

Meta-programming

I.e. *Code Generation:*

- Adaption to the target device
- Adding formal/testing targets
- Safety *with* Performance
- Offline adaptability
- Asymmetrical distribution

Examples in the ecosystem: lex/yacc, FFTW [Frigo1999], etc.

More Precisely …

Let’s focus on Packet Parsing Code:

- First block useful for the other ones
- *A lot* of security problems related to packet parsing
- Had to start somewhere …

In a Nutshell

We have:

- A generator for packet parsing code
- Optimised C code output
- Automatic formal proofs of security properties

So, next:
Input/Output, Example, Discussion …

Input

The *User* writes:

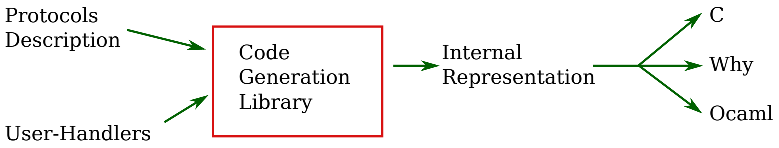
- A description of the (micro-)protocol layering graph (not only a *stack*):
  - The (binary) formats
  - The protocol/parsing transitions
  - Some run-time checks
- Parsing handlers:
  - The name of the format handled
  - A *request* on the fields (value, offset, pointer, and/or size)
  - A handler *function:*
    `request_results → continue_parsing`

Output

The *library* generates:

- Optimised C code:
  - verifications of buffer accesses are minimised
  - only the required fields (by the user, by the transitions, and by the run-time checks) and their *dependencies*
  - no malloc, minimised arithmetic, no function calls (except `ntohs` and `ntohl`)
- Easy-to-use OCaml code
- Input for Why [Filliatre2003] and Alt-Ergo [Conchon2007]
  → prove *automatically* that
  For any possible input packet:
  - **no wrong memory accesses can be provoked**
  - **infinite loops cannot be triggered**

The Block Diagram



An Example

Defining IPv4 Format

```
let ipv4_format =
  packet_format [
    fixed_int_field "version" 4;
    fixed_int_field "ihl" 4;
    fixed_int_field "tos_precedence" 3;
    fixed_int_field "tos_delay" 1;
    fixed_int_field "tos_throughput" 1;
    fixed_int_field "tos_reliability" 1;
    fixed_int_field "tos_reserved" 2; (* 16 bits *)
    fixed_int_field "length" 16;
    fixed_int_field "id" 16; (* 48 bits, 6 bytes *)
    fixed_int_field "reserved" 1;
    fixed_int_field "dont_fragment" 1;
    fixed_int_field "can_fragment" 1;
    fixed_int_field "frag_offset" 13; (* 8 bytes *)
    fixed_int_field "ttl" 8;
    fixed_int_field "protocol" 8;
    fixed_int_field "checksum" 16; (* 12 bytes *)
    fixed_int_field "src" 32;
    fixed_int_field "dest" 32; (* 20 bytes *)
    string_field "options" (size (`align32 (`sub (`mul (`var "ihl", `int 4),
                                               `add (`offset "dest", `int 4)))));
    payload
      ~size:(size (`sub (`var "length", `mul (`var "ihl", `int 4))))
      ~name:"ip_payload" ();
  ]
```

An Example

Defining IPv4's Transitions and Run-Time Checks

```
let ipv4_transitions =
  switch "protocol" [
    case_int_value  1 icmp "ip_payload";
    case_int_value  2 igmp "ip_payload";
    case_int_value  6  tcp "ip_payload";
    case_int_value 17  udp "ip_payload";
    case_int_value 33 dccp "ip_payload";
    case_int_value 47 gre  "ip_payload";
  ]
let ipv4_checks =
  [ check_range "ihl" 5 15 ]
```

An Example

Defining An IPv4 Handler

```
(ipv4,
 [ ‘value ”src”; ‘value ”dest”; ‘value ”version”;‘value ”ttl”;
   ‘value ”can_fragment”; ‘value ”frag_offset”;
   ‘pointer ”options”; ‘size ”options”; ‘value ”length”;],
 call_c_handler ”ipv4_handler_function”)
```

An Example
Some Generated C Code

```
value_of_version =
  (((unsigned char)(*((unsigned char *)((packet_pointer) + (0))))) >> (4)) & (15);
/* ... */
value_of_src = (unsigned int)(ntohl(*((unsigned int *)((packet_pointer) + (12)))));
/* ... */

if (current_packet_size <=
    ((value_of_ihl * 4) - (offset_of_dest + 4)) +
     (((value_of_ihl * 4) - (offset_of_dest + 4)) % 4) + 20))
{ /* Call Bad packet handler ... */ }
/* ... */
int users_decision = ipv4_handler_function(value_of_src, value_of_dest /* , ... */);
if (users_decision == 0) break;
/* ... */
if (value_of_protocol == 1) {  /* Next protocol is ICMP */
  next_packet_format = (unsigned long)(7);
  packet_pointer = (unsigned char *)(packet_pointer + offset_of_ip_payload);
  packet_size = (unsigned long)(packet_size - offset_of_ip_payload);
}
```

An Example

Some Generated Why Code

```
parameter bin_lsr : a:int -> b:int -> {} int { 0 <= result }
parameter bin_lsl : a:int -> b:int -> {} int { 0 <= result }
(* ... *)
parameter current_packet_size: int ref
parameter buffer_access:
   last_index:int -> { 0 <= last_index < current_packet_size } int { 0 <= result }
(* ... *)
   while !current_packet_format <> 5 do
     {
        invariant
        ( (current_packet_size >= 0) and
          ((current_packet_format = 3) or
           ((current_packet_format = 1) or
             (* ... *)
              ((current_packet_format = 14) or
               ((current_packet_format = 6) or
                (current_packet_format = 12))))))))))))))
        variant current_packet_size
      }
    begin
      (* ... *)
```

Formal Paranoia

“Formal Proving” always triggers the question:
‘Who and what do we have to trust?’

- Trust that the Why and C outputs are semantically equivalent
- Trust the following tools:
  - gcc: very buggy but well tested (and can use other compilers)
  - Why: algorithms have been formally proved but not the implementation
  - prover: Alt-Ergo

Current Limitations

As *work-in-progress* …

- currently does not implement any ad-hoc “parsing state”
  - the parsing of a format cannot use information of other (micro)-protocols
  - so protocols like DNS or DHCP require “manual” transitions
  - reassembling fragmented packets must also be done by hand
  - c.f. *Future Work*
- Errors are difficult to track from error-messages (Ergonomics)
- We prove that the parsing cannot “go wrong”, *but we do not yet* prove that it does what it is actually supposed to do

Experiments
The Tcpdump-lite

We generate a parser & pretty printer for Internet protocols:

- Ethernet, IPv4, ARP, GRE, TCP, UDP, etc.
- based on libpcap
- three different *"flavours"*
  - *Full*: requests & displays various meaningful fields
  - *Muted*: same fields as *Full* but no printing (except "error/alert" messages)
  - *Light*: requests & displays fields only for UDP and TCP

Experiments
Running The Tool-Chain

On an old Pentium 4, for the *Full* version:

- compiling the code generation program: 0.12 s
- running the code generation: 0.04 s
- compiling the generated C code: 0.13 s
- running the Why tool: 13.8 s
- proving all the goals with Alt-Ergo: 21 s

Experiments
Experimental Data

Two significant *PCAP capture files*:

- `Fuzz-10K`:
  - 10 000 packets
  - many are malformed, and contain potential “attacks”
- `24GRE-132`
  - 132 packets
  - 24 encapsulated GRE tunnels
    `Ethernet/IPv4/GRE/IPv4/GRE/IPv4/.../GRE/IPv4/UDP`

Experiments
Controlling The Amount Of Generated Code

	Size (bytes)		**Buffer Accesses**	
	Binary	C File	`Fuzz-10K`	`24GRE-132`
Full	19 157	35 027	242 050	8 457
Light	14 378	22 272	65 482	2 921

Table – Code Comparison of The *Full* and *Light* Versions

Experiments
Running The Code

Machine:	*Pentium 4*		*Core 2 Duo P8700*	
File:	`Fuzz-10K`	`24GRE-132`	`Fuzz-10K`	`24GRE-132`
Empty	10.97	6.88	5.16	2.40
Full	113.35	11.02	84.32	5.64
Muted	13.49	7.36	6.94	2.52
Light	19.71	7.44	11.97	2.61
T	122.38	10.79	86.57	5.86
T -v	168.76	14.67	123.49	8.69
T -vv	169.14	15.16	127.49	9.18
T -vvv	168.46	15.17	127.67	9.20

Table – Results for 2000 runs (times in seconds)

Experiments

Some More Stats …

- For the capture `Fuzz-10K`:
  OS+Pcap: 9.7 %, *Generated code*: 2.2 %, Printing: 88.1 %
  Average time per packet for the *Full* version: 5.67 µs
- For the capture `24GRE-132`: *OS+Pcap*: 62.4 %, *Generated code*: 4.4 %, Printing: 33.2 %
  Average time per packet for the *Full* version: 41.7 µs

Experiments
Adding An Output Langugue

OCaml support was added at the last moment; 4 hours to implement and test the OCaml output:

- 2.5h to implement the output itself and get “baby-tests” working
- 1.5h to get the full generated PCAP Internet pretty-printer, equivalent to the C one, tested.

Related Work
Bro's BinPAC

[Pang2006] *binpac: A yacc For Writing Application Protocol Parsers.*

- generates C++
- yacc-style mixed input
- only byte-aligned (no [arrays of] bits)
- "*State is maintained by extending parsing class*" i.e. "hand-made"
- protocol transitions are hand-made too (it's *Bro*'s code)

Related Work
GAPAL

[Borisov2007] *Generic Application-Level Protocol Analyzer and its Language*

- a high-level language to analyse *application* protocols
- *interpreter* written in C++

Related Work
Melange's MPL

[Madhavapeddy2007] *Melange: Towards a functional Internet*

- safest available: generates OCaml code
- many protocol transitions let to the user
  (but fit well with *ML style*)

Related Work
Comparison

All of them:

- **+** may handle more complex protocols and fragmentation thanks to ad-hoc state management
- **+** are *full-projects* (input language, website, documentation . . .)
- **-** have only one output language
- **-** do not give any formal proof
- **-** generate parsing code for the whole description of the protocols (useless code, and non-controllable size)

To Sum Up

So far:

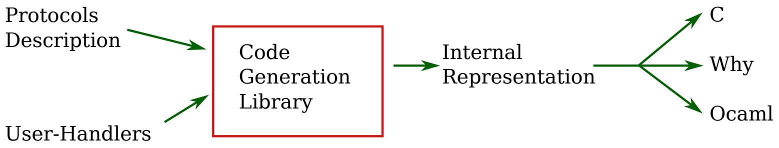
- simple and human readable input
- generation of packet parsing code
- optimised C code
- automatic formal proof of safety features

Future Work

- State/Persistence/Memory Management:
  - if done well, much wider than parsing (generating the rest of the middleware)
  - goal: fully parse any format without user help
- Performance:
  - More aggressive optimisations, [Castelluccia1997]
- Improve the proving:
  - more properties: code computes what the user has requested, no useless/dead code, etc.
  - framework for distributed proving, and proof reuse (make-like)
  - improve error reporting *ergonomically* (use previous framework to narrow the problems, try to detect most common errors earlier in the process)
  - make the "full chain" work with other provers (e.g. *Zenon* – *de Bruijn* criterion)
  - formally checked re-implementation of the "trust bottleneck"

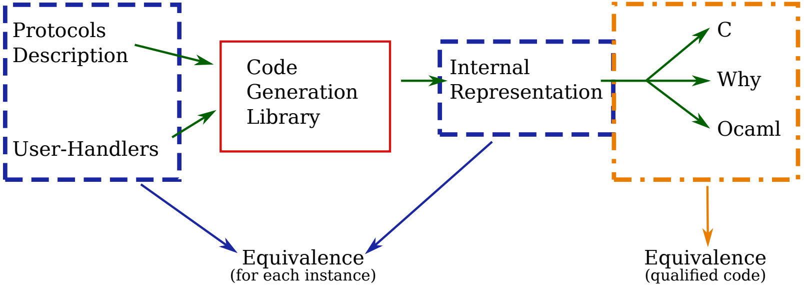
Future Work

To Prove



Future Work

To Prove



About Formal Methods

Hacking with formal stuff is also/more fun …

- Always something new to learn and hack with (both High- and Low- level)
- When something is *done*, it is **solid**
- Stop the propagation of The Disaster instead of feeding it
- It is *fun* → Look for “Software Foundations”, B. C. Pierce et al.

This Is The End

Thanks, Questions, Comments, Jobs?

Contacts

Generating Optimised and Formally Checked Packet Parsing Code

- **Sebastien Mondet**, University of Oslo, `smondet@ifi.uio.no`, `http://seb.mondet.org`
- **Ion Alberdi**, `ion.alberdi.research@gmail.com`
- **Thomas Plagemann**, University of Oslo, `plageman@ifi.uio.no`, `http://heim.ifi.uio.no/~plageman/`

References I

- `[Frigo1999]` Matteo Frigo ; *A fast Fourier transform compiler*. ACM SIGPLAN Notices, Volume 34, Issue 5, 1999.
- `[Filliatre2003]` Jean-Christophe Filliâtre ; *Why: A Multi-Language Multi-Prover Verification Tool*. Research Report 1366, LRI, Université Paris Sud, 2003.
- `[Conchon2007]` Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer ; *Lightweight Integration of The Ergo Theorem Prover Inside a Proof Assistant*. Proceedings of the second workshop on Automated formal methods, 2007.
- `[Pang2006]` Ruoming Pang, Vern Paxson, Robin Sommer, and Larry Peterson ; *binpac: A yacc For Writing Application Protocol Parsers*. Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, 2006.

References II

- [Borisov2007] Nikita Borisov, David Brumley, Helen J. Wang, John Dunagan, Pallavi Joshi, and Chuanxiong Guo ; *Generic Application-Level Protocol Analyzer and its Language*. Proceedings of the Network and Distributed System Security Symposium, NDSS'07, 2007.
- [Madhavapeddy2007] Anil Madhavapeddy, Alex Ho, Tim Deegan, David Scott, and Ripduman Sohan ; *Melange: Towards a functional Internet*. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, 2007.
- [Castelluccia1997] Claude Castelluccia, Walid Dabbous, and Sean O'Malley ; *Generating efficient protocol code from an abstract specification*. IEEE/ACM Transactions on Networking (TON), Volume 5, Issue 4, 1997.

About Testing

The “24 GREs” header length is a bit more than 600 bytes long, and $2^{8 \times 600}$ is quite a big number:

```
    87898 039204788 324925783 294151808 719402282 887071042 527340109 818285189 479347476 280738800 294433431 367894035 807990755 244809818 272023764 961381305

022250822 357470653 053907176 395036013 086794211 678347992 710891473 660639564 531752836 798248860 539589278 934330893 390996149 386567641 287056231 340459740

904172065 305971845 815799389 013662005 947076315 485830002 339080254 823931672 242049945 663214901 268959402 801955167 327783014 623154365 050794602 004967486

924919697 268934845 649227052 238071569 186822353 545103118 275855411 156180259 555739855 174253896 900332483 877411609 205996634 091134158 394255100 498334827

334946202 870597933 355803093 379167510 822235120 696993574 397468850 429986496 285306645 484578147 452267923 641194990 104371583 104636690 257284887 683043911

170630987 561301779 478101726 171822256 436435009 261235234 603825827 935643410 163401593 197125083 890261146 419052229 993008308 429415502 543800453 494465091

586322704 206500441 694572157 206809627 749201455 691550107 728035193 825784189 677789643 070587902 791756930 655177341 206308602 793965821 463151154 076832144

336308139 377824724 039777126 114346215 760728239 473957883 559191783 337100127 672649209 019207754 199004306 397255944 961856663 670548703 629320409 560225932

384478576 976210497 889755369 549845504 701102002 126666770 697347898 576116823 507136178 582760212 238521083 159821203 737531811 353385056 931835084 128761446

654606030 815034742 826163516 493820202 522149157 210943344 211549624 991974238 173452620 633250503 973226046 668369517 253720964 343419863 308669488 566515865

407717376
```