Generating Optimised and Formally Checked Packet Parsing Code

Sebastien Mondet, Ion Alberdi, Thomas Plagemann

Generating Optimised and Formally Checked Packet Parsing Code

Sebastien Mondet[1], Ion Alberdi[2], and Thomas Plagemann[1]

[1] University of Oslo, Norway {smondet,plageman}@ifi.uio.no
[2] ion.alberdi.research@gmail.com

Abstract While implementing distributed applications, the parsing of binary packets is a very difficult and error-prone task the developer has to face. Moreover, these programming mistakes are often the source of distant vulnerabilities. In this paper we present a code-generation library, called Promiwag, for creating optimised and safe packet parsing code. Its input is concise human-readable descriptions of the protocols and the interests of the application in specific pieces of information. Promiwag follows a dependency-based algorithm, and uses high-level optimisation techniques to generate minimal parsing automatons. These automatons can be compiled into C or OCaml code for efficient execution, and to annotated Why code. This latter output is then used to automatically prove that for any possible input packet, the generated code cannot perform any illegal memory access, and that no infinite loop can be triggered. We have used our code generator to implement a pretty-printer for Internet protocols, and we provide experimental results on the performance of the generated code.

1 Introduction

One of the currently observable trends for future distributed applications is the convergence of Internet and Mobile networks. More and more small, mobile, and cheap electronic devices must be able to communicate with personal laptops, servers, and/or *clouds*. Therefore, one needs to implement specialised and complex software for many different platforms. Some of these platforms require special attention to be given to performance. Indeed, computing resources may be very limited, e.g., on mobile devices. In parallel, the error-proneness and the lack of static guaranties of the common tools and languages used by developers (unsafe memory accesses, dynamic typing, etc.) lead to a very large amount of bugs and security vulnerabilities present in distributed applications[3].

One common task that many distributed application developers have to handle is the parsing of binary protocols. For example, this is the case in almost every piece of protection software (firewalls, intrusion detection systems, etc.). The code handling these aspects, especially when hand-written in C or C++, is very often swarming with implementation mistakes. The latter often lead to

[3] c.f. CVE (Common Vulnerabilities and Exposures); cve.mitre.org

exploitable buffer over-flow vulnerabilities or easy-to-trigger denial-of-service attacks [18].

Code generation techniques (a.k.a. *Meta-Programming*) have already been often used to automate programming tasks. Examples range from the `lex` and `yacc` tools developed in the 70s, to the compiler of the “Fastest Fourier Transform in the West” [11]. Generating code from a higher level representation allows adapting the output specifically to the target platform. It also allows providing aggressively optimised code while keeping the required level of safety. From the initial input representation, one can also generate additional output which may assist the developer in other ways; e.g. for testing or documentation extraction.

The safety provided by code generation will often be much better than with hand-written code. However, the code generator itself is generally a piece of software developed by one or more human beings. As such, it can (and will) be the victim of many programming mistakes. How can one trust the code generated by an external tool? Expert and careful proof-reading of thousands of lines of generated code is by far too expensive for most software projects; and *testing* requires identifying in advance all the possible ways to make a program go wrong. When dealing with input as binary packets which can be erroneous, or maliciously crafted, both approaches are not realistic. We address this issue by mathematically ensuring the safety of the generated code against any possible input. Formal properties are automatically proved on the output of the Promiwag library.

In this paper, we focus on the generation of code for parsing network packets. From a high-level representation of network protocols and their handling by the user, our code generator creates optimised code on which safety and security properties are formally proved. The user describes i) the binary formats of the packets, ii) the transitions between (micro-)protocols (e.g. how to pass from IP to TCP), iii) a few logical properties to insure, iv) which pieces of information he is interested in, and v) what user-function to call with the parsing results as arguments. Out of these concise declarations, the Promiwag library builds a parsing automaton which parses only the necessary fields (i.e. the fields required by the user and their dependencies). This intermediate representation also contains logical properties about the algorithms involved. It is then compiled into i) C code for high performance parsing, ii) OCaml code, and iii) annotated Why code [10]. This latter output allows, thanks to the Alt-Ergo theorem prover [8], automatically proving that, for any possible input packet, no wrong memory accesses can be provoked, and no infinite loops can be triggered.

In Section 2, we detail and discuss the design of our code generation approach. Then, in Section 3, we assess the performance of the code through experiments. We discuss related projects in Section 4, before concluding and suggesting future work (Section 5).

2 Packet Parsing Code Generation

Figure 1 illustrates the high-level idea of the structure of the code generation process in the Promiwag library. We have developed our code generation tool as an Objective Caml[4] library. The user can link their programs with the library to generate the code. A dedicated and independent input language could be easily implemented, but the slight benefit of readability would not overcome the loss in flexibility. Indeed, as Promiwag can generate code for different target platforms, and can adapt the code to the needs of the user, having a powerful programming language can be very practical for code reuse and maintainability.

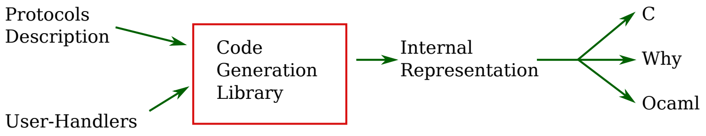

Figure 1. The Structure of the Code Generation Process in the Promiwag Library

We present the code generation process from an input/output point of view in Sections 2.1 and 2.2. Afterwards, we discuss the limitations and some issues related to the current state of our implementation in Sections 2.3 and 2.4.

2.1 Input

For each protocol or *micro-protocol* (e.g. for a given type of packet header, and its links to the "upper" layers), the input required from the user is composed of two declarative pieces of code; the (micro-)protocol description and the user handler.

The first, is the description of the (micro-)protocol itself. The user gives a precise definition of the binary format, i.e., they name the fields and their size(s). A field can be an integer of any size between 0 and 32 bits, or a string of any byte size. They also provide the "parsing transitions", which are a sequence of *switch-like* statements to tell the generator how to continue the parsing. For example, given the value of a protocol field, the parser has to continue with another (micro-)protocol included in a given payload. Finally, the user can provide "run-time checks" that tell the generator to include assertion checking code, about the fields of the parsed packet. The goal of the run-time checks is not to write a full stateless firewall, but to insure a few consistency properties needed for the formal proofs on the code. For example, the soundness of the binary format may depend on assumptions about certain fields in a given (micro-)protocol.

The second declaration, the *user-handler*, is optional. It describes the user's interest in specific parsable values and their handling by an external function.

[4] ocaml.org

Each user-handler refers to a described (micro-)protocol. The user specifies their request for values, offsets, pointers, and/or sizes in the binary format, and provides a function in the target programming language (C or OCaml) which expects these values as arguments. The function returns a boolean/integer deciding whether to continue parsing or not.

```
let ipv4_format = packet_format [
    fixed_int_field "version" 4;
    fixed_int_field "header_length" 4;
    fixed_int_field "tos_precedence" 3;
    fixed_int_field "tos_delay" 1;
    fixed_int_field "tos_throughput" 1;
    (* ... Skipped ...*)
    fixed_int_field "dest" 32;
    string_field "options"
      (size (`align32 (`sub
        (`mul (`var "header_length", `int 4),
         `add (`offset "dest", `int 4)))));
    payload  ~name:"ip_payload"
      ~size:(size (`sub (`var "length",
               `mul (`var "header_length", `int 4))))
  ]
let ipv4_transitions = switch "protocol" [
    (* ... Skipped ...*)
    case_int_value  6  tcp "ip_payload";
    case_int_value 17  udp "ip_payload";
    (* ... Skipped ...*)
    case_int_value 47 gre  "ip_payload";
  ]
let ipv4_checks = [ check_range "header_length" 5 15 ]
```

Listing 1.1. Example of Description of IPv4's Protocol

```
(ipv4_name, [ `value "src"; `value "dest"; `size "ip_payload" ],
 call_target_handler "ipv4_handler_function")
```

Listing 1.2. Example of User-Handler for IPv4

Listings 1.1 and 1.2 show (shortened) examples of user code regarding the IP protocol (version 4). The first is a (micro-)protocol description; the binary format, the parsing transitions, and a single run-time check. The second listing is the corresponding user-handler specification. In the binary format description, one can see how variable-length fields like the "`options`" are created with arithmetic expressions. This particular case requires a run-time check. Indeed, the positiveness of the size of the field depends on the fact that the field "`header_length`" is between 5 and 15 as required by the RFC 791. If the corresponding run-time check is not provided by the user, the formal proving of the parsing code will fail. This non-provability matches a hypothetical malicious attack of crafted IP packets containing wrong "`header_length`" fields. Note that this run-time check is the only one needed to implement and prove the *Tcpdump-lite* experiment in Section 3.

2.2 Output

From the previously described declarations, Promiwag builds the parsing automaton code. It starts from the fields required by the user request, the parsing transitions, and the run-time checks, and then follows their dependencies to minimise the amount of code. This parsing automaton is expressed in an internal representation. This representation contains the parsing algorithm in a strictly-typed imperative language. The language is low-level as it can handle pointer arithmetic, bit-wise operations, etc., but platform independent (for example, endianness is explicit). It is also *annotated* with logical properties gathered during its construction. Many human-readable comments are also kept for debugging or proof-reading purposes. During the code-generation, the library performs relatively high-level code optimisations, including constant propagation, partial evaluation, common sub-expression elimination, and "factorisation" of buffer access verifications. The lowest-level optimisations are left to the back-end compilers of the output languages; implementing them would be redundant.

In the example presented in Listings 1.1 and 1.2, the initial "required fields" correspond to the ones requested by the user (the values of "`src`" and "`dest`", and the size of the "`ip_payload`"), the field "`header_length`" for the run-time check, the value of "`protocol`" and the offset of the "`ip_payload`". The generator needs the last two to be able to continue the parsing with the next protocol. Then, the computation of, for instance, the length of the "`options`" field will be required as a dependency in order to compute the offset of the payload.

The internal representation of the parsing code can then be "compiled" into the three targets shown in Figure 1. On the one hand, the two "programming language" targets are C and OCaml. The former may be seen as the main target, as the code generator was originally designed for C. The transformation to this language is parametrised by a representation of the target platform (endianness, sizes of the types, etc.). The OCaml output was implemented in order to *show* that adding other target languages is a relatively easy task. Implementing this transformation from scratch took only 4 hours for one developer. After this time, the OCaml version of the *Tcpdump-lite* application presented in Section 3 was successfully tested. Generating OCaml is useful by itself: it is used for some experiments for instance to count the buffer accesses in Table 1.

On the other hand, Promiwag's code generator uses the intermediate representation to generate input for the *Why* tool. *Why*[5] is a "verification condition generator" created by Jean-Christophe Filliâtre. For a given annotated algorithm, it propagates "weakest preconditions" [9] and generates "proof goals" when an assertion needs to be proved. The annotations are, generally, intermediary proof goals or "hints" to *guide* the proving steps. The generated proof goals must then be proved by another tool, an automatic prover and/or a proof assistant.

In our case, we use the "fast" weakest precondition algorithm presented by Barnett and Leino [2]. Thanks to the annotations, the proof goals can be automatically proved by the Alt-Ergo[6] theorem prover [8]. The *Why* code generated by the Promiwag library represents the algorithm of the parsing automaton. The user-handlers for the protocols are ignored; we prove only the code we generate.

[5] why.lri.fr

[6] alt-ergo.lri.fr

The accesses to the packet buffer are abstracted as logical functional parameters. They are modelled in order to require a proof that they are not out of bounds each time they are "called/used", and at the same time, to express that accessing a packet can lead to *any* arbitrary value, i.e., there is absolutely no assumption on the content of the packet. The automaton is able to parse "protocol loops", like IP/GRE/IP, but we provide enough annotations to guide the proof that the parser always strictly "advances" in the packet. Hence, given that a packet has constant arbitrary size, the tools can prove that the parsing always finishes eventually.

At the end of a successful proving process, we can assert that the generated parsing automaton verifies the following theorem:

For any possible input packet,
– **no unsafe memory accesses can happen, and**
– **no infinite loops can be triggered.**

2.3 Current Limitations

As *work-in-progress*, the Promiwag library currently does not implement any ad-hoc "parsing state" management. This means that the generated code cannot *yet* use information from previous packets, and/or from other (micro-)protocols in the same packet. The implementation of features like the handling of IP fragmentation or the transitions present *inside* protocols like DNS or DHCP are subject to future work. They have to be implemented for now within the user's handlers; note that the user can still use generated code for the low-level parsing related to this.

The theorem which can be proved on the code, is certainly a huge benefit over most other implementations of packet parsing; especially from a security standpoint. The proof asserts that the code cannot be used to provoke a fatal error. However, we do not have *yet* any formal proof that the code does what it is actually supposed to do. This means that checking the correctness of the computation of a given field of a (micro-)protocol requires testing. Of course, testing for example that the "header length" field parsed for an IP packet has the expected value is much easier than trying to imagine all the possible ways to alter it in order to make the program crash. Moreover, in C or C++ programs, the error reporting of unsafe memory accesses depends on the exact memory layout at the time of testing. They may be completely silent in some cases and appear later *in production*; this problem is completely addressed by the current proof.

Finally, from an *ergonomics* point of view, the error reporting related to formal proving is still quite difficult to understand. In the example of Listings 1.1 and 1.2, if one forgets the run-time check on the "header length", the theorem prover just fails saying "I don't know" about a possibly very long proof goal. Then, tracking down the error to find the original mistake is not an easy task. For now, we have implemented a very explicit naming of generated variables together with high-level comments which are propagated to the different outputs. While these are helpful, more sophisticated techniques will be needed in the future.

2.4 Trust Issues

When hearing about mechanised and/or automated proofs, one has to ask the question: ‘Who and what do we *really* have to trust?’ Regarding the theorem proved on the generated code, apart from the soundness of the underlying mathematical framework, one has to trust our work in some respects, and the external tools in others.

On the one hand, the trust *bottleneck* in the Promiwag library is the final *compilation* stage. Currently, there is no proof that the different outputs are semantically equivalent. This means that when the theorem prover succeeds, one has to trust *us* that the C and OCaml outputs are equivalent, and actually *proved* by the proof on the Why code. To address this, we have restricted the size of the critical code to the minimum. We also note that implementing the OCaml output in half a day (c.f. Section 2.2) partially shows that this part of the code is quite straight-forward.

On the other hand, the whole tool-chain depends on many external tools. First, one has to trust the theorem proving tools; Why and Alt-Ergo themselves. Both rely on formally proved algorithms, but their implementations are not. Note that other theorem provers can be used with Why if needed. Second, every software project has to trust its compilers; in our case, the C and OCaml compilers. For C, we use the GNU Compiler Collection (GCC). Even though it is widely tested, at least for the C language within the mainstream architectures, many bugs are often found in official releases. As a side note, we have successfully tested the compilation of the C code generated for our Tcpdump-lite example with Compcert[7]. Compcert is a formally proved compiler for a subset of the C language [13]. For OCaml, we use the standard compilers. They may also be a trust issue but a substantial part of the system (compiler and run-time) has already been qualified under the DO-178B standard to be used as tool for critical aircraft software development [16].

3 Experimental Results

In this section, we provide the results of our experimental study into the performance of the generated code. We aim to show that the use of a code generator does not degrade the performance compared to hand-written code, and that the size of the generated code is still quite controllable by the user.

We have used the Promiwag library to implement a pretty printer for Internet protocols (Ethernet, IPv4, ARP, GRE, TCP, UDP, etc.). As previously mentioned, we have named it “Tcpdump-lite”. It is based on libpcap[8], and uses our code generator to call user-handlers which are only pretty-printing the requested values (`printf`). We have generated three different *“flavours”* of Tcmpdump-lite:

- *Full*: requests various meaningful fields for every (micro-)protocol, and displays them.

[7] compcert.inria.fr

[8] www.tcpdump.org

- *Muted*: requests the same fields as *Full* but does not print them; (micro-)protocol user-handlers are just empty, it just prints “error” messages (i.e. alerts when packets are malformed).
- *Light*: requests and prints fields only for UDP and TCP.

To estimate the running times of the tool-chain, we just measure the times taken by the different programs on an old Pentium 4 desktop computer for the *Full* version: i) compiling a code generation program and linking against our library: 0.12 s; ii) running the code generation: 0.04 s; iii) compiling the C code output with GCC: 0.13 s; iv) running the Why tool: 13.8 s; and v) proving all the goals with Alt-Ergo: 21 s.

We have done experiments listening to *“live”* network interfaces and with many “PCAP capture files” (generated by us or found on the Web). To simplify their presentations, we present here the results obtained with two characteristic files. We call the first one `Fuzz-10K`, it is a capture of 10 000 packets, where many are malformed, and contain potential “attacks”. The second one is `24GRE-132` which is composed of 132 packets among which many are using 24 encapsulated GRE tunnels. This means that for those packets the “protocol stack” is: `Ethernet/IPv4/GRE/IPv4/GRE/IPv4/.../GRE/IPv4/UDP` where there are 24 GRE encapsulations. Note that handling multiple encapsulations is an important feature [1] which many networking tools cannot handle; explicitly or not. For example the Section 1.8.1 of the Snort manual [19] states that for more than one GRE tunnel, packets will not be inspected, and the Linux kernel simply crashes for more than 36 encapsulations (c.f. Debian bug number 599816).

Table 1. Code Comparison of The *Full* and *Light* Versions

	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

The first experiment is an analysis of the control the user has on the generated code. Table 1 shows a comparison between the *Full* and *Light* versions of our Tcpdump-lite. We provide the raw size of the whole program, and the number of buffer accesses actually executed during the parsing of both capture files. This shows that, from the same set of (micro-)protocol descriptions, just by removing requests and user-handlers, the user can lighten the application significantly. It matches the goal of writing software *adapted* to the resources of the target device. For example, one can make a special version of a firewall application for mobile phones with a more affordable development effort.

We also provide results for brute performance evaluation. We compare the running times of the different versions of Tcpdump-lite, against an “empty” PCAP application (i.e., the same code as Tcpdump-lite but without any parsing of the packets) and different options of the original Tcpdump application. To be less unfair with the latter, we use the options `-n` so that IP addresses are not converted to host names and `-K` so that checksums are not computed. We also redirected all outputs to `/dev/null`. We measure Tcpdump's (noted "T") running times for the different verbosity levels: the default, `-v`, `-vv`, and `-vvv`. In order to try to reduce the entropy of the measurements, we run them 2000 times on an old machine (*Pentium 4*) and on a more recent one (*Core 2 Duo P8700*).

Table 2. Results for 2000 runs (times in seconds)

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

The results are presented in Table 2. Note that the `Fuzz-10K` capture involves some malformed packets which trigger error-message printing even for the *Muted* version, and that the `24GRE-132` one induces a lot of parsing logic compared to the display of information. If we assume the two following approximations: i) the *Empty* program represents the operating system plus the "PCAP machinery", and ii) the difference between *Full* and *Muted* measures the time spent in printing, then we can compute the following statistics (for the *Pentium 4*[9]):

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

The running times for the original Tcpdump are given as a comparative indication. Indeed, the application is more feature-full, and dynamically configurable (i.e. on command line).

[9] The results for the *Core 2 Duo P8700* are slightly better for the generated code but the *true* parallelism of the architecture may make these computations *less meaningful*.

4 Related Work

The use of code-generation techniques in order to improve the performance and/or the safety of distributed systems has been gaining a lot of interest in literature. For instance, with the *Statecall Policy Language* presented by Madhavapeddy et al. [14], one can describe statefull automatons which are *compiled* into three different targets: OCaml code to embed in an application, Promela code to check temporal properties with the SPIN[10] model checker, and HTML/Ajax code for real-time monitoring of the application. Another example is the declarative sensor networks (DSN) platform (Chu et al. [7]). There, the user describes application overlays using a declarative language (paradigm similar to Prolog) and the compiler generates code for sensors in NesC (the C dialect for the TinyOS operating system).

Regarding the management of packet streams, two projects use meta-programming for high-level packet filtering purposes. First, the BPF+ packet filter [3] uses a high-level language to describe boolean predicates on packet flows. This language is compiled to bytecode which can be verified or interpreted (or JIT-assembled). Among the verifications, the *halting* of the application is insured by forbidding any kind of cycles or loops in the filtering program. Second, the FFPF project (*Fairly Fast Packet Filter* [5]) implements a filtering language which allows one to “chain” (mostly independent) packet analysis applications while sharing kernel-space buffers to avoid copying.

Examples of projects using code-generation especially for packet parsing are *Binpac*, *GAPAL*, and *Melange's MPL*. Binpac [17] is a C++ code generator for the Bro intrusion detection system[11]. It uses a yacc-like language to mix the specification of binary formats and user's C++ code in order to help them in their task. *State* and parsing transitions are written directly by the user as well as bit-level accesses, e.g. by including inline C++ code. GAPAL [4] is a high-level language for analysing application protocols which is evaluated by a C++ interpreter. The Meta-Packet Language of the Melange framework [15], is a domain specific language to describe Internet protocols. Its compiler generates optimised OCaml pieces of code for each protocol; the use of a memory-safe language naturally avoids wrong buffer accesses. It also features ad-hoc parsing state management for situations commonly found in Internet protocols. Parsing transitions between protocols are left to the developer (but they fit well with ML-style programming).

Even though these projects may handle more complex protocols and/or fragmentation thanks to ad-hoc parsing-state management, they all have a unique output language and they do not give any formal proof on the generated code. Moreover, they all generate code for the whole defined set of protocols, i.e. even for non-used portions. As they rely on a lot of hand-written code, brute performance comparisons would be quite meaningless. For example, to compare the performance of the *Tcpdump-lite*, implemented with Promiwag, against another

[10] spinroot.com

[11] www.bro-ids.org

one implemented with *Binpac*, the developer would have to implement the parsing transitions or the “bit-level” accesses directly in C++. This would mean comparing a lot of generated code against hand-written code.

5 Conclusion

Our goal is to provide reduced development costs while keeping performance and enforcing safety. We have presented a code-generator which, from concise and high-level descriptions, can generate optimised code for parsing binary packets. The code is safe-by-construction thanks to automated formal proofs done on the output. The user control the amount and the performance of the generated code. Indeed, the generation is *dependency-based*; we build only the code that is actually needed by the user. The experiments we provide assess that the run-time performance will not be degraded by the use of our library.

Future work will be divided in three directions. First, state, persistence, and memory management are needed to add features to the parsing generation. This problem is much more generic than just packet parsing and will have to be treated as such to be used in other aspects of distributed applications. Second, more aggressive optimisations on the code could be implemented. As the performance of a given program transformation depends on the actual code, potential optimisations could be evaluated on the generated code in order to choose the right *option* at the last time. Finally, the formal proving framework gives us a lot of room for interesting improvements. More properties could be proved on the code, e.g. the accuracy of the computations regarding the user’s requests, or the absence of dead code. The “trust bottleneck” of the Promiwag library, i.e. the code which transforms the (last) internal representation to the different targets, could be re-implemented in a *directly* certified way with a proof assistant like Coq[12]. This family of formal tools have “executable code extraction” capabilities, and these kinds of program transformations, even if they require a lot of specification and proving work, are well suited for programming with proof assistants [12,6].

References

1. Alberdi, I., Owezarski, P., Nicomette, V.: Luth: composing and parallelizing midpoint inspection devices. In: NSS ’2010: Proceedings of the 4th International Conference on Network and System Security. pp. 9–16. IEEE Computer Society, Melbourne, Australia (September 2010)
2. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: PASTE ’05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 82–87. ACM, New York, NY, USA (2005)

[12] coq.inria.fr

3. Begel, A., McCanne, S., Graham, S.L.: Bpf+: exploiting global data-flow optimization in a generalized packet filter architecture. In: SIGCOMM '99: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication. pp. 123–134. ACM, New York, NY, USA (1999)
4. Borisov, N., Brumley, D., Wang, H.J., Dunagan, J., Joshi, P., Guo, C.: Generic application-level protocol analyzer and its language. In: NDSS (2007)
5. Bos, H., de Bruijn, W., Cristea, M., Nguyen, T., Portokalidis, G.: FFPF: Fairly Fast Packet Filters. OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation (2004)
6. Chlipala, A.: Certified Programming with Dependent Types. Online in-progress textbook (2009)
7. Chu, D., Popa, L., Tavakoli, A., Hellerstein, J., Levis, P., Shenker, S., Stoica, I.: The design and implementation of a declarative sensor network system. Proceedings of the 5th international conference on Embedded networked sensor systems (2007)
8. Conchon, S., Contejean, E., Kanig, J., Lescuyer, S.: Lightweight integration of the ergo theorem prover inside a proof assistant. In: AFM '07: Proceedings of the second workshop on Automated formal methods. pp. 55–59. ACM, New York, NY, USA (2007)
9. Filliâtre, J.C.: Verification of non-functional programs using interpretations in type theory. J. Funct. Program. 13(4), 709–745 (2003)
10. Filliâtre, J.: Why: A Multi-Language Multi-Prover Verification Tool. Research Report 1366, LRI, Université Paris Sud (2003)
11. Frigo, M.: A fast Fourier transform compiler. ACM SIGPLAN Notices, Volume 34, Issue 5 (1999)
12. Leroy, X.: Mechanized semantics. In: Logics and languages for reliability and security, NATO Science for Peace and Security Series D: Information and Communication Security, vol. 25, pp. 195–224. IOS Press
13. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), 107–115 (2009)
14. Madhavapeddy, A.: Combining Static Model Checking with Dynamic Enforcement using the Statecall Policy Language. International Conference on Formal Engineering Methods (2009)
15. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R.: Melange: Towards a functional Internet. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (2007)
16. Pagano, B., Andrieu, O., Moniot, T., Canou, B., Chailloux, E., Wang, P., Manoury, P., Colaço, J.L.: Experience report: using Objective Caml to develop safety-critical embedded tools in a certification framework. In: ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming. pp. 215–220. ACM, New York, NY, USA (2009)
17. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: a yacc for writing application protocol parsers. In: IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement. pp. 289–300. ACM, New York, NY, USA (2006)
18. SANS Institute: Top 20 internet security problems, threats and risks. section 5 anti-virus software (2007), `http://www.sans.org/top20/2007/#s5`
19. Snort Team: Snort Users Manual. The official documentation produced by the Snort team at Sourcefire (2010)