documentation of the BISMON static source code analyzer.

This report and the *Bismon* static analyzer has been funded by European H2020 projects **www.chariotproject.eu** (grant 780075) and **www.decoder-project.eu** (grant 824231).

The opinions are those of the author(s) only. The content of the publication herein is the sole responsibility of the publishers and it does not necessarily represent the views expressed by the European Commission or its services.

author: Basile STARYNKEVITCH (CEA, LIST) DILS/LSL
from its SOFTWARE SECURITY LABORATORY
basile.starynkevitch@cea.fr
http://starynkevitch.net/Basile/

unverified, unapproved, unchecked draft **Ful of Mistaks**.

Some version of this **DRAFT** report might be downloadable from **http://starynkevitch.net/Basile/bismon-doc.pdf**

This partly generated document and the *Bismon* software itself are co-developed in an agile and incremental manner, and have exactly 3932 `git` commits on *Mon 01 Feb 2021 03:26:24 PM MET*. For more, please see **https://github.com/bstarynk/bismon/commits/** for details. These commits are too many and too fine grained to be considered as "versions". Given the agile and continuous workflow, it is unreasonable, and practically impossible, to identify any formalized versions.

This document is co-developed with the *Bismon* software itself, it was typeset using LaTeX on Linux and contains some *generated* documentation [1], mixed with hand-written text. During development of `bismon`, the amount of generated documentation will grow. The entire history of *Bismon* (both the software -including its persistent store- and this document) is available on **https://github.com/bstarynk/bismon/commits** and has, for this document of commit id `61e424abbb482e7e++` (done on *2021-Feb-01*) generated on *Feb 01, 2021*, exactly 3932 commits (or elementary changes). Since changes on any file in the `git` repository can affect this document, no "version" is identifiable.

For convenience to the reader, here are the last [2] three `git commit`-s:

```
commit 61e424abbb482e7eb81dcd2da7d12bb1e667f2b8
Author: Basile Starynkevitch <basile@starynkevitch.net>
Date:   Mon Feb 1 14:08:34 2021 +0100

    mention both chariot and decoder

M       doc/bismon-doc.tex

commit 9e198ce90e47a60a45c6cdc025c166fb159d3bf4
Author: Basile Starynkevitch <basile@starynkevitch.net>
Date:   Mon Feb 1 11:05:16 2021 +0100

    the output of ./bismon -help goes into the documentation

M       build-bismon-doc.sh
M       doc/appendix-bm.tex
M       doc/bismon-doc.tex
A       doc/genscripts/005-bismon-help.sh

commit 3eee9ed1ff85ef0f28ef74049c31014618bd72e7
Author: Basile Starynkevitch <basile@starynkevitch.net>
Date:   Mon Feb 1 10:24:32 2021 +0100

    show gitid in -help message

M       main_BM.c
```

[1] The generated parts are clearly identified as such, and are extracted from the *Bismon* system.
[2] Obtained by the `git log -name-status -3` command running in *bismon* top source directory.

There is no notion of any identifiable "version" in *bismon*, so also in this report. The work is incremental and the development is agile.

copyright message

notice

This work is funded (from start of 2018 to end of 2020) thru the CHARIOT project (see its web site on `http://chariotproject.eu/`) which has received funding from the European Unions Horizon 2020 research and innovation programme under the Grant Agreement No 780075. This work is also *partly* funded -from 2019 to 2021- by the DECODER H2020 project, under its Grant Agreement 824231.

Contents

1 Introduction **5**

1.1 Mapping CHARIOT output 5
1.2 Deliverable Overview and Report Structure 5
1.3 Expected audience 6
1.4 The CHARIOT vision on specialized static source code analysis for more secure and safer IoT software development 8
1.4.1 About static source code analysis and IoT 8
1.4.2 The power of an existing compiler: GCC 11
1.4.3 Leveraging simple static source analysis on *GCC* 17
1.5 Lessons learned from *GCC MELT* 19
1.6 Driving principles for the *Bismon* persistent monitor 20
1.6.1 About *Bismon* 20
1.6.2 About *Bismon* as a domain-specific language 22
1.6.3 About *Bismon* as a evolving software system 24
1.6.4 About *Bismon* as a static source code analyzer framework 26
1.7 Multi-threaded and distributed aspects of *Bismon* 26

2 Data and its persistence in *Bismon* **29**

2.1 Data processed in *Bismon* 29
2.1.1 Immutable values 29
2.1.2 Mutable objects 30
2.2 garbage collection of values and objects 32
2.3 persistence in *Bismon* 33
2.3.1 file organization of the persistent state 34
2.3.2 persisting objects 35

3 Static analysis of source code in *Bismon* **37**

3.1 static analysis of *GCC* code 37
3.2 static analysis of IoT firmware or application code 37
3.3 static analysis related to pointers and addresses 38

4 Using *Bismon* **41**

4.1 How JSON is used by Bismon 41
4.1.1 The canonical *JSON* encoding of *Bismon* values 41
4.1.2 The nodal *JSON* decoding into *Bismon* values 42
4.1.3 *JSON* extraction with `extract_json` 43
4.2 Web interface internal design 45
4.3 Using `bismon` for CHARIOT 48

5 Miscellanous work **49**

5.1 Contributions to other free software projects 49
5.1.1 Aborted contribution to `libonion` 49
5.1.2 Contribution to *GCC* 49
5.2 Design and implementation of the compiler and linker extension 49

6 Conclusion **50**

A Building `bismon` from its source code **51**

A.1 Prerequisites for building `bismon` 51
A.2 File naming conventions in `bismon` 51
A.3 Naming conventions and source files organization for `bismon` 52
A.4 Generators and meta-programs in `bismon` 54

B Configuring `bismon` from its source code **54**

C Building `bismon` from its source code **54**
C.1 Checking the version of `bismon` 55

D Dumping and restoring the `bismon` persistent heap **57**

Index **58**

References **65**

List of Figures

1 the CHARIOT compilation of some IoT firmware or application **(simplified)** 11
2 The *Bismon* monitor used by some IoT developer team following the CHARIOT approach. . . 19
3 crude (soon deprecated) GTK interface oct. 22, 2018, git commit `cbdcf1ec351c3f2a` 25
4 generated dump example: `first_test_module` in file `store2.bmon` 35
5 syntax of values in dumped data files. 35

List of Tables

1 adherence to CHARIOT's GA deliverable and tasks descriptions 5
2 recursive inlining with constant folding in *GCC* (C source) 12
3 recursive inlining with constant folding in *GCC* (generated early Gimple) 12
4 recursive inlining with constant folding in *GCC* (generated SSA form) 13
5 recursive inlining with constant folding in *GCC* (generated optimized) 14
6 recursive inlining with constant folding in *GCC* (generated MIPS assembler) 14
7 optimization around heap allocation by *GCC* (C source) 15
8 optimization around heap allocation by *GCC* (generated Gimple) 16
9 optimization around heap allocation by *GCC* (generated SSA/optimized) 16
10 optimization around heap allocation by *GCC* (generated x86-64 assembly) 17
11 canonical JSON encoding $[\![v]\!]_{json}$ of a Bismon value v. 41
12 nodal JSON decoding $\langle js\rangle^{nod}$ of a JSON value js. 42
13 simple extraction from some JSON thing js; (see also table 14 below.) 43
14 complex extraction from some JSON thing js; (see also table 13 above.) 44

Glossary of terms and abbreviations used

Abbreviation / Term	Description
binutils	GNU free software package containing assembler `as`, linker `ld` and other utilities to operate on object files or binary executables, etc... https://www.gnu.org/software/binutils/
`bismon`	the free software framework and persistent monitor described here ; source repository on http://github.com/bstarynk/bismon/
CLANG	The Clang open-source project provides a language front-end and tooling infrastructure for languages in the C language family (C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript) for the LLVM project http://clang.llvm.org/
FRAMA-C	a free sofware extensible platform for analysis of C software http://frama-c.com/
FSF	Free Software Foundation http://fsf.org/
GCC	Gnu Compiler Collection http://gcc.gnu.org/
GCC MELT	was a (GPLv3+-licensed) GCC plugin and framework providing a DSL to ease GCC extensions ; it is archived on http://starynkevitch.net/Basile/gcc-melt/
Generic	language-independent abstract syntax tree (internal representation) in GCC
Gimple	middle-end internal representation in GCC
GPL	Gnu General Public Licence (a copylefted free software license) https://www.gnu.org/licenses/gpl.html
IoT	Internet of Things
`libonion`	an HTTP server library https://www.coralbits.com/libonion/
LLVM	The LLVM Project is an open-source collection of modular and reusable compiler and toolchain technologies http://www.llvm.org/
MELT	the Lisp-like domain specific language used in GCC MELT
Persistence	From Wikipedia : “In computer science, *persistence* refers to the characteristic of state that outlives the process that created it. This is achieved in practice by storing the state as data in computer data storage”. https://en.wikipedia.org/wiki/Persistence_(computer_science)
RTL	(register transfer language) back-end internal representation in GCC
static code analysis	(or static program analysis) “is the analysis of computer software that is performed without actually executing programs, in contrast with dynamic analysis, which is analysis performed on programs while they are executing.” (from Wikipedia: https://en.wikipedia.org/wiki/Static_program_analysis). In this D1.3[v1] report, it means static source code analysis, in practice analysis of C or C++ code for IoT fed to the *GCC* compiler.
SSA	Static Single Assignment (in GCC, a variant of Gimple)

1 Introduction

This D1.3 ^{v1} CHARIOT deliverable is a first **draft** -and *preliminary*- version of D1.3 ^{v2} that will be formally submitted as the complete and final deliverable at M30 on “Specialized Static Analysis tools for more secure and safer IoT software development”. This deliverable targets software engineering (and indirectly also software architects) experts working on IoT software coded in C or C++.

1.1 Mapping CHARIOT output

We refer to CHARIOT *Grant Agreement* (GA). See table 1 below.

Table 1: adherence to CHARIOT’s GA deliverable and tasks descriptions

CHARIOT GA component title	**CHARIOT GA component outline**	**respective document chapter[s]**	**justification**
	deliverable		
D1.3 Specialized Static Analysis tools for more secure and safer IoT software development.	The source code (top level documentation) of the prototype static analysis tools developed in task T1.3, including the definition of data formats and protocols, updates and adapation of existing libraries and software components, the persistent monitor outline and documentation, anf features description and documentation of the compiler and linker extensions. An initial version set (V1) will be compiled by M12 followed by a revised version set (v2) in M30.	this whole document	a single deliverable (with two versions of it, a preliminary draft one D1.3^{v1} and a final one D1.3^{v2}) describes the work. §1 is an overview and introduces the main concepts. §2 explains persistence. §3 relates to static analysis. §4 will become a user manual. §5 relates miscellanous work. The conclusion is in §6.
	tasks		
T1.3 Specialized Static Analysis tools for more secure and safer IoT software development.	**ST1.3.1** definition of data formats and protocols	§2; §5.2	The persistent monitor data and format are described in §2. The protocol to interact with CHARIOT’s blockchain is related to §5.2 and chapter 6 of D1.2
	ST1.3.2 significant patches to existing free software components	§5.1	Section §5.1 describe past work, and why future contributions to *GCC* could be needed.
	ST1.3.3 design and implementation of the persistent monitor	§1.4; §1.6; §1.7; §2;	§1.4 gives the CHARIOT vision of (informal) static analysis; §1.6 explains the driving principles of our *Bismon* persistent monitor, and (in §1.7) its multi-threaded and distributed aspects; §2 explains its persistent data.
	ST1.3.4 design and implementation of the compiler and linker extension	§3; §5.2	static analysis involves *generated* GCC plugins, as (in this D1.3^{v1} preliminary draft) partly explained in §3; compiler and linker extensions are (in D1.3^{v1}) drafted in §5.2

1.2 Deliverable Overview and Report Structure

This CHARIOT deliverable D1.3^{v1} is the *preliminary draft* of a report D1.2^{v2} scheduled at M30 on *Specialized Static Analysis tools for more secure and safer IoT software development* and relates to the work performed in

[3] Our favorite definition of *source code* is inspired by the FSF: the source code is the preferred form on which software developers should work. In practice, source code is what usually (but not always) should be managed by some version control system like a `git` code repository, or in some software forge.

T1.3 *Specialized Static Analysis tools for more secure and safer IoT software development*.

The introduction (this §1) describes the CHARIOT vision on static source code[3] (mostly of C and C++ code for IoT firmware and application) analysis (see §1.4), proposing a simple static analysis *framework* leveraging on the powerful recent *GCC* [cross-]compiler and explaining the necessity of persistence, then gives the driving principles of our *Bismon* persistent monitor (in §1.6); and explains its multi-threaded and distributed aspects (in §1.7). The data and its persistence is detailed (in §2, notably §2.1 for the processed data, §2.2 for the garbage collection, §2.3 for persistence). The §3 needs still to be mostly written and will describe (in the D1.3v2) how static analysis works. The §4 will contain the (mostly generated) user documentation. The §5 describes some miscellanous work.

Related previous CHARIOT deliverables include: D1.1 (on *Classification and use guidelines of relevant standards and platforms*), which provides a taxonomy of standards and guidelines (notably on cybersecurity, at a high and abstract level); but does mention much source code (except as open source projects such as IoTivity, FiWire, OM2M, etc). and D1.2 (on *Method for coupling preprogrammed private keys on IoT devices with a Blockchain system*) which describes the CHARIOT blockchain and its *Web API* (which should be adapted into functions or libraries callable from C code).

1.3 Expected audience

The numerous footnotes in this report are for a second reading (and may be used for forward references). **To understand this report** describing a circular and reflexive system, you should **read it twice** (skipping footnotes at the first read).

The reader of this document (within CHARIOT, a software engineering expert working on IoT software or firmware coded in C or C++) is expected to:

- be fluent in C (cf. Kernighan and Ritchie [1988]; Gustedt [2019]) and/or C++ (Stroustrup [2014, 2020]) programming (notably on Linux -see Mitchell et al. [2001]; Kerrisk [2010] and the `man7.org` and `kernelnewbies.org` and `kernel.org` websites- and/or for embedded products, perhaps for IoT),
- be knowing a bit the C11 standard (cf. ISO [2011a]; Memarian et al. [2016]) and/or the C++11 one (ISO [2011b]) and understanding well the essential notion of *undefined behavior* [4] in C or C++ programs,
- be a daily advanced user of Linux for software development activities using GCC and related developer tools (e.g. *binutils*, version control like `git`, build automation like `make` or `ninja`, source code editor like `emacs` or `vim`, the LaTeX text formatter [5]) on the *command line*.
- be *easily* able, in principle, to **compile** [6] his/her or other software coded in C (or in C++) **on the command line** (*without* any IDE - integrated software environment- or SDK - software development kit-) with a ***sequence* of `gcc` (or `g++`) commands** [7] **on Linux**.
- to be capable of building large free software or open source projects (such as the GCC compiler (cf GCC Community [2018] [8]), the Linux kernel, the QT or FLTK graphical toolkits and other open source projects of perhaps millions of source code lines) and smaller ones (e.g. `libonion` [9]) from their *source* form.
- have successfully downloaded and built the *Bismon monitor* from its source code available on **`https://github.com/bstarynk/bismon`**, on his/her Linux workstation.

[4] See **`http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html`** and **`https://blog.regehr.org/archives/1520`**

[5] See **`https://www.latex-project.org/`**. Some knowledge of LaTeX is useful to improve or contribute to this document.

[6] When compiling IoT software such as firmware, it usually is of course some *cross*-compilation.

[7] In practice, we all use some *build automation* tool, such as `make`, `ninja` or generators for them such as `cmake`, *autoconf*, `meson`, etc... But the reader is expected to be able to configure that, e.g. to add more options to `gcc` or to `g++` (perhaps in his/her `Makefile`) and is able to think in terms of a sequence of elementary `gcc` or `g++` compilation commands (or, when using Clang, `clang` or `clang++` commands).

[8] See **`http://gcc.gnu.org`** and notice that many cross-compiler forms of *GCC* may need to be compiled from the source code of that compiler distributed by the *FSF*, in particular because GCC plugin ability is needed within CHARIOT, or because hardware vendors provide only old versions of that compiler.

[9] see **`https://coralbits.com/libonion/`**

- have contributed or participated to some free software or open source projects and understanding their social (cf Raymond [2001]) and economical (cf Weber [2004]; Tirole [2018]; Nagle [2018]; Di Cosmo and Nora [1998]; Lerner and Tirole [2000]) implications, the practical work flow, the importance of developer communities and of business [10] support.
- be interested in static source code analysis, so have already tried some such tools like *Frama-C* [11] (cf. Cuoq et al. [2012]), *Clang analyzer* [12], ..., and be aware of compiler concepts and technologies (read Aho et al. [2006]).
- be interested by knowledge base tools and symbolic artificial intelligence approaches (cf Nouira and Fouet [1996]; Pitrat [1990, 1996]; Polito et al. [2014]; Raj [2018]; Rodriguez et al. [2018]; Doyle [1985]; Starynkevitch [1990]; Khalilian et al. [2021] and REFPERSYS) to software engineeering problems (see Rich and Waters [2014]; Beckert et al. [2007]; Happel and Seedorf [2006]; Rus et al. [2002]; Baudin et al. [2002]; Guilbaud et al. [2001]; Starynkevitch [2007, 2009]; Fitz et al. [2019]; Gabbay and Smets [2013]).
- be familiar with operating systems principles (see Tanenbaum [1992]; Arpaci-Dusseau and Arpaci-Dusseau [2015]) and well versed in Linux programming (cf. Mitchell et al. [2001]; Kerrisk [2010] [13]).
- be interested in various programming languages (cf. Abelson et al. [1996]; Scott [2007]; Queinnec [1996]) and their implementation[14] including domain specific ones.
- is aware that *most software projects fail* [15] (for *some* definition of failure; see also Brooks [1995]; Khan et al. [2019]; Attarzadeh and Siew Hock [2008], etc...), and that obviously includes research software projects, which fail even more often, and any IoT software in general. I believe that such a high failure rate is *intrinsic* [16] to any non-trivial software developed by humans (because of Braun et al. [1956], of "leaky abstractions" [17] and of the *Halting problem*, etc...), and that formal methods approaches are still vulnerable to specification [18] bugs. Agile and lean approaches could be effective for improving IoT software development processes (see Rodriguez et al. [2018]). Code review by senior programmers is needed.
- is understanding the notion of *Technical Readiness Level* (TRL) and its implication in innovative projects, notably H2020 ones (see Héder [2017]).
- is familiar with the idea of generating documentation from software source code, either thru literate programming techniques, such as the `nuweb` system combined with LaTeX, or with dedicated documentation generating tools such as `doxygen` documentation generator or `ocamldoc` documentation generator. Documentation framework tools like `pandoc` document converter, small batch document formatters like LOUT software or schemas such as DOCBOOK document schama are also relevant.

---

[10] See also Daniel Oberhous' blog February 2019 post on https://motherboard.vice.com/en_us/article/43zak3/the-internet-was-built-on-the-free-labor-of-open-source-developers-is-that-sustainable: *The Internet Was Built on the Free Labor of Open Source Developers. Is That Sustainable?*

[11] See **http://frama-c.com/**

[12] See **https://clang-analyzer.llvm.org/**

[13] look into `man` pages on **http://man7.org/linux/man-pages/**

[14] See also **https://www.tweag.io/blog**, since several posts there are relevant to ideas inspiring *Bismon*.

[15] See **https://www.geneca.com/why-up-to-75-of-software-projects-will-fail/**

[16] IMHO, allocation of much more time and efforts, including code reviews, on software development is necessary - but sadly it is not sufficient - to lower that failure rate. Read about the *Joel Test* on **https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/** for more.

[17] Cf. Spolsky's *Law of leaky abstractions* on **https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/** etc... for more.

[18] A speculative example of a tragic specification bug might include something inside the Boeing 737 MAX - see **https://en.wikipedia.org/wiki/Boeing_737_MAX** - which could have recent crashes related to bugs in specifications, and probably developed with the most serious formal methods approaches, dictated by DO-178-C -see **https://en.wikipedia.org/wiki/DO-**

1.4 The CHARIOT vision on specialized static source code analysis for more secure and safer IoT software development

1.4.1 About static source code analysis and IoT

There are many existing documents related to improving safety and security in IoT software (e.g. Chen and Helal [2011]; Medwed [2016]; Kumar and Goyal [2019]; Chakkaravarthy et al. [2019]), and even more on static source code analysis in general (cf. Gomes et al. [2009]; Goseva-Popstojanova and Perhinschi [2015]; Binkley [2007] and many others).

Several conferences are dedicated to static analysis[19]. All dominant C compilers (notably GCC and Clang, but also MicroSoft's *Visual C*™) are using complex static source code analysis techniques for optimizations and warnings purposes (and that is why C compilers are monsters [20]). It is wisely noticed (in Goseva-Popstojanova and Perhinschi [2015]) that **state-of-the-art static source code analysis tools are *not* very effective in detecting security vulnerabilities** [21], so they are not a "silver bullet" (Brooks [1987]). Many taxonomies of software defects already exist (e.g. Silva and Vieira [2016]; Wagner [2008]; Levine [2009] etc....), notably for IoT (see Carpent et al. [2018]; Ahmad et al. [2018]; László et al. [2017]); however the relation between an explicit defect and a source code property is generally fuzzy, or ill-defined.

Observe that, while Debian and several other Linux distributions are packaging many thousands of C or C++ programs and libraries and several free software source code analyzers (notably FRAMA-C and CLANG tools and COCCINNELLE), very few Debian packages -coded in C or C++ for example- are conditionnally "build-depending"[22] on them. This could be explained by the practical difficulties, for Debian developers or packagers, to effectively use these source code analyzers. Large C or C++ Linux-related software - such as the Linux kernel, standard or widely used C or C++ libraries (including `libcurl`, QT, GTK, LIBZ, OPENSSL, etc...), Firefox, Libreoffice, various interpreters (Python, Guile), runtime (Ocaml's or SBCL's ones) or compilers (GCC, Clang) etc... are still not really analyzable automatically today, in a cost-effective and time-efficient manner. But Debian cares a lot about software quality and stability, so when automatic tools are available and *practically* useful, they *are* used! The same observation holds even for specialized Linux distributions[23] used in many non-critical embedded and connected IoT systems.

Language specifications and their implementations allow weird behavior; for example, simply testing in *C++* or *C* code an unitialized 'bool' variable may crash the entire program (as explained here), even if implementations doing so are not common (but definitely could happen in the IoT world with DSP embedded processors with VLIW architecture), related to so called *trap representations* permitted by language standards (see ISO [2011b,a]). Also, some implementations of C have `sizeof(char)`, `sizeof(int)`, `sizeof(long)`, `sizeof(void*)` and `alignof(char)`, `alignof(long)` being one (e.g. on word-addressable but not byte-addressable Harvard architectures). . The common Intel x86 IA-32 bits processor architecture (still used in VORTEX86 chips) allow -with some runtime performance penalty- access to misaligned 64 bits floating point numbers, but the same C code would crash on e.g. ARM systems, such as RASPBERRYPI boards. Of course endianness matters a lot in any IoT devices and is very important in binary communication protocols (e.g. SATA , ETHERNET or IEEE 802.11 WIFI, LI-FI (see Khandal and Jain [2014]) or USB or in the automotive industry the CAN bus. It also matters in network protocol stacks, including IP (see Cerf and Icahn [2005]) or ASN-1 (see Barry [1992]), used in cryptographic certificates (such as X.509 for HTTPS secure web servers. The recent HTTP/2 web protocol is a binary one and requires to care about endianness also. X Window System protocols, WAYLAND display protocols SSH, RFB protocol are also binary, with endianness issues, and extremely popular in non-critical IoT systems based on Linux. For multimedia connected consumer devices, HDMI on the hardware side, and MPEG-4 for audio/video encoding are also binary protocols or encodings. The OGG container format is free of patent and should be prefered for audio encoding. The many JPEG digital image formats, very

178C etc... But in mid-2019 this is only a speculation (details are unknown to me). See also the controversial but interesting analysis of Graeber and Cerutti [2018] explaining the plausibility of such speculations.

[19] The 25th Static Analysis Symposium happened in august 2018, see **http://staticanalysis.org/sas2018/sas2018.html**; most ACM SIGPLAN conferences such as POPL, PLDI, ICFP, OOPSLA, LCTES, SPLASH, DSL, CGO, SLE... have papers related to static source code analysis.

[20] see **https://softwareengineering.stackexchange.com/a/273711/40065** for more.

[21] Se we can only hope an *incremental* progress in that area. Static source code analysis in CHARIOT won't make miracles.

[22] The Debian packaging system is sophisticated enough to just *suggest* a tool useful to *build* a package from its source code.

[23] The Raspbian distribution is a typical example, see **https://raspbian.org/** for more. But look also into **https://www.automotivelinux.org/** or **https://www.genivi.org/** as another examples.

common in digital consumer photography, needs also endian-sensitive binary processing, and is increasingly used in most IoT connected cameras (presumably Linux based), some of which are still designed in Europe.

Static source code analysis tools can -generally speaking- be [24] viewed as being of one of two kinds:

- strongly formal methods based, semantically oriented, "sound" tools (e.g. built above abstract interpretation -cf. Cousot and Cousot [2014, 1977]-, model checking -cf. Schlich [2010]; Siddiqui et al. [2018] and Jhala and Majumdar [2009]-, and other formal techniques on the whole program... See also Andreasen et al. [2017]) which can give excellent results but require a lot of expertise to be used, and may take a long time to run [25]. For examples, *Frama-C* (cf Cuoq et al. [2012]), *Astrée* (cf Miné and Delmas [2015]), *Mopsa* (cf Miné et al. [2018]) etc... The expert user needs either to explicitly annotate the analyzed source code (e.g. in ACSL for *Frama-C*, see Baudin et al. [2018]; Delahaye et al. [2013]; Amin and Rompf [2017]), and/or cleverly tune the many configuration knobs of the static analyzer, and often both. Often, the static analyzer itself has to be extended to be able to give interesting results on one particular analyzed source code [26], when that analyzed code is complex or quite large. Many formal static analyzers (e.g. Greenaway et al. [2014]; Vedala and Kumar [2012]) focus on checking just *one* aspect or property of security or safety. Usually, formal and sound static analyzers can practically cope only with small sized analyzed programs of at most one or a few hundred thousands lines of C code (following some *particular* coding style or using some definable *subset* of the C language[27]). Some formal analysis approaches include a definition of a strict subset of *C*, thru perhaps some automatically generated code (cf. Bhargavan et al. [2017]) from some DSL. In practice, the formal sound static analyzers are able to prove *automatically* some *simple* properties of small, highly critical, software components (e.g. avoiding the need of *unit testing* at the expense of very *costly software development efforts*).
- lightweight "syntax" oriented "unsound" tools, such as Coverity Scan [28] or Clang-Analyzer, or even recent compilers (GCC or Clang) with all warnings and link-time optimization [29] enabled. Of course, these simpler approaches give many false positive warnings (cf Nadeem et al. [2012]), but **machine learning** techniques (cf Perl et al. [2015]; Flach [2012]; Shalev-Shwartz and Ben-David [2014]) using bug databases could help.

A generalization of strictly static source code analysis enables a mixed, semi-static and semi-dynamic approach, but leverages on extending some existing *compilers* (such as `gcc` or `clang`...), linkers (e.g. `ld` started by `g++`), or even run-time loaders (e.g. `ld-linux.so` or *crt0*): inserting some runtime checking code into the compiled executable, during compilation time or at link time[30]. This is the design idea of widely used tools such as the `valgrind` memory checker[31] or the address sanitizer (see Serebryany et al. [2012]) originally in Clang, and now also in *GCC*. Today, both `gcc` and `clang` have several *compiler sanitizers* with such an approach. Some of them are very intrusive because they slow down the debugged program run time by an important factor of at least 10x, others are almost imperceptible, since they may increase memory consumption by perhaps 25% at runtime, but CPU time by just a few percents. Some specialized semi-static source code analyzers also adopt a similiar instrumenting approach (for example, Biswas et al. [2017]). Most

[24] This is a gross simplification! In practice, there is a continuous spectrum of source code analyzers, much like there is a spectrum between compilers and interpreters (with e.g. bytecode or JIT implementations sitting in between compilation and naive interpretation).

[25] There are cases where those static analyzers need weeks of computer time to give interesting results.

[26] The *Astrée* project can be seen as the development of a powerful analyzer tool *specifically* suited for the needs of Airbus control command software; it implements many complex abstract interpretation lattices wisely chosen to fit the relevant traits of the analyzed code. Neither *Astrée* nor *Frama-C* can easily -without any additional tuning or annotations- and successfully be used on most Linux command line utilities (e.g. `bash`, *`coreutils`*, *`binutils`*, `gcc`, ...) or servers (e.g. `systemd`, `lighttpd`, `Wayland` or `Xorg`, or IoT frameworks such as MQTT...). But *Frama-C* can be extended by additional plugins so is a *framework* for sound static analysis.

[27] For instance, both *Frama-C* and *Astrée* have issues in dealing with standard dynamic C memory allocation above `malloc`; since they target above all the safety critical real-time embedded software market where such allocations are *forbidden*.

[28] See **https://scan.coverity.com/**

[29] Link-time optimization (e.g. compiling and *linking* with `gcc -O2 -flto -Wall -Wextra` using GCC) slows down the build time by more than a factor of two since the intermediate internal representation (IR) of the compiler (e.g. Gimple for *GCC*, see GCC Community [2018] §12) is kept in object files and reload at "link-time" which is done by the *compiler* working on the whole program's IR, so is rarely used.

[30] That could even be at dynamic-link time, e.g. in `ld-linux.so` just before running `main` in some C or C++ program.

[31] See **http://valgrind.org/**, and be aware that `valgrind` is capable of runtime checking many other properties, such as *some*

sanitizers[32] are whole-program[33] tools and need some operating system kernel at runtime, but they provide yet another effective tool to embedded software developers.

Most (fully) static[34] source code analyzers require some kind of interaction with their user (cf Lipford et al. [2014]), in particular to present partial analysis results and explanations about them, or complex information like control flow graphs, derived properties at each statements.

The VESSEDIA project is an H2020 IoT-related project [35] which is focusing on a strong formal methods approach for IoT *software* and insists on a "single system formal verification" approach (so it makes quite weak hypothesis on the "systems of systems" view); it "aims at enhancing safety and security of information and communication technology (ICT) and especially the Internet of Things (IoT). More precisely the aim of [the VESSEDIA] project consists in making formal methods more accessible for application domains that want to improve the security and reliability of their software applications by means of Formal Methods" [36]. Most VESSEDIA partners (even the industrial ones, cf. Berkes et al. [2018]) are versed in formal static analysis techniques (many of them being already trained to use *Frama-C* several years before, and several of them contributing actively to that tool.). Some of the major achievements of VESSEDIA includes formal (but *fully automatic*) proofs of often *simple* (and sometimes very complex and very specific) properties of some basic software components (e.g. lack of undefined behavior in the memory allocator, or the linked list implementation, of Contiki). Some automatically proven properties can be very complex, and the very hard work [37] is in formalizing these properties (in terms of C code!) and then in assisting the formal tool to prove them.

In contrast, CHARIOT focuses mainly on a *systems of systems* (e.g. networks of systems and systems of networks) approach, so [38] "aims to address how safety-critical-systems should be securely and appropriately managed and integrated with a fog network made up of heterogeneous IoT devices and gateways.". Within CHARIOT, static analysis methods have to be "simple" and support its *Open IoT Cloud Platform* thru its *IoT Privacy, Security and Safety Supervision Engine* [39], and some industrial CHARIOT partners, while being IoT network and hardware experts, are noticing that their favorite IDE (provided by their main IoT hardware vendor) is running some GCC under the hoods during the build of their firmware, but are not used to static source code analysis tools. The CHARIOT approach to static source code analysis does *not* require the same level of expertise as needed for the *Verified in Europe* label pushed by the VESSEDIA project.

Non-critical [40], but communicating, industrial IoT is programmed in various languages[41]: when a non-critical equipement should be autonomous so needs to consume very little energy, programming it in C, C++ or Rust is preferable (think of smart watches and similar wearables increasing worker productivity). When such a non-critical computing device is part of some larger equipement requiring constant and significant electric power (automatic expressway tollgate, some smart sensors in an oil refinery, high-end digital oscilloscopes, office air-conditioning facilities, etc...) it could make sense to code it in a higher-level language, such as C++, Java, Python, Lua, JavaScript . . . making the embedded software developer more productive when producing

race conditions or other *undefined behavior*.

[32] Since our *Bismon* framework is becoming capable of customizing *GCC* by generating ad-hoc plugins, it could be later used in such a way too, to easily develop ad-hoc and *project-specific* compiler sanitizers.

[33] That "whole-program" holistic aspect is shared by most static source code analyzers, such as *Clang-analyzer* or *Frama-C* and is of course the major selling point of our *Bismon* framework.

[34] Some compiler sanitizers, i.e. semi-static analyzers, may also require user interaction, but that is generally done thru ad-hoc source code annotations such as `#pragma`-s.

[35] The VESSEDIA project (Verification Engineering of Safety and Security Critical Industrial Applications) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 731453, within the call H2020-DS-LEIT-2016, started in january 2017, ending in december 2019.

[36] Taken from **https://vessedia.eu/about**

[37] In terms of human efforts, the formalization of the problem, and the annotation of the code and guidance of the prover, takes much more time and money (often more than a factor of 30x) than the coding of the C code. Within CHARIOT no large effort is explicitly allocated for such concrete but difficult tasks.

[38] Taken in october 2018 from **https://www.chariotproject.eu/About**, *§Technical Approach*.

[39] Taken from **https://www.chariotproject.eu/About/**

[40] In this report, a critical industrial software is a piece of embedded code whose *direct* failure impacts *human life* (e.g. train braking system, but not its preventive maintenance), or *costly industrial installations* (e.g. entire oil platforms, large power plants) involving large equipement costing many dozens of M; anything else is non-critical : air conditioning of a office building, measurement of brake wear on a train for preventive maintenance optimization, door opening automation in an airport which can be quickly disabled in emergency cases, etc...

[41] See **https://www.iotforall.com/2018-top-3-programming-languages-iot-development/** for more.

code for industrial systems built in small series.

The **CHARIOT approach** to static source analysis **leverages on** an ***existing*** *recent* **GCC cross-compiler** [42] so focuses on GCC-compiled languages[43]. Hence, the IoT software developer following the CHARIOT methodology would just add some additional flags to *existing* `gcc` or `g++` cross-compilation commands, and needs simply to change slightly his/her build automation scripts (e.g. add a few lines to his `Makefile`). Such a gentle approach (see figure 1) has the advantage of not disturbing much the usual developer workflow and habit, and addresses also the *junior* IoT software developer. Of course the *details* of compilation commands would change, the commands shown in the figure 1 are grossly simplified! The compilation and linking processes are communicating -via some additional *GCC* plugins (cf. GCC Community [2018] §24) doing inter-process communication- with our *persistent monitor*, tentatively called `bismon` . It is preferable (see Free Software Foundation [2009]) to use free software GCC plugins (or free software generators for them) when compiling proprietary firmware with the help of these plugins; otherwise, there might be [44] some licensing issues on the obtained proprietary binary firmware blob, if it was compiled with the help of some hypothetical *proprietary* GCC plugin. The insight driving the `bismon` tool is the hope to provide small IoT software development teams with a semi-automatic assistant working in parallel with the developers and improving communication between team members, later enabling some limited software features extraction (see Asanovic et al. [2006]; Brooks [1987, 1995]; Zuboff [2015]; Pérez et al. [2020]).

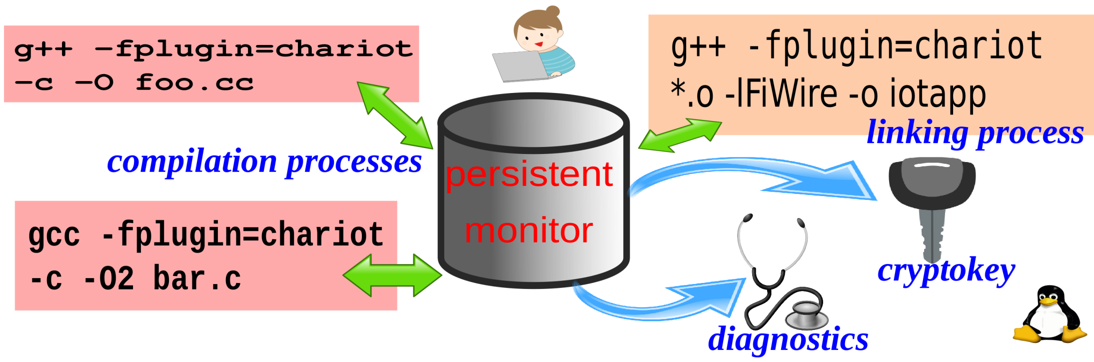

Figure 1: the CHARIOT compilation of some IoT firmware or application **(simplified)**

1.4.2 The power of an existing compiler: GCC

CHARIOT static analysis tools will leverage on the mainstream GCC [45] compiler (generally used as a *cross-compiler* for IoT firmware development) Current versions of *GCC* (that is, GCC 8.2 as of September 2018) are capable of quite surprising optimizations (internally based upon some sophisticated static analysis techniques and advanced heuristics). But to provide such clever optimizations, the ***GCC* compiler has to be quite a large software, of more than 5.28 millions lines** [46] of source code (in `gcc-8.2.0`, measured by `sloccount`). This figure is an under-estimation [47], since *GCC* contains a *dozen of domain specific languages* and their transpilers to generated C++ code, which are not well recognized or measured by `sloccount`.

We show below several examples of optimizations done by recent *GCC* compilers. Usually, these optimizations happen in the middle-end and work on internal intermediate representations which are mostly not [48]

[42] The actual version and the concrete configuation of *GCC* are important; we want to stick -when reasonably possible- to the latest GCC releases, e.g. to GCC 10 in summer 2020. In the usual case, that *GCC* is a cross-compiler. In the rare case where the IoT system runs on an x86-64 device under Linux, that *GCC* is not a cross-, but a straight compiler.

[43] The 2019 Gnu Compiler Collection is able to compile code written in C, C++, Objective-C, Fortran, Ada, Go, and/or D.

[44] Of course, I -Basile Starynkevitch- am not a lawyer, and you should check any potential licensing issues with your own lawyer.

[45] "Gnu Compiler Collection". See **http://gcc.gnu.org/** for more. In practice, it is useful to build a recent GCC cross-compiler, fitted for your IoT system, from its published source code - see **https://gcc.gnu.org/mirrors.html** for a list of mirrors.

[46] Measured by David Wheeler's `sloccount` utility

[47] The Unix `wc` utility gives 14.6 millions lines, including empty ones but excluding generated C++ code, in 498 megabytes.

[48] Most of the internal *GCC* representations -e.g. *Gimple* or *SSA*- are common to all target systems; however, some constants, like the size and alignment of primitive data types such as `long` or pointers, are known at preprocessing phase or at early *Gimplification* phase.

target specific.

recursive inlining with constant folding

Table 2: recursive inlining with constant folding in *GCC* (C source)

```c
// file factinline12.c
static int fact (int n) {
  if (n <= 1)
    return 1;
  else
    return n * fact (n - 1);
}

int fact12 (void) {
  return fact (12);
}
```

C source code

The table 2 shows a simple example of C code (file `factinline12.c`). After preprocessing and parsing, it becomes quickly expanded in some *Gimple* representation (cf. §12 of GCC Community [2018]), whose elementary instructions are arithmetic with two operands, or simple tests with jumps, or simple calls (in so called *A-normal form* , where nested calls like `a=f(g(x),y);` get transformed into a sequence introducing a temporary τ such as τ`=g(x)` then `a=f(`τ`,y)`, etc...), shown in table 3.

Table 3: recursive inlining with constant folding in *GCC* (generated early Gimple)

```
fact12 ()
{
  int D.1420;
  D.1420 = fact (12);
  return D.1420;
}

fact (int n)
{
  int D.1424;
  if (n <= 1) goto <D.1422>; else goto <D.1423>;
  <D.1422>:
  D.1424 = 1;
  // predicted unlikely by early return (on trees) predictor.
  return D.1424;
  <D.1423>:
  _1 = n + -1;
  _2 = fact (_1);
  D.1424 = n * _2;
  // predicted unlikely by early return (on trees) predictor.
  return D.1424;
}

// factinline12.c.005t.gimple generated by ...
//... /usr/bin/mipsel-linux-gnu-gcc-10 -O3 -S -fverbose-asm \
//...  -fdump-tree-gimple  -fdump-tree-ssa -fdump-tree-optimized factinline12.c
```

Gimple code

This is a textual (and quite *incomplete* since a partial view of some) dump of some in-memory *internal intermediate representation* during compilation. What really matters to the CHARIOT static source code analyzer framework is the data inside the compilation process `cc1`, not its partial textual dump [49]. The

[49] It is possible to pass the `-fdump-tree-all` flag to `gcc`; then hundreds of intermediate textual dump files are emit-

Gimple internal in-memory representation is declared inside several source files of *GCC*, including its `gcc-8*/gcc/gimple.def`, `gcc-8*/gcc/gimple.h`, `gcc-8*/gcc/gimple-iterator.h`, etc...

After gimplification, many other optimizations happen. **The *GCC* compiler runs more than two hundred optimization passes !**. The table 4 shows the "static single assignment" form (SSA, see Pop [2006] and GCC Community [2018] §13), where variables are duplicated so that each SSA variable gets assigned only *once*. The control flow is reduced to *basic blocks* (with a single entry point at start, and perhaps several conditional exit edges). Then special ϕ nodes introduce places where such a variable may come from two other ones (after branches).

Table 4: recursive inlining with constant folding in *GCC* (generated SSA form)

```
;; Function fact (fact, funcdef_no=0, decl_uid=1414, cgraph_uid=1, symbol_order=0)
fact (int n)
{
  int _1;
  int _2;
  int _3;
  int _8;
  int _9;
  <bb 2> :
  if (n_5(D) <= 1)
    goto <bb 3>; [INV]
  else
    goto <bb 4>; [INV]
  <bb 3> :
  _9 = 1;
  // predicted unlikely by early return (on trees) predictor.
  goto <bb 5>; [INV]
  <bb 4> :
  _1 = n_5(D) + -1;
  _2 = fact (_1);
  _8 = n_5(D) * _2;
  // predicted unlikely by early return (on trees) predictor.
  <bb 5> :
  # _3 = PHI <_9(3), _8(4)>
  return _3;
}

;; Function fact12 (fact12, funcdef_no=1, decl_uid=1417, cgraph_uid=2, symbol_order=1)
fact12 ()
{
  int _3;
  <bb 2> :
  _3 = fact (12);
  return _3;
}
```

Static Single Assignment (SSA) code

At last, many other optimizations happen. And the optimized form in table 5 shows that `fact12` just returns the constant 479001600 (which happens to be the result of `fact(12)` computed *at compile-time*).

ted, including `factinline12.c.004t.gimple` and `factinline12.c.020t.ssa` and many others for the compilation of `factinline12.c` source file.

Table 5: recursive inlining with constant folding in *GCC* (generated optimized)

```
;; Function fact12 (fact12, funcdef_no=1, decl_uid=1417, cgraph_uid=2, symbol_order=1)
fact12 ()
{
  <bb 2> [local count: 118111601]:
  return 479001600;
}
```

optimized form

Table 6: recursive inlining with constant folding in *GCC* (generated MIPS assembler)

```
### skipped 72 lines in factinline12.s

.text
.align 2
.globl fact12
.set nomips16
.set nomicromips
.ent fact12
.type fact12, @function
fact12:
.frame $sp,0,$31 # vars= 0, regs= 0/0, args= 0, gp= 0
.mask 0x00000000,0
.fmask 0x00000000,0
.set noreorder
.set nomacro
 # factinline12.c:10:   return fact (12);
li $2,478937088 # 0x1c8c0000  # tmp196,
 # factinline12.c:11: }
jr $31  #
ori $2,$2,0xfc00  #, tmp196,

.set macro
.set reorder
.end fact12
.size fact12, .-fact12
.ident "GCC: (Ubuntu 10.2.0-5ubuntu1~20.04) 10.2.0"
.section .note.GNU-stack,"",@progbits
```

MIPS assembler

Finally, the generated assembler code has no trace of `fact` function, and contains just what is shown in table 6 (where many useless comment lines, giving the detailed configuration of the cross compiler, have been removed).

heap allocation optimization

Our second example shows that *GCC* is capable of clever optimizations around dynamic heap allocation and de-allocation. Its source code in file `mallfree.c` is shown in table 7.

Table 7: optimization around heap allocation by *GCC* (C source)

```
/// file mallfree.c
#include <stdlib.h>
int weirdsum(int x, int y) {
  int*ar2 = malloc(2*sizeof(int));
  ar2[0] = x;
  ar2[1] = y;
  int r = ar2[0] + ar2[1];
  free (ar2);
  return r;
}
```

C source code

The straight *GCC* compiler [50] (on Linux/x86-64) is optimizing and able to remove the calls to `malloc` and to `free`, following the *as-if rule*.

The *Gimple* form shown in table 8. Pointer arithmetic has been expanded to target-specific *address* arithmetic in byte units, knowing that `sizeof(int)` is 4.

[50] Similar optimizations also happen with a GCC MIPS targetted cross-compiler.

Table 8: optimization around heap allocation by *GCC* (generated Gimple)

```
weirdsum (int x, int y)
{
  int D.2639;
  int * ar2;
  int r;

  ar2 = malloc (8);
  *ar2 = x;
  _1 = ar2 + 4;
  *_1 = y;
  _2 = *ar2;
  _3 = ar2 + 4;
  _4 = *_3;
  r = _2 + _4;
  free (ar2);
 D.2639 = r;
  return D.2639;
}


// mallfree.c.005t.gimple generated by ...
//... -O3 -S -fverbose-asm \
//... -fdump-tree-gimple -fdump-tree-ssa -fdump-tree-optimized mallfree.c
```

Gimple code

The *SSA/optimized* form appears in table 9. It shows that the call to `malloc` and to `free` have been optimized away, so the `weirdsum` function don't use heap allocation anymore.

Table 9: optimization around heap allocation by *GCC* (generated SSA/optimized)

```
;; Function weirdsum (weirdsum, funcdef_no=16,
;;;   decl_uid=2634, cgraph_uid=17, symbol_order=16)

weirdsum (int x, int y)
{
  int r;

  <bb 2> [local count: 1073741824]:
  r_4 = x_2(D) + y_3(D);
  return r_4;

}
```

SSA/optimized code

So the generated x86-64 assembler code in table 10 contain no calls to `malloc` or `free`, hence contains the same code that would be generated from just `int weirdsum(int x, int y) {return x+y;}`.

This `mallfree.c` example could look artificial (because human developers won't *directly* code this way). However, a similar example might happen in real life after preprocessor expansion and inlining in large header-mostly libraries. Also, equivalent code happens (perhaps after some inline expansion done by any optimizing compiler) with machine generated C code (e.g. by tools like ANTLR or BISON parser generators, in some JSON libraries or JSONRPC services) In addition, most genuine C++11 code (e.g. using standard container templates from `<map>` or `<vector>` standard headers) would produce conceptually similar code (since many standard constructors and destructors would call internally the standard `::operator new` and `::operator delete` operations, which get inlined into calling the system `malloc` and `free` functions).

Table 10: optimization around heap allocation by *GCC* (generated x86-64 assembly)

```
## 64 lines removed
        .text
        .p2align 4
        .globl        weirdsum
        .type        weirdsum, @function
weirdsum:
.LFB16:
        .cfi_startproc
        endbr64
# mallfree.c:7:   int r = ar2[0] + ar2[1];
        leal        (%rdi,%rsi), %eax        #, r
# mallfree.c:10: }
        ret
        .cfi_endproc
.LFE16:
        .size        weirdsum, .-weirdsum
        .ident        "GCC: (Ubuntu 10.2.0-5ubuntu1~20.04) 10.2.0"
        .section        .note.GNU-stack,"",@progbits
        .section        .note.gnu.property,"a"
        .align 8
        .long        1f - 0f
        .long        4f - 1f
        .long        5
0:
        .string        "GNU"
1:
        .align 8
        .long        0xc0000002
        .long        3f - 2f
2:
        .long        0x3
3:
        .align 8
4:
```

x86-64 assembly

1.4.3 Leveraging simple static source analysis on *GCC*

By hooking through CHARIOT specific GCC *plugins* [51] into usual [cross-] *compilation* processes (some `gcc` or `g++`, etc... such as a `mips-linux-gnu-gcc-8` or a `arm-linux-gnueabi-g++-8`, etc...), IoT software developers will be able to take advantage of all the numerous optimizations and processing done by *GCC*. However, a typical firmware build would take many dozens of such compilation processes, since every translation unit (practically `*.c` C source files and `*.cc` C++ source files of the IoT firmware) requires its compilation process [52]. In practice, IoT software developers would use some *existing* **build automation** tool (such as `make`, `ninja`, `meson`, `cmake` etc...) which is running suitable compilation commands. They would need to adapt [53] and configure their build process (e.g. by editing their `Makefile`-s, etc...), notably to fetch their *GCC* plugin C++ code and compile it into some `*.so` shared object to be later `dlopen`-ed by some cross-compiler `cc1` process, and to use these plugins in their cross-compilation of their IoT firmware. By working in cooperation with existing *GCC* compilation tools, the IoT developer don't have to change much his/her current development practices. However, these various compilation processes are producing partial new (or updated) internal representations, and need to know about other translation units. So some **persistence is needed** to keep some data (such as the control flow graph, etc...) related to past [54] compilation processes during perhaps the entire project development work.

[51] Notice that GCC plugins work mostly on Linux -but not really on Windows-, so this explains the CHARIOT requirement of [cross-]compiling IoT firmware on a Linux workstation.

[52] Technically, a C compilation process would be running some `cc1` internal executable started by some `*gcc*` [cross-] compilation command.

[53] How to adapt cleverly their `Makefile` to take advantage of CHARIOT provided static analysis is of course the responsability of the IoT developer. Surely several extra lines are needed, and the `CFLAGS=` line of their `Makefile` should be changed. For other builders such as `ninja`, etc..., some similar configuration changes would be needed.

[54] Notice that recent *GCC* provide a link-time optimization (LTO) ability, if the developer compiles *and links* with e.g. `gcc -O2 -flto`. But LTO is not widely used since it slows down a lot the building time, and the plugin infrastructure of GCC is not very

IoT developers need to interact with their static source code analysis (*GCC* based) tool. In particular, they might need to understand more the optimizations done by their (CHARIOT augmented, thru *GCC* plugins) compiler, and they also need to be able to query some of the numerous intermediate data related to that static analysis and compilation. Practically, a small team of IoT developers working on the same IoT firmware project would interact with the static analysis infrastructure, that is with a single *persistent monitor* process. That **persistent monitor** (*Bismon*) would be some "server-like" program, started probably every working day in the morning and loading its previous state, and gently stopped every evening and dumping its current state on disk. It needs to keep its persistent state in some files. For convenience, a textual format is preferable [55]. These persistent store files could (and actually should) be regularly backed up and kept in some version control system.

Since a single *Bismon* process is used by a small team of IoT developers, it provides some web interface: each IoT developer will interact with the persistent monitor through his/her web browser [56]. In addition, a static analysis expert (which could perhaps be the very senior IoT developer of the team) will configure the static analysis (also through a web interface).

The figure 2 gives an overall picture: on the top, both Alice and Bill are working on the same IoT project source code (and each have a slightly different version of that code, since Alice might work on routine `foo_start` in file `foo.cc`, while Bill is coding the routine `bar_event_loop` in file `bar.c`). Both Alice and Bill (IoT developers in the same team, working on the same IoT firmware) are compiling with the same *GCC* cross-compiler (the GCC egg) enhanced by a plugin (the small green puzzle piece, at right of GCC eggs). They use their favorite editor or IDE to work on the IoT source code, and run from time to time a builder (e.g. `make`). They use a browser (with a rainbow in the figure) to interact with the monitor and query static analysis data. The purple dashed arrows represent HTTP exchanges between their browser and the monitor. The compilation processes, extended by the GCC plugin, communicate with the monitor (thru the gray dashed arrows). A static analysis expert (the "geek", at left of the Bismon monitor) is configuring the monitor thru his own web interface. The monitor is *generating* (orange arrow) the C++ code for GCC plugins (small blue hexagon at right), and the IoT developer needs to change his/her build procedures to compile and use that generated GCC plugin. *Bismon* also uses meta-programming techniques to emit (curved blue arrow to left) internal code (bubble) - notably C code dynamically loaded by the monitor and JavaScript/HTML used in browsers. The several "Tux" penguins remind that all this (cross-compilers, builders, the persistent monitor, etc...) should run on Linux systems. The monitor persists its entire state on disk, so can restart in the state that it has dumped in its previous run.

LTO friendly and would be brittle to use. At last, there won't be any practical user interface with such an approach. So persistence is practically needed, both without LTO and if using LTO.

[55] This is conceptually similar the the SQL dump files of current RDBMS. But of course *Bismon* don't use an SQL format, but its own textual format.

[56] We don't aim compatibility with all web browsers -including old ones- but just with the latest *Chrome* (v.70 or newer) and *Firefox* (v.63 or newer) browsers, using HTML5, JavaScript, WebSockets, AJAX technologies.

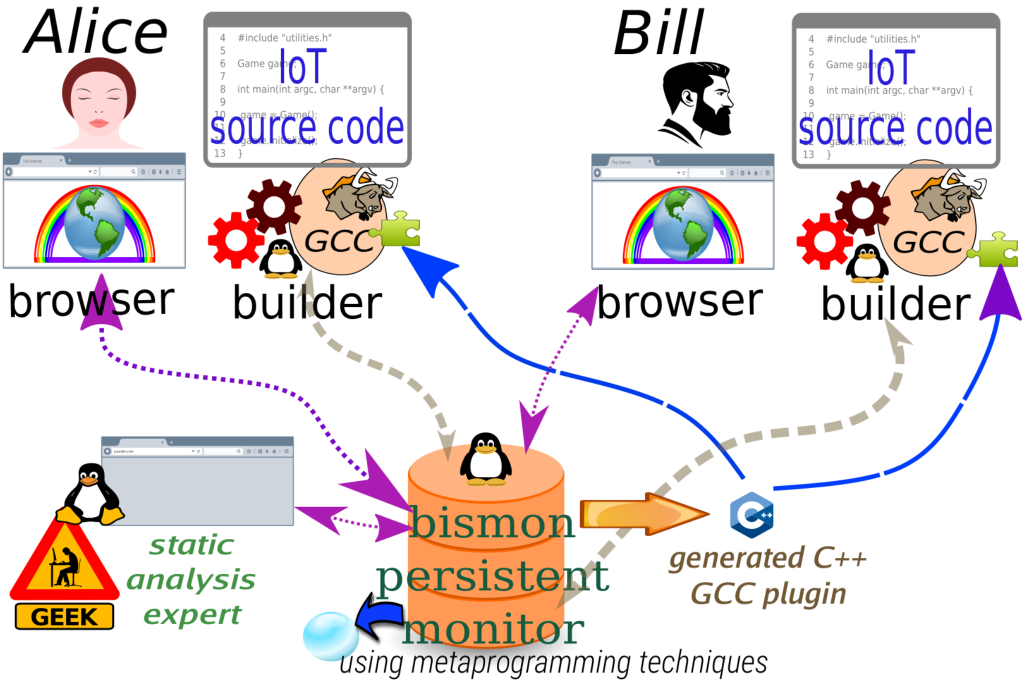

Figure 2: The *Bismon* monitor used by some IoT developer team following the CHARIOT approach.

1.5 Lessons learned from *GCC MELT*

Our previous *GCC MELT* project (see Starynkevitch [2011, 2008-2016, 2007]) provided a bootstrapped Lisp-like dialect (called *MELT*) to easily extend *GCC*. That Lisp-like dialect was translated to *GCC*-specific C++ code (in a transpiler itself coded in *MELT*). Then *GCC MELT* gave us the following insights:

- the *GCC* software is a huge free software project (with several hundreds of contributors, many of them working on GCC at least half of their time), which is *continuously* evolving. Following that evolution requires a significant effort by itself (see for example the mail traffic -of several hundreds messages monthly- on `gcc@gcc.gnu.org` and `gcc-patches@gcc.gnu.org`, and the amount of work shown at GCC summits or GNU Tools Cauldron, etc...). Switching to *Clang* would also require a lot of efforts.

- pushing a patch or contribution inside *GCC* is very demanding, since the community is quite strict (but that explains the quality of GCC).

- the *GCC* plugin API [57] is not "carved in stone", and can evolve incompatibly from one release of *GCC* to the next. Therefore, the C++ code of a plugin for `gcc-7` may require some (perhaps non-trivial) modifications to be usable on `gcc-8`, etc...

- a lot of C++ code (nearly two millions lines) was generated within our *GCC MELT* project, and the compilation by `g++` into a shared object of that emitted C++ code was a significant bottleneck.

- Implementing a generational copying garbage collector above *Ggc* [58] was challenging (debugging GC code takes a lot of time).

- Describing, in MELT, the interface to the many [59] C++ classes internal to *GCC* takes a lot of effort. Some automation would be helpful.

[57] including of course the API related to internal GCC representations, such as *Gimple* or *SSA*.
[58] This is the internal GCC garbage collector, a mark-and-sweep one which can run only *between* GCC passes.
[59] Current *GCC-8* have several thousands *GTY*-ed classes.

- In practice, whole-program static analysis requires a persistent system, since the "link-time optimization" ability of *GCC* is not plugin-friendly and is very rarely used.

The final D1.3^{v2} document may add further relevant items to the above list.

1.6 Driving principles for the *Bismon* persistent monitor

To enable whole program static source code analysis (for IoT software developers coding in C or C++ on a Linux developer workstation), we are developing *Bismon* [60], a persistent monitor (free software, for Linux). It leverages above existing recent *GCC* [cross-] compilers.

1.6.1 About *Bismon*

Bismon (a *temporary name*) is a free software (GPLv3+ licensed)[61] static source code whole-program analysis framework whose initial domain will be *Internet of Things* (or IoT)[62]. It is designed to work with the *Gcc* compiler (see **gcc.gnu.org**) on a Linux workstation[63]. *Bismon* is conceptually the successor of *GCC MELT* [64] (see Starynkevitch [2007, 2011]), but don't share any code with it while retaining several principles and approaches of *GCC MELT*.

Bismon is **work in progress**, and many things described here (this preliminary draft D1.3 ^{v1} of a future report D1.3 ^{v2} scheduled for M30 - in 2020) are not completely implemented in 2018 or could drastically change later.

Bismon is (like *GCC MELT* was) a **domain specific language**[65] implementation, targetted to ease static source code analysis (above the *GCC* compiler), with the following features:

- **persistence**[66], somehow *orthogonal persistence*. It is needed in particular because compiling some software project (analyzed by *Bismon*) is a long-lasting procedure involving *several* compiling and linking processes and commands. So most of the data handled by *Bismon* can be persisted on disk, and reloaded at the next run (cf. Dearle et al. [2010, 2009]). However, some data is temporary by nature [67] and should not be persisted. Such data is called temporary or **transient**. But the usual approach is to run the *Bismon* program from some initial loaded state and have it **dump** its memory state on disk [68]. before exiting (and reload that augmented state at the next run), and probably more often than that (e.g. twice an hour, or even every ten minutes).

- **dynamic typing**[69], like many scripting languages[70] (such as Guile, Python, Lua, etc). Of course the dynamically typed data inside the *Bismon monitor* is **garbage collected** (cf. Jones et al. [2016]). The initial GC of the monitor is a crude, mark and sweep, precise garbage collector, but multi-thread compatible (cf. §2.2 below); it uses a naive stop-the-world mark&sweep algorithm. That GC should be replaced by a bet-

[60] Notice `bismon` is a **temporary name** and could be changed, once we find a better name for it. Suggestions for better names are welcome.

[61] The source code is unreleased but available, and continuously evolving, on **https://github.com/bstarynk/bismon**

[62] IoT is viewed as the first application domain of *Bismon*, but it is hoped that most of *Bismon* could be reused and later extended for other domains

[63] Linux specific features are needed by *Bismon*, which is unlikely to be buildable or run under other operating systems. My Linux distribution is *Debian/Unstable*

[64] The *GCC MELT* web pages used to be on `gcc-melt.org` -a DNS domain relinquished in april 2018- and are archived on **https://starynkevitch.net/Basile/gcc-melt**

[65] See the *Domain-specific language* wikipage.

[66] See the *persistence* wikipage.

[67] E.g. data related to a web session, or to a web HTTP exchange, or to an ongoing `gcc` compilation process, etc... needs not to be persisted, and would be useless when restarting the *Bismon monitor*.

[68] Look also into Liam Proven FOSDEM 2018 talk about *The circuit less traveled* on **https://archive.fosdem.org/2018/schedule/event/alternative_histories/** for an interesting point of view regarding persistent systems.

[69] See also the *dynamic programming language* and *dynamic typing* wikipages.

[70] See the *scripting language* wikipage.

ter one, such as Ravenbrook MPS [71] something better, if good performance and scalability is wanted in *Bismon*. The hard point is the multi-threaded aspect of *mutator threads*[72] in that `bismon` program. For more details about the difficulty of clever trade-offs in such multi-threaded and user interaction-friendly garbage collectors, read carefully chapters 14 to 18 of Jones et al. [2016].

- **homoiconicity**[73] and **reflection**[74] with **introspection**[75] (cf Pitrat [1996, 1990, 2009a,b]; Doucet et al. [2003]; Carle [1992]): all the DSL code is explicitly represented as data which can be processed by *Bismon*, and the current state (including even its continuation represented as a reified call stack, cf Fouet and Starynkevitch [1987]; Fouet [1987]; Starynkevitch [1990]; Reynolds [1993] and §1.6.2) is accessible by the DSL.

- **translated**[76] to *C* code; and **generated** *JavaScript + HTML* in the browser, and generated *C++* code of *GCC* plugins

- **bootstrapped**[77] **implementation**: (cf. Pitrat [1996]; Polito et al. [2014]) ideally, all of *Bismon* code (including C code dealing with data representations, persistent store, etc...) should be generated (but that won't happen entirely with the CHARIOT timeframe). However, this ideal has not yet be attained, and there is still a significant amount of hand-written C code. It is hoped that most of the hand-written C code will eventually become replaced by generated C code.

- ability to **generate GCC plugins**: the C++ code of GCC plugins performing static analysis of a single translation unit should be generated (this was also done in GCC MELT, see Starynkevitch [2011]).

- with **collaborative web interface**[78], used by a *small* team of *trusted and well-behaving* developers [79]. The users of *bismon* are expected to trust each other, and to use the `bismon` tool responsibly[80] (likewise, developers accessing a `git` version control repository are supposed to act responsibly even if they are technically capable of removing most of the source code and its history stored in that repository). So protection against malicious behavior of `bismon` users is out of scope.

  Since *Bismon* should be usable by a small team of developers (perhaps two or a dozen of them)[81], it is handling some personal data (relevant to GDPR), such as the first and last names (or pseudos) of users and their email and maintain a password file (used in the Web login form). Compliance to regulations (e.g. European GDPR Voigt and Von dem Bussche [2017]; Zuboff [2015]) is out of scope and should be in charge of the entities and/or persons using and/or deploying *Bismon*. The login form template [82] could and probably should be adapted on each deployment site (by giving there site-specific contacts, such as the GDPR data controller, and perhaps add corporate logos and social rules, etc...). We are sadly aware[83]

[71] Improving that GC is a difficult work, and past experience on *GCC MELT* taught us that developing and debugging a GC is hard, and is a good illustration of Hofstadter's law (See Hofstadter [1979]). We should consider later using MPS from **`https://www.ravenbrook.com/project/mps/`** -or maybe some other state of the art garbage collector, since MPS might become an orphaned free software project- but doing that could require recoding or regenerating *a lot* of code, since MPS -like any other GC- has specific calling conventions and invariants, including A-normal form. So, switching to MPS or to any other good enough garbage collector in *Bismon* would require at least several months of work. And, as an open-source project, MPS looks barely maintained in mid-2019.

[72] In garbage collection parlance, a mutator thread is an applicative thread needing GC support, e.g. for allocation or updates of GC-ed memory zones.

[73] See the *homoiconicity* wikipage.

[74] See the *reflection* wikipage.

[75] See the *self-awareness*, *type introspection* and *virtual machine introspection* wikipages.

[76] See also the *source-to-source compiler* (or *transpiler*) wikipage.

[77] Read the *bootstrapping (compilers)* and *Chicken or the egg* wikipages.

[78] See also the *web application* wikipage, but `bismon` has highly specific pecularities, detailed more in §4.2 below.

[79] The initial `bismon` implementation had a hand-coded crude GTK interface, nearly unusable. That interface is temporarily used to fill the persistent store till the web interface (generated by *Bismon*) is usable. The GTK interface is already obsolete and should disappear at end of 2018. The Web interface (work in progress!) is mostly generated - all the HTML and JavaScript code is generated (or taken from outside existing projects e.g. *JQuery* or *CodeMirror*), and their HTML and JavaScript generators are made of generated C code.

[80] For example, each *bismon* user has the technical ability to erase most of the data inside *Bismon monitor*, but is expected to not do so. There is no mechanism to forbid or warn him doing such bad things.

[81] So *Bismon*, considered as a Web application, would have at most a dozen of browsers -and associated users- accessing it. Hence, scalability to many HTTP connections is not at all a concern (in contrast with most usual web applications).

[82] on **`https://github.com/bstarynk/bismon/blob/master/login_ONIONBM.thtml`**

[83] Rabelais wrote in his *Pantagruel* : "Science sans conscience n'est que ruine de l'âme." and that very polysemic (so quite difficult

that a mature *Bismon* system might -exactly like even `git` could be, and for quite similar reasons- in the future be unethically [ab-]used (as most current other distributed or Cloud digital technologies, Thain et al. [2005]; Dikaiakos et al. [2009]; Attiya and Welch [2004]; Peleg [2000]) in abusive ways (e.g. excessive surveillance Helbing et al. [2019]; Zuboff [2015]; O'Neil [2016]; Huws [2015] of developers using it), especially when combined with other intrusive automated personel monitoring techniques, such as face recognition (cf Jain and Li [2011]), email classification (e.g. Klimt and Yang [2004]) or systems like Chinese Social Credit System.

- **multi-threaded** - as many "server" like programs, *Bismon* should practically be multi-threaded to take advantage of current multi-core processors with shared virtual memory . Therefore synchronization issues (using condition variables and mutexes and/or atomic variables) between threads become important to avoid race conditions.
- with a **syntax-oriented editor** or syntactical *editor* (for our DSLs), inspired by the ideas of MENTOR in Donzeau-Gouge et al. [1980] (so see also Jacobs and Rideau-Gallot [1992], a tutorial on the related, even older, CENTAUR system. We should also follow Amershi et al. [2019]). So the static analysis expert is not typing some raw text (in some concrete syntax of our DSL) later handled by traditional parsing techniques (as in Aho et al. [2006]) but should interact using a web interface to *modify* and *enhance* the persistent store (like old Lisp machines or Smalltalk machines did in the 1980s), partially shown in a web browser (see also §4.2 below). That web interface is facilitating *refactoring* of DSL code.

It should be noticed that ***Bismon* is** actually **a somehow generic framework**, designed and usable to ease static analysis of C or C++ programs using generated *GCC* plugins and other runtime emitted code. As an orthogonally persistent, meta-programmable, reflexive, and multi-threaded framework, and with a few years of additional work and funding, *outside* of the CHARIOT project, **it could be even used for many other purposes**, including *artificial intelligence* and *data mining* applications on small volumes[84] of data, web-based *collaborative software* tools (see also Echeverria et al. [2019]; Kou and Gray [2019]; Gulay and Lucero [2019]; Dragicevic et al. [2019]), various multi-user *web applications*, *declarative programming* approaches, etc...

1.6.2 About *Bismon* as a domain-specific language

Notice that *Bismon* is not even thought as a *textual* domain specific language [85] (and this is possible because it is persistent). There is not (and there won't be) any canonical *textual* syntax for "source code" of the domain specific language in *Bismon* [86]. *Source code* is defined (socially) as the preferred form on which developers are working on programs. For C or C++ or Scheme programs, source code indeed sits in textual files in practice (even if the C standard don't speak of files, only of "translation units", see ISO [2011a]), and the developer can use any source code editor (perhaps an editor unaware of the syntax of C, of C++, of Scheme) to change these source files. In contrast, a developer contributing to *Bismon* is browsing and changing some internal representations thru the *Bismon* user interface (a Web interface [87]; see also Myers et al. [2000] for a survey) and interacts with *Bismon* to do that. There is no really any abstract syntax *tree* (or AST) in *Bismon*: what the developer is working on is some *graph* (with circularities), and the entire persistent state of *Bismon* could be viewed as some large graph in memory.

Conceptually the initial *Bismon* DSL is at first a dynamic programming language, semantically similar to Scheme, Python (or to a lesser degree, to JavaScript: however, it has classes, not prototypes, with single-inheritance), and is somehow *compiled* or *transpiled* to C. It is *essentially* (unlike most Scheme or Python

to translate) sentence was written in 1532!

[84]The entire analyzed data should fit in *Bismon* persistent store, so dozens of gigabytes, not terabytes.

[85]In contrast, *GCC MELT* was textual and had `*.melt` source *files* edited with `emacs` using its Lisp mode. This made refactoring difficult, since automatic move of textual fragments was not realistically possible.

[86]This idea is not new: neither Smalltalk (cf. Goldberg and Robson [1983]), nor Common Lisp (cf. Steele [1990]), are defined as having a textual syntax with an associated operational semantics based on it. Even the syntax of C is defined *after* preprocessing. What is -perhaps informally- defined for Smalltalk and Common Lisp is some abstract internal representation of "source code" in "memory" and its "behavior". In contrast, Scheme has both its textual syntax and its semantics well defined in R5RS, see Adams et al. [1998].

[87]in mid-2018, that Web interface was incomplete, and I still had to temporarily use some obsolete GTK-based interface that even I

or JavaScript implementations) a *multi-threaded* language, since the emitted routines can run in parallel in our agenda machinery (cf. §1.7 below). Meta-programming techniques (inspired by Lisp macro systems, see Queinnec [1996], and largely experimented in *GCC MELT*) will ease the extension of that language.

The base *Bismon* DSL is currently implemented [88] as a naive transpiler to C code (respecting the coding rules of our implementation, in particular of our garbage collector, see §2.2).

Our initial DSL is designed in terms of code representations as objects (see §2.1.2 below) and immutable values (see §2.1.1 below). It is not defined by some EBNF textual syntax. For example, an assign statement $\alpha = \beta$ is represented by an object of class `basiclo_assign` with its first component representing the left hand-side α and the second component representing the right hand-side β. Expressions in our DSL are either objects, or nodes, or scalars (integers, strings, etc...).

What is transpiled to C are Bismon "modules" (for example our `webjs_module` contains code related to emission of JavaScript), each with a sequence of routines. A module can be dumpable (into the persistent state, which then contains also the generated C code) or temporary (then the generated C code is not kept in that state). A routine or function [89] is an object of class `basiclo_function` or its subclass `basiclo_minifunction`, etc... A function knows its arguments, local variables, local numbers, body by various attributes (e.g. `arguments`, `locals`, `numbers`, `body` etc...). Its body is a block made of statements.

Statements of our DSL include:

- assignments, of class `basiclo_assign`, as explained above.
- run statements, of class `basiclo_run`, which "evaluates" its single operand for side effects (similar to expression statements in C or Go). As a special (and rather common) case, that operand can be a "code chunk" (conceptually similar to GCC MELT's code chunks, see Starynkevitch [2011] §3.4.1), that is a node of connective `chunk` providing a "template" for expansion as C code.
- conditional statements, of class `basiclo_cond`, inspired by Lisp's `cond`. Its components are a sequence of when clauses (which are objects of class `basiclo_when`) followed the "else" statements or blocks. The `nb_conds` attribute in the statement gives the number of when clauses.
- for loops, we have a `basiclo_while` class of statements (for "while" loops) and a `basiclo_loop` class (for infinite loops). Exiting of loops and blocks are using the `basiclo_exit` class. Return statements use the `basiclo_return` class.
- failure (inspired by Go's `panic`) statements are of class `basiclo_fail`. Failures are not exceptions, but prematurely terminate the tasklet (of the agenda, see §1.7 below) running the function containing that statement.
- locking of objects use a `basiclo_lockobj` class (mentioning both an object to lock and a sequence of sub-statements or blocks). A locked object is unlocked when the end of its locking statement is reached, or when the currently active routine terminates (on failure or on return).
- execution of primitive side-effecting operations with no result happens in C-expansion statements (of class `basiclo_cexpansion`), inspired by GCC MELT's primitives (see Starynkevitch [2011]) returning `:void`.
- etc...

Expressions in our DSL are typed (with types like `value`, `object`, `int`, `string`, etc...) and include:

- scalar (integers, constant strings)

find so disgusting that I won't explain it, and sometimes even to edit manually some `store2.bmon` data file, cf § 2.3.1.

[88] See notably our hand-written files `gencode_BM.c` and `emitcode_BM.c` in october 2018. Our bootstrap philosophy might require replacing later these hand-written files by better, *bismon* generated, modules.

[89] Technically the routine would be in the module's shared object binary; the function is a *Bismon* object reifying the code of that routine.

- local variables, or arguments, or numerical variables (of the current function).
- constant objects (mentioned with `constants` in the function)
- closure application (represented by a node of connective `apply`). Often, the closure's connective would be a function object.
- quotations (like in Lisp, represented by an unary node of connective `exclam`)
- message sending (represented by a node of connective `send`)
- primitives (inspired by GCC MELT's ones, see Starynkevitch [2011]; the connective of the node is of class `basiclo_primitive`)
- builtin objects like `current_closure`, `current_closure_size`, `current_module`, `current_routine`, `null_value`, `null_object` are expanded in some ad-hoc fashion [90].
- etc...

1.6.3 About *Bismon* as a evolving software system

So *Bismon* is better thought of as an evolving software system. We recommend to try it. Notice that *Bismon* is **provided as free software** (available on **https://github.com/bstarynk/bismon/** but unreleased in 2018) in *source form only* and should be **usable** *only* **on a Linux/x86-64 workstation**... (typically, at least 32 gigabytes of RAM and preferably more, at least 8 cores, several hundreds gigabytes of disk or SSD).

The *Bismon* system contains **persistent data** (which is part of the system itself and should not be considered as "external" data; each team using *Bismon* would run its own customized version of their *Bismon monitor*.), and should be **regularly backed up**, and preferably version controlled at the user site. It is strongly recommended to use `git` [91] or perhaps some other distributed version control system, to `git commit` its files several times a day (probably hourly or even more frequently, as often as a developer is committing his C++ code), and to backup the files on some external media or server at least daily. How that is done is outside of the scope of this document. The *dump facilities* inside *Bismon* are expected to be used quite often (as often as you would save your report in a word processor, or your source file in a source code editor), probably several times per hour. So a developer team using *Bismon* would probably `git clone` either `git@github.com:bstarynk/bismon.git` thru SSH or **https://github.com/bstarynk/bismon.git**, build it (after downloading and building required dependencies), and work on that `git` repository (and of course back-up it quite often).

We are still growing *Bismon* by feeding it with additional interactions changing its persistent state. At first, we developed (at begin of bootstrap) a crude GTK interface, shown in figure 3, which is a screenshot made on October 22nd 2018 on git commit `cbdcf1ec351c3f2a`, when working on the JavaScript generator inside *Bismon*. It shows several windows: the large top right window (named `new-bismon`) has a command textview (ivory background, top panel) and a command output (azure background, bottom panel). The small top left window (named `bismon values` show the read-eval-print-loop output (as `$a` in two panes). The mid-sized bottom left window (titled `bismonob#1`) shows (in two text-views of the same GTK text buffer) shows (in top text-view) a large part of the body of the `emit_jsstmt` method for the `basiclo_while` class of *Bismon* and (in bottom text-view) some components of our `webjs_module` object. In the rear, bottom right, a tiny part of our `emacs` editor (used to run `bismon -gui`...) is visible, and shows a backtrace [92].

This crude GTK3 interface [93]. Anonymous objects are displayed also with their `comment` predefined attribute[94]. We won't debug that GTK code -which crashes often[95]- but will remove it once it can be replaced

[90] That is: `current_closure` → the current closure; `current_closure_size` → its size; `current_module` → the current module; `current_routine` → the current routine; `null_value` → the null value; `null_object` → the null object; respectively.

[91] See **http://git-scm.com/**

[92] Ian L. Taylor's `libbacktrace` is used in *bismon* to provide symbolic backtraces.

[93] It is implemented in 12.5KLOC of C code in `gui_GTKBM.c`, `newgui_GTKBM.c` and `guicode_BM.c`

[94] See not only the comment wikipage but think of a `comment` macro in LISP, SCHEME or GCC MELT which would ignore all its arguments and stays macro-expanded to nil.

[95] Because of a design bug related to garbage collection, practically too costly to be fixed.

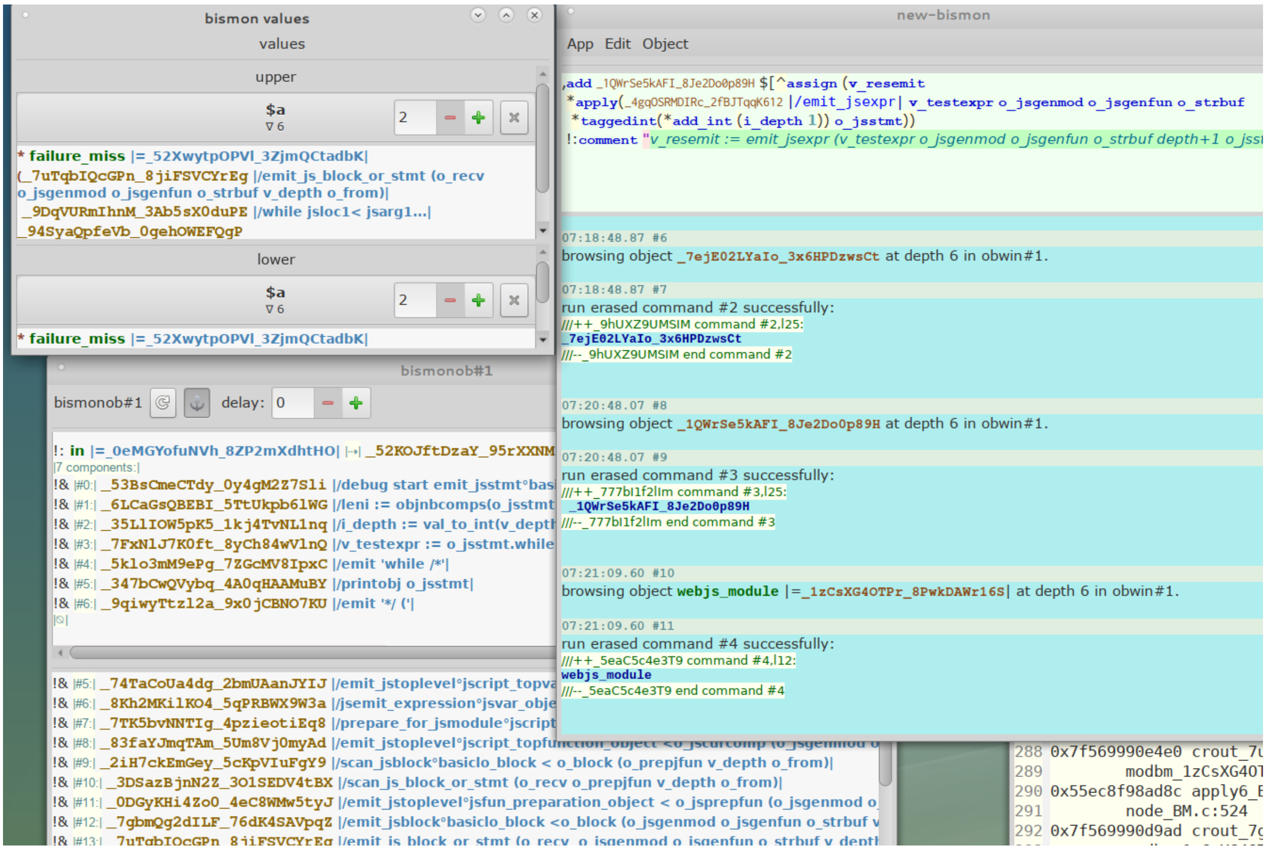

Figure 3: crude (soon deprecated) GTK interface oct. 22, 2018, git commit `cbdcf1ec351c3f2a`

by a web interface. It will and needs to be replaced by a Web interface. Several lessons have been gained with this experience:

- using GTK [96] is in practice incompatible with a multi-threaded precise garbage collector [97], like the one in *Bismon* (cf §2.2 below), in particular because GTK may have several nested event loops, so many local garbage collector pointers in internal call frames (which are not accessible from routines above).
- the model and the C API provided by GTK text views and text buffers is not adequate for structured syntactic editing (like pionneered in Mentor, see Donzeau-Gouge et al. [1980]). It is still too low-level and oriented for plain textual edition.
- GTK is not compatible with several X11 displays, so a single *bismon* process cannot handle several users each having its own screen.

So we decided to stop investing efforts on the GTK interface, and give more priority to a Web interface, which is required once a small team of *several* IoT developers need to interact with the *bismon* persistent monitor. The GTK interface is just temporarily needed to fill the persistent store (till our web interface is usable). We hope that it will be entirely scrapped and should be replaced by a web interface[98] , and the static analysis expert (and other users pof *Bismon*) will interact with *bismon* thru some Web interface.

Work on the future Web interface has significantly progressed [99]. New users -called *contributors* - can be

[96] Since GTK is a free software library, we could consider patching its source code, but such a huge effort is not reasonable within the timeframe of CHARIOT, and GTK is still evolving a lot, so patching it would require freezing its version. `gtk+-3.24.1` has 1.2 millions lines of source code, measured by D.Wheeler `sloccount` but it depends also on other libraries, such as Pango, Glib, etc... so patching GTK source for our precise GC is not reasonable at all.

[97] See also **https://stackoverflow.com/q/43141659/841108** about GTK and Boehm's conservative GC.

[98] Notice that copy/pasting becomes then a difficult issue, see **https://softwareengineering.stackexchange.com/q/393837/40065**.

[99] With the hand-written `web_ONIONBM.c` using `libonion`, and the *bismon* module `webjs_module` translated into the `modules/modbm_1zCsXG4OTPr_8PwkDAWr16S.c` emitted C file of more than 6.5KLOC in october 2018.

voluntarily registered and unregistered on the command line [100] into *bismon* in a way similar to `git`. When they access any dynamic web page, a login web form appears (with some GDPR related notice) if no web cookie identifies them. But that Web interface is still incomplete in October 2018. Several design decisions have been made: we will use the `codemirror` [101] web framework to show the analyzed source code of IoT software. The web interface for IoT developers should be a "single-page application" one (so AJAX, HTML5, CSS3 techniques, with generated JavaScript and HTML code). *WebSockets* should be used for asynchronous communication between browser and the *bismon* monitor. The `jquery`, `angular`, `semantic-ui`, etc... web frameworks are considered as building blocks for that Web interface and should be installed with inside *bismon* [102] to enable using *bismon* without any external Internet connection.

IoT developers working with the *Bismon monitor* will use some Web interface to interact with it.

1.6.4 About *Bismon* as a static source code analyzer framework

The *Bismon* persistent monitor will generate the C++ code of GCC plugins, leveraging on the experience of GCC MELT (see Starynkevitch [2011, 2007, 2008-2016]). The C++ code generator will have a design similar to (and share some code and classes with) our internal initial DSL (cf §1.6.2). It is extremely likely that in many cases, such a generated GCC plugin would just insert its appropriate passes by using the pass manager (cf GCC Community [2018] §9 and §24.3), and these passes will "serialize" internal representations (either in JSON, or using Google protocol buffer, or using a textual format close to our dump syntax, see figure 5 below, etc...) such as *Gimple*-s, *Basic Block*-s and transmit some form of them to the *Bismon* persistent monitor. In some simple cases, it is not even necessary to transmit most of that representation. For instance, a whole program static analysis to help avoiding stack overflow needs just the size of each call frame [103] and the control flow graph (so only the *Gimple* call statements, ignoring anything else); with that information (and the control flow graph) the monitor should be able to estimate an approximation [104] of the consumed call stack, whole program wide.

Several design decisions have been made regarding the style of the generated C++ code of GCC plugins: it will use existing scalar data and `GTY`-ed classes (see GCC Community [2018] §23), to take advantage of the existing GCC garbage collector (*Ggc*). Contrarily to GCC MELT, it won't provide a generational garbage collector (because most of the processing happens in the monitor, hence performance of the generated GCC plugin[105] is less important), so transforming to A-normal form is not required at translation (to C++) time.

1.7 Multi-threaded and distributed aspects of *Bismon*

The *Bismon monitor* is by itself a multi-threaded process [106]. It uses a *fixed thread pool* of *worker threads* (often active) [107], and additional (generally idle) threads for web support and other facilities. The *Bismon monitor* is occasionally starting some external processes, in particular for the compilation of generated *GCC* plugins, and for the compilation into *module*s -technically "plugins"- of dynamically generated *C* code by *Bismon*; later it will dynamically load (with `dlopen`) these modules, and thus *Bismon* can increase its code (but cannot decrease it, even if some code becomes unused and unreachable); however such modules are *never* garbage collected (so `dlclose` is never called). So in practice, it is recommended to restart *Bismon* every day (to avoid endless growth of its code segments).

[100] Use the option `-contributor=` to add them, `-remove-contributor=` to remove them and `-add-passwords=` to set their encrypted *bismon*-specific password.

[101] See **http://codemirror.net/** for more.

[102] For example, the *bismon* source tree has a `webroot/jscript/jquery.js` local file to serve HTTP `GET` requests to an URL like `http://localhost:8086/jscript/jquery.js` handled by the *bismon* monitor.

[103] Notice that *GCC* compute these call frame sizes (see the `-fstack-usage` option), and can detect excessively big call frame with `-Wstack-usage=` option.

[104] Of course dynamic calls, e.g. call thru function pointers, make that much more complex and will require manual annotation.

[105] In practice, generated GCC plugins would simply "digest" some internal GCC representations and transmit their outcome to the Bismon monitor.

[106] In contrast of most scripting languages implementations such as Python, Ocaml, Ruby, etc..., we try hard to avoid any "global interpreter lock" and strive to develop a genuinely multi-threaded monitor.

[107] The number of worker threads is given by the `-job` program argument to `bismon`. For an 8-cores workstation, it is suggested to set it to 5 or 6. It should be at least 2, and at most 15. This number of jobs also limits the set of simultaneously running external processes, such as `gcc` processes started by *Bismon*.

The worker threads of *Bismon* are implementing its **agenda** [108] machinery. Conceptually, the agenda is a 5-tuple of first-in first-out queues of **tasklets**, each such FIFO queue is corresponding to one of the five priorities : *very high*, *high*, *normal*, *low*, *very low*. Each agenda worker thread removes one tasklet (choosing the queue of highest possible priority which is non empty, and picking the tasklet in front of that queue) and runs that tasklet quickly. A tasklet should run during a few milliseconds (e.g. with some implicit kind of non-preemptive scheduling) at most (so cannot do any blocking IO; so input and output happens outside of the agenda). It may add one or more tasklets (including itself) to the agenda (either at the front, or at the end, of a queue of given priority), and it may remove existing tasklets from the agenda. Of course tasklets run in parallel since there are several worker threads to run the agenda. The agenda itself is not persisted as a whole, but tasklets[109] themselves may be persistent or transient. Tasklets can also be created outside of the agenda (e.g. by incoming HTTP requests, by completion of external processes, by timers, ...) and added asynchronously into the agenda.

Outside of the agenda, there is an *idle queue* of delayed todo closures (a queue of closures to be run, as if it was an idle priority queue) with some arguments to apply to them. But that idle queue don't contain directly any tasklets. That idle queue can be filled by external events [110]. Of course the idle queue is not persisted.

In its final version, the *Bismon system* will involve several cooperating Linux processes:

- the *Bismon monitor* itself, with several threads (notably for the agenda mechanism described above)
- the web browsers of developers using that particular *Bismon monitor*; each developer probably runs his/her own browser. That web browser is expected to follow latest Web technologies and standards (HTML5, Javascript6 i.e. EcmaScript 2016 at least, WebSockets, ...). It should probably be a Firefox or a Chrome browser from 2017 or after. The HTML and Javascript is dynamically generated by the *Bismon monitor* and should provide (to the developer using *Bismon*) some "single-page application" (cf. Atkinson [2018]; Queinnec [2004]; Graunke et al. [2003]) feeling [111].
- the IoT developers using *Bismon* will build their IoT firmware as usual; however they will add some extra options (to their `gcc` or `g++` cross-compilation commands) to use some *Bismon* generated GCC plugin in their cross-compilation processes. So these cross-compilation processes (i.e. `cc1` started from `gcc`, or `cc1plus` started from some `g++`, etc...), augmented by generated plugins, are involved.
- sometimes *Bismon* would generate some `modules/*.c` file during execution, and fork a (direct) compilation of it (technically forking a `./build-bismon-persistent-module.sh` -for persistent modules- or a `./build-bismon-temporary-module.sh` -for temporary modules- shell script, which invokes `make` which runs some `gcc` command) into a "plugin" module `modubin/*.so`, which would be `dlopen`-ed.
- *Bismon* should also generate the C++ code of *GCC plugins*, to be later compiled then used (with `gcc` or `g++` option `-fplugin`). Two kinds of *GCC* plugins are considered to be generated:
    1. usually, the GCC plugin [112] would be generated to assist [cross-] compilation (e.g. of IoT software) by developers using *Bismon*. So for an IoT developer targeting some RaspberryPi, it could be a GCC plugin targeting the `arm-linux-gnueabi-gcc-8` cross-compiler (but the C++ code of that plugin needs to be compiled by the native `gcc` on the host system).
    2. But the GCC API is so complex (and under-documented) that it is worth extracting it automatically

[108] Details about the agenda, such as the fixed set of available priorities, are subject to change. We describe here the current implementation in mid-2018.

[109] Actually tasklets are objects (see §2.1.2 page 30 below), and to run them, the agenda is sending them a message with the predefined selector *run_tasklet*.

[110] For example, when an external compilation process completes, that queue is filled with some closure -provided when starting that compilation- and, as arguments, an object with a string buffer containing the output of that process, and the integer status of that process.

[111] So using your browser's backward and forward navigation arrows won't work well because in single-page applications they *cannot* work reliably

[112] It is tempting to call such plugins *cross*-plugins, since they would be `dlopen`-ed by a cross-compiler.

by sometimes generating a GCC plugin [113] to inspect the public headers of GCC[114]. Even when the end-user developer is targetting a small IoT chip requiring a cross-compiler (like `arm-linux-gnueabi-gcc-8` above), these GCC inspecting plugins are for the native `gcc` (both Schafmeister [2016] and Schafmeister [2015] are inspirational for such an approach).

We are considering several ways of providing (to the IoT developer using them) such generated C++ code for GCC plugins. We might generate (at least for the first common case of GCC plugins generated for developers using *Bismon*, and large enough to need several [115] generated C++ files) `*.shar` archives (obtained by Web requests, or perhaps some `wget` or `curl` command in some `Makefile`) for *GNU sharutils* [116] containing the C++ code and also the `g++` command compiling it. That archive could instead be just a `.tar.gz` file (and the IoT developer would extract it, and run `make` or `ninja` inside the extracted directory to build the shared object GCC plugin binary file), etc... For a *small* generated GCC plugin fitting in a single generated C++ file of less than a dozen thousands lines, we could simply serve in *Bismon* an URL like `http://localhost:8086/genplugin23.c` and require the IoT developer to fetch then use that. Other approaches could also be considered. The rare second case (GCC plugin code generated to inspect the GCC API, running on the same machine as the *Bismon* server) could be handled thru external processes (similar to compilation of *Bismon* modules). Alternatively, we might consider delegating such plugin-enhanced cross-compilation processes to the *Bismon* monitor itself, etc, etc...

In principle, the various facets of *Bismon* can run on different machines as distributed computing (obviously the web browser is not required to run on the same machine as the *Bismon monitor*, but even the various compilations -of code generated by *Bismon*, and the cross-compilation of IoT code- could happen on other machines).

Conceptually, we aim for a **multi-tier programming** approach (inspired by Ocsigen [117] with the high-level DSL inside *Bismon* generating code: in the *Bismon monitor*, as modules; in the *web browser*, as generated Javascript and HTML; in the *GCC* compiler, as generated GCC plugins.

[113] It is tempting to call such plugins *straight*-plugins, since they would be `dlopen`-ed by a straight compiler, not a cross-compiler.

[114] In *GCC MELT*, we tried to describe by hand-coded *MELT* code a small part of that GCC API and its glue for *MELT*. This approach is exhausting, and makes following the evolution of GCC very difficult and time-consuming, since new *MELT* code should be written or manually adapted at each release of *GCC*. Some partial automation is needed to ease that effort of adapting to successive *GCC* versions and their non-compatible plugins API

[115] By past experience in GCC MELT, we did generate C++ files totalizing almost a million lines of C++ code, and compiling such a large generated C++ code base took dozens of minutes, and created a bottleneck.

[116] See **https://www.gnu.org/software/sharutils/** for more.

[117] See **https://ocsigen.org/**

2 Data and its persistence in *Bismon*

Notice that *persistent memory* is also a hardware technology sold by Intel. See `pmem.io` and Scargall [2020]. With care and programming efforts, such hardware could be used by the *Bismon monitor*.

Several checkpoint/restarting mechanisms, such as the CRIU library, are conceptually similar to the persistence mechanism[118] existing in BISMON, but have been developed with different trade-offs, and no concern for portability.

2.1 Data processed in *Bismon*

The *Bismon monitor* handles various kinds of data. A lot of data is immutable (its content cannot change once the data has been created, for example strings). But **objects** are of course mutable and can be modified after creation. Since *Bismon* is multi-threaded and its agenda is running *several* worker threads in parallel, these mutable objects contain a mutex for locking purposes.

So the *Bismon monitor* handle **values** [119] (represented in a 64 bits machine word, holding some pointer or some tagged integer) : they can be immutable, or objects (and such objects are the only kind of mutable data).

All values (immutable ones and mutable objects) are hashable [120] and totally ordered.

Since *Bismon* is able to persist and process efficiently and concurrently many kinds of symbolic data organized more or less in a semantic network (including most abstract syntax trees or AST-s and their symbolic annotations), it should be re-usable with additional efforts as a foundation for many other aspects [121] of static source code analysis, even of programming languages which are not tied to *GCC*. Of course, some significant work related to parsing is then required.

The BARRELFISH operating system (see Gerber [2018]; Giceva et al. [2016]; Schüpbach et al. [2008]...) has a *System Knowledge Base* "used to store, query, unify, and compute on a variety of data about the current running state of the system" which is "based on a port of the ECLiPSE constraint logic programming system" Apt and Wallace [2006] (from Barrelfish [2013]), and BARRELFISH aims be used for IoT computer networks.

2.1.1 Immutable values

They include

- **tagged integer** (of 63 bits, since the least significant bit is a tag bit). The integer won't change (and integer values don't require extra space to keep that integer, since they are encoded in the pointer).
- UTF-8 encoded **string** (the bytes inside such strings don't change).
- boxed constant **double** (IEEE 754) floating point [122] numbers; but `NaN` is *never* boxed and becomes reified as the `nan_double` *object*; hence boxed doubles stay comparable (with the convention [123] that $-0.0 < +0.0$, even if they compare equal in IEEE 754).
- **tuple** of objects, that is an ordered (but immutable) sequence of object pointers (the size or content of a tuple don't change). A given object could appear in several positions in a tuple.
- **set** of objects, represented internally as a sorted array of objects' [i.e. pointers]. A given object can occur only once in a set, and membership (of an object inside a set) is tested dichotomically in logarithmic time. Of course, the size and content of a set never change.

[118] See `stackoverflow.com/a/62927624/841108` for more.

[119] To extend *Bismon* to handle some new kind of custom data (such as bignums, images, neural networks, etc...) processed by external libraries, it is advised to define new *payloads* inside objects (cf. §2.1.2 below), without adding some new kind of values.

[120] Their hash is a non-zero 32 bits unsigned number. Only the nil pointer has an hash of 0.

[121] For an example, look into the ongoing DECODER H2020 project, ICT-16 call, grant agreement number 82423.

[122] The practical motivation for floating point numbers is mostly related to JSON, since the `jansson` library handle differently floating JSON and integer JSON values, and perhaps to rare storage of timing data (CPU or elapsed time) expressed as a floating point number. We don't intend *Bismon* to be used for complex numerical processing. Should machine learning libraries become useful - after the CHARIOT project - in *bismon*, their data would probably become some object payload.

[123] The intuition behind such a convention would be that -0.0 is a very (or infinitely) small negative number, so less that $+0.0$ which could be a very small positive number.

- **node**. A node has an object connective, and a (possibly empty, but fixed) sequence of sons (sons are themselves values, so can themselves be integers, strings, tuples, sets, sub-nodes). The connective, size and sons of a node don't change with time. Since a node is immutable and knows all its sons at creation time, circularity (e.g. a node having itself as some grand-son) inside it is impossible, and the node has a finite depth. Nodes whose connective is an instance of `transient_connective_object` are not dumped so are transient.

- **closure**. A closure is like a node (it has a connective and sons), but its connective is interpreted as the object giving the routine (see §2.1.2 below) to be called when that closure is applied, and its sons are considered as closed values. Closures whose connective is an instance of `transient_connective_object` are not dumped so are transient.

The **nil** value is generally not considered as a value, but as the absence of some value. We might (later) add other kind of values (perhaps vectors of 64 bits integers, of doubles, bignums ...), but they should all be immutable. However, it is very likely that we prefer complex or weird data to sit inside objects, as payload. There is also a single **unspecified** value (which is non-nil so cannot be confused with lack of value represented by nil).

Tuples and nodes and closures could contain nil, but sets cannot. A node or closure connective is a genuine object (so cannot be nil), even if nodes or closures could have nil sons.

Sets and tuples are sometimes both considered as **sequences** and share some common operations.

The immutable values are somehow lightweight. Most of them (strings, sets, tuples, nodes) internally keep some hash-code (e.g. to accelerate equality tests on them, or accessing hash tables having values as keys). The memory overhead for values is therefore small (a few extra words at most, to keep GC-data type and mark, size hash).

The size of values (byte length of strings, number of objects in tuples or sets, number of sons in nodes or closures) can in principle reach $2^{31} - 1$ but is generally much smaller (often less than a few dozens) and could be 0.

Mutable values outside of objects (and their payload, see §2.1.2 below) cannot exist.

Values (even references to objects, e.g. inside sequences or nodes) are represented as a machine pointer and fit in a 64 bits word. When its least significant bit is 1, it is a tagged integer.

Values, including objects, are comparable so sortable. For strings, nodes, closures, sets, tuples we use a lexicographical order. Values also have an hashcode to be easily put in hash tables, etc..

2.1.2 Mutable objects

Objects are the only kind of mutable values, and are somehow heavy (at least several dozens of machine words in memory for each object). They can be accessed nearly simultaneously by several worker threads running different tasklets, so they need a locking mechanism and contain a (recursive) mutex [124] (so in reality only one thread is accessing or modifying them at a given instant).

Conceptually, objects contain the following data:

- a constant *unique* serial id (of about 128 bits), called the **objid**, randomly generated at object creation time and never changed after. In many occasions, that objid is printed as 24 characters (two glued blocks of 12 characters each, the first being an underscore _, the second being a digit, the 10 others being alphanumerical with significant case) such as `_4ggW2XwfXdp_1XRSvOvZqTC` [125] or `_0xbmmxnN8E8_0ZuEqJmqMNH`. It is expected that objid collisions never occur in practice, e.g. that even thousands of *Bismon monitor* processes (running on many distant computers) would in fact never generate the same objid. In other words, our objids are as unique as UUIDs (from RFC 4122) are (but are displayed differently, without hyphens). In practice, the first 5 or a few more characters of an objid are enough to uniquely identify it, and we use

[124] Each object has its mutex initialized with `pthread_mutex_init(3p)` with the `PTHREAD_MUTEX_RECURSIVE` attribute, and lockable with `pthread_mutex_lock(3p)` etc...

[125] That objid `_4ggW2XwfXdp_1XRSvOvZqTC` is for the predefined object *`the_system`*, and corresponds to the two 64 bits numbers 3577488711679049683 (encoded in base 62 $= 10 + 2 \times 26$ as `4ggW2XwfXdp`), 1649751471969277032 i.e. to 128 bits hexadecimal `0x31a5cb0767997fd316e5183916681468`.

the objid abbreviation [126], e.g. _4ggW2XwfX - an abbreviation for _4ggW2XwfXdp_1XRSvOvZqTC - to uniquely identify it. The concrete textual "syntax" for objid-s (starting with an underscore then a digit, etc...) is carefully chosen to be compatible and friendly with identifiers in C, C++, JavaScript, Ocaml, etc. The *Bismon* runtime maintains a large array of hashtables and mutexes to be able to quickly find the object pointer of a given objid (if such an object exists in memory). The objid is used to compare (and sort) objects. The (32 bits, non-zero) hash code of an object is obtained from its objid (but it is cached in the object's memory, for performance reasons).

- the recursive **mutex lock** of that object [127]. So locking (or unlocking) an object really means using that lock on pthread mutex operations [128].
- a **space** number fitting in a single byte. The space 0 is for *transient* objects that are not persisted to disk. The space 1 is for *predefined* objects (there are about 60 of them in Q3 of 2018), which are conceptually created before the start of *Bismon monitor* processes and are permanently available, even at initial load time of the persistent store. Those predefined objects are dumped in file `store1.bmon`, the objects of space 2 (conventionally called the *global* space) are dumped and persisted in file `store2.bmon`, etc...
- the **mtime** of an object holds its modification time, with a millisecond granularity, since the Unix Epoch. Touching an object is updating its *mtime* to the current time.
- the (mutable!) **class** of an object is an *atomic* [129] pointer to an object (usually another one) describing its class, as understood by *Bismon*. It is allowed to change dynamically [130] the class of any object. Classes describe the behavior (i.e. the dictionary of "methods"), not the content (i.e. the "attributes") of objects and enable single-inheritance (every class has one super-class).
- the *attributes* of an object are organized as an hash-table associating attribute or key objects to arbitrary non-nil values. An **attribute** is an arbitrary object, and its value is arbitrary (but cannot be nil).
- the *components* of an object are organized as a vector (whose size can change, grow, or shrink) of values. A **component** inside an object is a value (possibly nil).
- objects may contain one **routine** pointer (or nil), described by
  1. the *routine address* inside an object is a function or routine pointer (in the C sense, possibly and often nil). The signature of that function is described by the routine signature [131]
  2. the *routine signature* is (when the routine address is non-nil) describing the signature of the routine address above.

  Notice that routine address and signature can only change when a new module [132] is loaded (or at initial persistent state load time), and that can happen only when the agenda is inactive. Conceptually they are mostly constant (and do not require any locking).

  Most (in 2018, all) routines have the same C signature `objrout_sigBM` corresponding to the predefined object *function_sig*. For an object of objid Ω of that signature its routine address corresponds to the C name croutΩ_BM. For instance, to initialize (at load time) the object of oid _9CG8SKNs6Ql_4PiHd8cny the initial loader (or the module loader) would `dlsym` the `crout_09Hug4WGnPK_7PpZby8pz84_BM` C function name.

[126] This objid abbreviation is for documentation purposes; it is not acceptable in persistent store files.

[127] We have considered using a pthread `rwlock` instead of a `mutex`, but that would probably be more heavy and perhaps slower, but could be experimented in the future.

[128] So accessing without the protection of that lock being hold, any data inside an object, other than its constant objid, its class, its routine pointer, is forbidden and considered as *undefined behavior*

[129] Here, "atomic" is understood in the C or C++ memory sense; so a pointer declared `_Atomic` in C or `std::atomic` in C++, supposing that they are the same and interoperable. Hence the class of an object can be obtained *without* locking that object.

[130] Changing classes is permitted within reasonable bounds: the class of all classes should remain the *class* predefined object; all objects should be instances of the predefined *object* or more often of some indirect sub-class of it; of course these invariants cannot be proved.

[131] Perhaps all our routines will keep the same signature, and then it would not need to be explicitly stored.

[132] The generated C code of modules also contains an array of constant objids, ana another array of routine objids.

- objects may also have some (nearly arbitrary) **payload** - which can contain anything that don't fit elsewhere. That payload [133] is a pointer (possibly nil) to some client data owned by the object; the payload is usually not a value but something else. The garbage collector should know all the payload types. In 2018 the following payloads are possible (with other specialized payloads, e.g. for parsing, loading, dumping and web request and web session support, contributors):

  1. mutable *string buffer*.
  2. mutable *class* information (with its super class, and the method "dictionary" associating objects -selectors- to closures). The class objects are required to have such a payload.
  3. mutable *vector* of values (like for components).
  4. mutable doubly *linked list* of non-nil values.
  5. mutable *associative table* associating objects to non-nil values (like for attributes)
  6. mutable *hash set* of objects.
  7. mutable *hash maps* to associate arbitrary non-nil values used as keys to other arbitrary non-nil values.
  8. mutable *string dictionaries* associating non-empty strings to non-nil values.
  9. mutable JSON data [134]
  10. etc...

  Of course, the payload of an object should be initialized (so created), accessed, used, modified, changed to another payload, cleared (so deleted) only while that object is locked, and each payload belongs to only one object, its owner.

The ability to have *arbitrary* attributes and components in every *Bismon* object makes them **very flexible** (cf. Lenat [1983]; Lenat and Guha [1991]), and is *on purpose* related to frame languages (cf. Bobrow and Winograd [1977]), semantic networks (cf Van De Riet [1992]) and ontology engineering (cf Nicola et al. [2009]).

For convenience, (some) objects can also be (optionally) named, in some top-level "dictionary" or "symbol table" (which actually contain weak references to named objects). But the name of an object is not part of it.

2.2 garbage collection of values and objects

The *Bismon monitor* has in 2018 a precise, but naive, mark&sweep stop-the-world garbage collector for values [135], of course including objects. When the GC is running, the agenda has been de-activated, and no tasklets are running. Our initial GC is known to not scale well and to be unfriendly to serious interactive usage, so it should be replaced (see footnote 71 in §1.6.1 above).

In contrast to most GC implementations (but inspired by the habits of *GCC* itself -in its *Ggc* garbage collector- in that area), the garbage collector of the *Bismon monitor* is **not** *directly* triggered in allocation routines (but is started by the agenda machinery). When allocation routines detect that a significant amount of memory has been consumed, they set some atomic flag for wanting GC, and later that flag would be tested (regularly) by the agenda machinery which runs the GC. So when the GC is actually running, the call stacks are conceptually empty [136], and no tasklet is active.

The garbage collection roots include:

[133] Some weird payloads, in particular for web exchanges and web sessions, cannot be created programmatically by public functions or by Bismon code. Web exchange and web session payloads (cf §4.2) are only internally created by the HTTP server code in *bismon*, and cannot and should not be persisted.

[134] That is, `json_t*` from the *Jansson* library. It is practically useful for WebSocket messages.

[135] See also previous footnote 71 on page 21 for possible improvements of the GC.

[136] But the support threads, e.g. for web service with `libonion`, add complication to this scheme. However, ignoring conceptually the call stacks don't require us to use A-normal forms in module code, as was needed in *GCC MELT*, and facilitate thus the generation of C code inside them.

- all the tasklets queued in the (several queues) of the agenda
- all the predefined objects
- all the constants (objects [137]) referred by both hand-written and generated C code (including constants referred by modules, and objects reifying modules).
- some very few global variables (containing values), so conceptually the idle queue of closures, and the queue related to external running processes, the hash-set of active web request objects, etc.

The naive *Bismon monitor* garbage collection [138] works as follow: a queue of non-visited objects to be scanned is maintained, with an hash-set of marked objects. Initially, we visit the GC roots above. Visiting a value involves marking it (recursively for sequences, nodes, closures, ...) and if it is a newly marked object absent from the hash-set, adding that object to the scan queue and to the hash set of marked objects, but nodes or closures whose connective is an instance of `transient_connective_object` are ignored so transient. We repeatedly extract objects to be scanned from the queue and visit their content (including their attributes, their components, their signature and payload and the values inside that payload). When the scan queue is empty, GC is finished.

2.3 persistence in *Bismon*

Persistence is an essential feature of the *Bismon monitor*, and did inspire the ongoing work in REFPERSYS. It always starts by loading some previous persisted state, and usually dumps its current state before termination. On the next run, that new state is loaded, etc.... For convenience and portability, *the persistent state is a set of textual files* [139].

In the Internet world, persistence is generally handled *outside* of the application, by using databases. These databases can be relational (see Date [2005]) or non-relational (so called *NoSQL* databases, see Raj [2018]). Relational databases are often SQL based (cf. Date [2011]), so applications are using a relational database management system (or server) such as POSTGRESQL [140] or MYSQL [141], etc etc... Databases are routinely capable to deal with a very large amount of data (e.g. many terabytes or even petabytes), much more than the available RAM, because the application using some database is *explicitly* fetching and updating only a tiny part of it. Hence such persistence is not orthogonal.

Orthogonal persistence (see Dearle et al. [2010]) is defined in Wikipedia as : “Persistence is said to be "orthogonal" or "transparent" when it is implemented as an intrinsic property of the execution environment of a program. An orthogonal persistence environment does not require any specific actions by programs running in it to retrieve or save their state.”.

Since the *Bismon* monitor deals only with the source code of some (perhaps large) IoT firmware and its related internal representations, the total volume of data should easily fit inside the RAM of a high-end workstation. For example, several millions lines of source code makes a large IoT firmware [142] but can be kept in RAM. Hence *Bismon* favors a (nearly) orthogonal persistence. The only action required to keep its state is a full dump of its entire persistent heap. An important insight is the similarity between the depth-first exploration algorithm (see also Cormen et al. [2009]; Christian and Griffiths [2017]) used to dump a persistent state, and classical copying or tracing garbage collection algorithms (such as Cheney [1970], or more recent generational copying GCs, see Jones et al. [2016]).

The persistent state should be considered as precious and as valuable as source code of most software, so it

[137] The object of objid `_1FEnnpEkGdI_5DAcVDL5XHG` should be designed as `BMK_1FEnnpEkGdI_5DAcVDL5XHG` in hand-written C code if it is not predefined, and a special utility collects all such names and generates a table of all these constants.

[138] Some previous experimentation with Boehm's GC in multithreaded settings has been unsatisfactory.

[139] A gross analogy is the textual dump of some SQL database. That dump is the only way to reliably recover the database, so it should be done frequently and the backed-up `database.sql` textual dump can be a large file of many gigabytes.

[140] See **http://postgresql.org/** for more.

[141] See **http://mysql.com/** and its close variant **http://mariadb.org/** for more. Notice also that **http://sqlite.org/** is a library often used to embed a single, simple, relational database inside a program.

[142] Most of the firmware we heard of, even from CHARIOT partners, have less than several hundreds thousands lines of C or C++, which expands to one or two millions Gimple statements. This should fit into one or a few dozens of gigabytes at most, even taking into account that several variants of internal representations are kept.

should be backed-up and probably version-controlled [143] at every site using the *Bismon* monitor.

Notice that in *Bismon* only the *heap* is persisted, but “continuations” or “call stacks” or threads [144] are *not persisted* [145] by themselves.

2.3.1 file organization of the persistent state

The persistent state contains both data and “code”. So it is made of the following files:

- data files `store1.bmon`, `store2.bmon` etc... Each such generated data file describes a (potentially large) collection of persistent objects, and mentions also the modules required for them. There is one data file per space, so `store1.bmon` is for the space#1 (containing predefined objects), `store2.bmon` is for the space#2 (conventionally containing “global” objects useful on every instance of *Bismon*), etc...

- code files [146] contain the *generated* C code of persistent modules (in the `modules/` sub-directory). Since each module is (also) reified by an object representing (and generating) that module, the code file paths contain objids. For example, our object `_3BFt4NfJmZC_7iYi2dwM38B` (it is tentatively named `first_misc_module`, of class `basiclo_dumpable_module`) is emitting its C code in the generated `modules/modbm_3BFt4NfJmZC_7iYi2dwM38B.c` file, so that code file is part of the persistent state.

Ideally, in the future (after end of CHARIOT), the *Bismon monitor* should be entirely bootstapped [147], so all its files should be generated (including what is still in 2018 the hand-coded “runtime” part of *Bismon* such as our `*_BM.c` files, notably the load and dump machinery in `load_BM.c` and `dump_BM.c`, the agenda mechanism in `agenda_BM.c`, miscellanous routines including the support of module loading in `misc_BM.cc`, etc, etc...). However, we are still quite far from that ideal. Existing bootstrapped systems [148] such as CAIA (see Pitrat [2000, 2009a,b], Pitrat [2013-2019][149], OCAML (cf Leroy et al. [2018]; Leroy [2000] and many other papers by Xavier Leroy and the Gallium team at INRIA), CHICKEN/SCHEME[150], SELF (cf Ungar and Smith [1987]), SBCL[151] and CLASP (cf Schafmeister [2015]) show that it is possible. The major advantage of generating *all* the code of *Bismon* would be to deal with internal consistency in some automated and systematic way and facilitate refactoring [152]. An important insight is that the behavior of a bootstrapped system can be improved in two ways: the “source” of the code could be improved (in the case of *Bismon*, all the objects describing some module) and the “generator” of the code could also be improved (cf. partial evaluation and Futamura projections, e.g. Futamura [1999]; also, Pitrat [2009b, 2013-2019] gives some interesting perspectives for artificial intelligence with such an approach).

[143] How and when the persistent state is dumped, backed up and version controlled is out of scope of this report. We strongly recommend doing that frequently, at least several times every day and probably a few times each hour. If the *Bismon monitor* crashes, you have lost everything since the latest dumped persistent store. The textual format of the persisted state should be friendly to most version control systems and other utilities.

[144] The call stack is not formally known to the C11 or C++11 standard (and the optimizing C or C++ compiler[s] which optimize[s] *Bismon* code is permitted to “mess it” during compilation of the *Bismon* runtime). Trying to reify and persist it some “portable” C or C++ code is not reasonable, or would require some very implementation- and architecture- specific and tricky code. Even generated C as a portable assembler don’t know about the call stack.

[145] Some data related to the agenda (cf. §1.7) might be persisted. Persisting continuations is however an interesting research topic (to be worked out outside of CHARIOT).

[146] Conceptually, the `dlopen`-ed shared object files, such as our `modubin/modbm_1zCsXG4OTPr_8PwkDAWr16S.so` (which corresponds to the emitted `modules/modbm_3BFt4NfJmZC_7iYi2dwM38B.c` code file) are *not* part of the persistent state. In practice, these shared object files obviously need to be built before starting *bismon*, since it `dlopen`-s them at initial load time.

[147] This was not completely the case of *GCC MELT*, but almost: about 80% of *GCC MELT* at the end of that project was coded in *MELT* itself. However, it was tied to a particular version of *GCC*.

[148] Observe that an entire Linux distribution is also, when considered as a single system of ten billions lines of source code, fully bootstrapped. You could regenerate all of it. See **http://www.linuxfromscratch.org/** for guidance.

[149] That blog explains that all the 500KLOC of the C code of CAIA and its persistent data are generated.

[150] Notice that *semantically* Bismon is quite close to *Scheme* and shares many features with that language; see **http://starynkevitch.net/Basile/guile-tutorial-1.html** for more. The bootstrapped CHICKEN system is on **https://www.call-cc.org/** and is, like Bismon, translated to C.

[151] See **http://sbcl.org/** for more

[152] In 2018, if we decide painfully to change the representation of attribute associations in objects, we have to modify a lot of hand-written code and objects simultaneously, and that is a difficult and brittle effort of refactoring. If all our code was generated, it would be still hard, but much less.

2.3.2 persisting objects

Obviously, the objects of *Bismon* (§ 2.1.2) may have circular references, and circularity can only happen thru objects (since other composite values such as nodes or sets are immutable, § 2.1.1). So the initial loader of the persistent state proceeds in two passes. The first pass is creating all the persisted objects as empty and loads the modules needed by them, and the second pass is filling these objects.

```
«_9oXtCgAbkqv_4y1xhhF5Nhz |=first_test_module|
!~ name (~ first_test_module ~)
± 1546366846.488
∈ _5bP4nozCTp0_1DPPTK398m1 |=basiclo_dumpable_module|
↦ _3dmcFZldtxI_1bEIFl4jqe3 |=also|
 % _5DDSY1YgVZr_6dOU4tiBldk (  { _6SSe2uyt8Cn_4QNhKM5hsDA _9O2lgu1TweO_0mVlpTwrBG1 })
↦ _4rz1Bi4weGz_8Y03gU81cRL |=see|
 {
 _43Y25VLmh6s_3JRpERevcR4 _6SSe2uyt8Cn_4QNhKM5hsDA _9O2lgu1TweO_0mVlpTwrBG1 }
!# 3

↳ _43Y25VLmh6s_3JRpERevcR4
↳ _9O2lgu1TweO_0mVlpTwrBG1
↳ _0qxuQEfimtp_1Wp2YuKHsJ3
// emitted persistent module modbm_9oXtCgAbkqv_4y1xhhF5Nhz.c
»_9oXtCgAbkqv_4y1xhhF5Nhz
```

Figure 4: generated dump example: `first_test_module` in file `store2.bmon`

The figure 4 shows an example of the textual dump for some object (named `first_test_module`) of objid `_9oXtCgAbkqv_4y1xhhF5Nhz` extracted from the data file `store2.bmon`.

> The lines starting with « or `!(` and with » or `!)` are delimiting the object's persistent representation. Comments [153] can start with a bar `|` till the following bar, or with two slashes `//` till the end of line. `!~` with matching `(~` ... `~)` are for "modifications" (here, we set the `name` of that object to `first_test_module`). Object payloads are also dumped in such "modification" form. `!@` puts the *mtime*. `!$` *classobjid* sets the class to the object of objid *classobjid*. `!:` *attrobj valattr* put the attribute *attrobj* associated with the value *valattr*. `!#` *nbcomp* reserve the spaces for *nbcomp*, and `!&` *valcomp* appends the value *valcomp* as a component. The Σ or `!|` is for function signatures.
>
> Within every dumped object, attributes whose class is `temporary_attribute_object` are *not* dumped. This enables to mix both dumped attributes -as objects these attributes are not indirect instances of `temporary_attribute_o` and non-dumped attributes in the same object.

value	←	*int*	; tagged integers
	|	*float*	; double precision floats
	|	`__`	; nil
	|	`"`*string*`"`	; string with JSON-like escapes
	|	*objid*	; object of given *objid*
	|	`{` $objid_{elem}$ ... `}`	; set of elements of given *objid*
	|	`[` $objid_{comp}$ ... `]`	; tuple of components of given *objid*
	|	`*` $objid_{conn}$ `(` $value_{son}$ ... `)`	; node of given connective and son[s]
	|	`%` $objid_{conn}$ `(` $value_{son}$ ... `)`	; closure of given connective and son[s]

Figure 5: syntax of values in dumped data files.

Data files start first with the *objid* of modules used by routines (in objects mentioned in that data file). These module-objids are prefixed with `!^` or with μ. Then the collection of objects (similar to figure 4 each) follows.

[153] Once the persistence code - loading and dumping of the state - is mature enough, we will disable generation of comments in data files.

[154] That `temporary_attribute_object` class has objid `_23vPTNrGYB`.

In data files, objects are represented by their *objid*, perhaps followed by a useless comment like |*this*|. And immutable values are in the grammar given in figure 5 (where __, representing *nil*, can also appear inside tuples, nodes, closures but not within sets). The *float*-s are dumped [155] with best effort. At dump time, a transient object is replaced as *nil*, so may be dumped as __. Within a set, it is skipped so ignored. When the connective of a node or of a closure is a transient object, that node or closure is not dumped, but entirely replaced by *nil* so dumped as __.

The dump works, in a similar fashion of our naive GC, in two phases: a scanning phase to build the hash-set of all dumped objects. A queue of objects to be scanned is also used. Then an emission phase is dumping them (one data file per object space). A *dumper* transient object -of class `dumper_object` is made to reify the "global" dump status (notably the queue of objects to scan, and the hash-set of dumped objects). So dumping happens by sending messages with selectors -used by the dumping routine sending messages- like `dump_scan`, `dump_value`, `dump_data`. The `dump_scan` message is sent[156] to a scanned object having some payload during the scanning phase. The `dump_value` message is sent[157] is to dumped values during the emission phase. The `dump_data` message is sent[158] when emitting an object content -notably its payload - if relevant-, after having dumped its class, attributes and components. But attributes which are direct or indirect instances of `temporary_attribute_object` are not dumped. Coupling databases with graphical interfaces and GCC plugins have been achieved in Dean [2009].

[155] We don't care about some IEEE-754 double-precision 64 bits floating point numbers not retaining all their significant bits between dump and reload.

[156] The *dumper* object is the additional argument to `dump_scan`.

[157] Additional arguments to to `dump_value` are a string buffer object, the *dumper* object, and the depth as a tagged integer. Notice that dumping of integers is implemented with the `dump_value` method of class `int`, dumping of tuples is implemented with the `dump_value` method of class `tuple`, dumping of object references as objids is implemented with the `dump_value` method of class `object`, etc....

[158] Extra arguments to `dump_value` are: the *dumper* object and the string buffer object.

3 Static analysis of source code in *Bismon*

Static analysis involves a *generated* GCC plugin (whose C++ code is generated by the *bismon* persistent monitor) which communicates with the monitor and sends to it some digested form of the analyzed C or C++ code. Some translation-unit specific processing can happen in that GCC plugin, but the whole program aspects of the static code analysis should obviously be done inside the monitor, and requires -and justifies- its persistence. The complexity and non-stability of *GCC* internal representations justify some semi-automatic approach in extracting them (see §3.1 below).

3.1 static analysis of *GCC* code

The *GCC* compiler has a complex (and ill-defined, under-documented and evolving, so unstable) application programming interface (API) which can be used by plugins. So *Bismon* needs to analyze the various *GCC* plugin related *header files* to extract important information about that API, so to be later able to generate *GCC* plugin code. Such an extraction (inspired by the approach inside *Clasp*, which does similar things with the help of *Clang*, see Schafmeister [2015] for details) needs not to care about the *Gimple* instructions, but only about the abstract syntax tree in *Tree* and *Generic* forms (see GCC Community [2018] §11) to retrieve the full description of *GCC*.

This approach of extracting semi-automatically [159] the GCC API (of parsing GCC header files with some simple GCC plugin) is motivated by past GCC MELT experience (where every feature of the GCC API had to be *explicitly* and manually described in MELT language; these descriptions took a lot of time to be written and had to be manually maintained; however, most of them could in theory be extracted automatically from GCC headers).

A bootstrapping and incremental approach, in several "steps", is worthwhile (and possible because of persistence): we will first extract some very simple information from GCC header files, and use them to improve the next extraction from the same GCC header files. The *slow* evolution [160] of GCC API is practically relevant (most of the API of `gcc-8.3` should stay in the next `gcc-9.0` version).

Descriptive data related to the API of a particular version of GCC will thus stay persistently in the *Bismon* monitor, but should be updated at each release of *GCC*. We care mostly about API related to optimization passes, *GENERIC*, *Gimple*, *SSA* and *Optimized-Gimple*. We probably don't need to go at the *RTL* level. The version 10 of GCC (released in May 2020) incorporates several static analysis options, that are activated with the `-fanalyzer` option to `g++` or `gcc`. The version 10 of CLANG (released in March 2020) contains an improved Clang static analyzer and `clang-tidy` "linter" like tool, check both coding styles and portability related conventions. Both compilers are open source and available [161] on a LINUX desktop and both should be of interest for advanced *IoT* software developers coding in C or in C++ and capable of using the command line. GCC analysis features be configured thru appropriate `#pragma`-s, and CLANG analysis features are changeable by conventional comments such as `// NOLINT(google-explicit-constructor, google-runtime-int)`.

3.2 static analysis of IoT firmware or application code

Once the API of the current version of *GCC* is known to the persistent monitor, we can generate the C++ code of *GCC* plugins for cross-compilers used by IoT developers.

A first static analysis, useful to IoT developers, will be related to whole-program detection of *stack overflow* (see also Payer [2018]). By the way, such an analysis is currently not doable by *Frama-C*, because it don't know the size of each call frame. However, *GCC* is already computing that size (see the `-fstack-usage` option which dumps the size of the call frame of each function, and the `-Wframe-larger-than=`*`bytesize`* option), and we simply need to extract and keep it. We also need to get a good approximation of the *control flow graph*. For that we need to extract basic blocks and just `GIMPLE_CALL` *Gimple* statements (ignoring other kinds of *Gimple* statements). Of course, indirect calls (thru function pointers, which are infrequently used in most IoT code) are harder to handle (and could require interaction with the IoT developer using our monitor, to annotate them).

A proof-of-concept GCC plugin for GCC 8 (and 9) to take advantage of existing internal GCC passes to compute some upper approximation of the call stack size has been developped. That hand-written GCC plugin, coded in file `gcc8plugin-demo-chariot-2019Q2.cc` of about a thousand lines of C++, communicate with the `bismon` monitor using some REST HTTP protocol with ad-hoc HTTP `POST` requests having a JSON payload, in some CHARIOT specific JSON format. The `bismon` monitor should display that diagnostic in a Web browser tab. It could also use the *language server protocol* [162] which is, in 2019, understood by most free software source code editors running on Linux,

[159] We are well aware that some work still needs to be done manually, in particular giving the really useful subpart of the *GCC* API.

[160] GCC internals are *slowly* evolving, because GCC itself is huge: its "navigation" is as slow as that of a supertanker which needs hours to turn and change directions. So for *social* reasons the GCC community is changing the API slowly, but there is no promise of stability.

[161] Both recent GCC and CLANG are buildable as cross compiler for major 32 bits or 64 bits architectures such as PowerPC, x86, x86-64, or ARM, provided one download their source code.

[162] See **`https://langserver.org/`** for more.

including emacs, or vim, or VSCode. It might even later use the new *Sarif*[163] protocol, designed for communication between static source code analyser.

Notice that according to this webpage, nearly 70% of security bugs affecting the Chrome web browser (by Google) are related to memory management issues in C++. It is expected that junior European software developers of non-critical IoT systems would experiment a similar bug distribution in their IoT code. A long-term approach could be the costly training of IoT software engineers to switch to programming languages with better memory management, such as Go or Rust. However, rewriting an entire IoT code base is too costly, and mixing several programming languages[164] in the same software product can be worthwhile but requires some rare and qualified labor. GCC 10 has a new static analysis framework (with its -fanalyzer compiler option) and powerful warning options[165]. The CLANG static analyzer could be useful. Some coding rules (such as Holzmann [2006] or MISRA C) are available, but the Rice's theorem forbids the possibility of a sound and complete static analyzer: false alarms cannot be avoided, and code reviews by senior programmers is still necessary.

We probably would also take as an example the analysis of some MQTT library. The insight is to trust some existing MQTT implementation [166], and to help *junior* developers in using it, by checking simple coding rules relevant to MQTT.

An interesting CHARIOT-compatible approach could be to use *topological data analysis* (cf Chazal and Michel [2017]) techniques, combined with some machine learning (cf Flach [2012]) and big data / data mining (cf Wu et al. [2013]; Clarke [2016]; Zuboff [2015]; Helbing et al. [2019]) approaches, on some of the several directed graphs (notably the *control flow graph*, the *call graph*, the *dependency graph* for example) of the whole analyzed program. Reputable free software libraries[167] are available on Linux. In principle, such an approach might be used in bismon for a *semi-automatic* detection of *code smells*. Sadly, the lack of allocated human resources, and the strong focus (see Héder [2017]) on high TRL[168] results, forbids even trying such an interesting approach in CHARIOT, taking into account that industrial corporations are not even dreaming of it. However, these approaches might be tried in some other projects, perhaps DECODER.

3.3 static analysis related to pointers and addresses

Pointers are an important part of the C11 and C++11 language specifications (see ISO [2011a,b]), but are difficult to understand. They need several chapters[169] in C or C++ textbooks such as Gustedt [2019]; Stroustrup [2014]. Practically speaking, a pointer is an address, with NULL (in C) or **nullptr** (in C++) having a special and distinguished meaning: it is never the same as the result of the *address-of* operator (unary & prefix operator).

From the IoT or firmware developer's point of view, pointers -viewed as addresses- may behave strangely in practice, and differently from the language specifications: in theory, the NULL pointer might not sit at address 0. In practice, IoT or firmware developers do know that (on most implementations) it *is* at address 0. Dereferencing the NULL pointer is the prototypical example of *undefined behavior*, yet some firmware code[170] may do that with good reasons. For example, some AVR microcontrollers (used in ARDUINO platforms "the working registers are mapped in as the first 32 memory addresses" (from the AVR microcontrollers wikipage). The MIPS architecture (used by some CHARIOT partners) has an instruction set without proper I/O ports (see this), so input/output happens by accessing some dedicated memory locations: from the IoT programmer's point of view, physical I/O happens by writing into documented memory locations. The opensource RISC-V architecture , used in IoT (see Lee et al. [2020]; Waterman [2016])) also has memory-based input/output, but admit extensions with separate input/output ports.

Most IoT used operating system kernels (see Arpaci-Dusseau and Arpaci-Dusseau [2015], osdev.org) such as the LINUX kernel (see also kernelnewbies.org and stackoverflow.com), FREEBSD and FREERTOS are managing processes having each their own virtual address spaceand provide the virtual memory abstraction, with some file systems to application code.

[163] See **http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.html** for more.

[164] See this for a discussion of why is that interesting. Notice that recent GCC compilers share some common internal representations between several language front-ends.

[165] It is helpful to pass -Wall -Wextra to the gcc or g++ compiler, usually thru some build automation tool such as ninja

[166] Our purpose is not to prove the correctness of a given MQTT implementation, which would require a formal methods approach à la VESSEDIA, but to help the developer using and trusting it, by checking some specific coding rules.

[167] See TENSORFLOW on **https://www.tensorflow.org/**, and GUDHI on **http://gudhi.gforge.inria.fr/**, and many other similar libraries.

[168] *Technical Readiness Level* and the TRL wikipage for more.

[169] A novice programmer should be explained that after `int tab[4]; int* p = &tab+1;` both `tab[2]` and `p[1]` are pointer aliases, but `sizeof(tab)` is not `sizeof(p-1)` even if `tab == p-1`.

[170] A typical example would be some BIOS or UEFI firmware or operating system kernel on most PC desktop motherboards, see osdev.org and tinyvga.com for more

In most C or C++ source programs, either in some *freestanding environment* (like operating system kernels, low-end embedded software without any `main`, e.g. cheap ARDUINO-like devices) or in a *hosted environment* (so using the standard C or C++ libraries and started thru their `main` function), for example some web server running under LINUX on something similar to a RASPBERRYPI board), dynamically allocated memory is very important in practice. See also Karvinen et al. [2014] and `raspberrypi.org` and `arduino.cc` websites.

In hosted environments, dynamic memory allocation often uses functions like `malloc` or its friends `calloc` with `realloc` and `free`. On some operating systems, it could happen thru lower-level system calls such as `mmap` or `sbrk`. Deallocation would use other system calls such as `munmap`.

Many freestanding environments provide more or less equivalent memory allocation facilities. For instance, the FREERTOS kernel has in some cases a `pvPortMalloc` function with a behavior close to `malloc`, and the the LINUX kernel uses `kmalloc`, with other deallocating functions close in behavior to `free`. But on ARDUINO devices using `malloc` is indeed discouraged.

Dynamic memory allocation is often used but relevant for two important concerns: buffer overflows and call stack management, including avoidance of stack overflow. Call stacks are needed for any multi-threading or multi-tasking approach. Junior developers may easily write code that blows them up (e.g. with a too naive recursion in C, or by having huge automatic variables). A rule of thumb is to avoid needing, in hosted environments, call stacks above a megabyte, or call frames bigger than a few kilobytes. With recent `gcc` compilers, passing something like `-Wstack-usage=2048` to the `gcc` compilation command is practically helpful.

Of course, call stacks are a scare resource, and even on Linux it is generally unreasonable to have many thousands of them in the same process. This explains why *thread*-s are expensive, and why the PTHREAD function `pthread_create` (or C++11 `std::thread`-s) should be used with caution. See also `pthreads(7)`, `nptl(7)` and the source code of GNU GLIBC and of `musl libc`.

Depending on the development efforts (so costs) available and on the criticity of the IoT software, out of memory conditions are handled differently. A lot of C code incorrectly assumes that `malloc` always[171] succeed. In practice, this assumption is generally[172] correct, so for non-critical IoT software[173] it does have some economical sense.

Most non-critical IoT software (e.g. inside consumer devices, perhaps RASPBERRYPI based, or wifi routers based upon OPENWRT router software) won't care about and won't even handle (or even try to cleverly detect and report) rare error conditions or failures such as :

- rare out of memory conditions (e.g. `malloc` failures)
- file system errors (e.g. `fopen` failure on some configuration file under `/etc/`), including I/O errors (e.g. `fscanf` or `fread` failures) due to hardware failure (like malfunctioning USB storage keys).
- floating point precision issues. See the STANCE European project (grant 317753), the `floating-point-gui.de` website, and Kiss et al. [2015]; Goubault and Putot [2011].
- losing time. See the `time(7)` `man` page for an interesting overview. A battery powered real-time clock chip is cheap, but in most IoT consumer devices (typically routers or printers) it is not useful enough, since they are often connected to some distant NTPD service
- synchronization bugs related to multi-threading or concurrent systems, e.g. with POSIX THREADS, See Goubault and Haucourt [2005]; Sangiorgi and Walker [2003]; David et al. [2013]; Guerraoui and Kuznetsov [2018]. Concurrency is also a concern of client-server architectures, including Web services using HTTP , SMTP , IMAP (perhaps mixed with TLS).

Pragmatically good reasons to avoid testing such error conditions include: 1. cost of additional development efforts for extra C code which is difficult to test in a reproducible way without fault injections; 2. scarse code memory space (e.g. in some read-only memory) on cheap devices.

Please notice that above error conditions are related: for example, a loss of time may corrupt an entire file system, and timing information (or meta-data) is quite often kept and needed in small SQLITE databases. .

[171] See this joke-implementation of `malloc`, conforming to the letter but not the spirit of the C standard.

[172] On Linux, be aware of memory overcommit which could be disliked.

[173] As a consumer, I guess that most consumer rented Internet boxes - generally running some Linux - do have some system-wide

The remote backup of such small or large databases (see Date [2005]; Kornacker et al. [2015]) could happen on a periodical basis (see `crontab(5)` and `rsync(1)` for more). Large relational databases are practically needed for most machine learning algorithms (see Flach [2012]), which are usually concurrent programs. So a non-critical machine-learning distributed IoT system would probably be made of cheap IoT devices communicating with some powerful, semi-centralized, database servers. Current meteorological computing grids for numerical weather prediction operated by national meteorological services such as Météo-France require such continent-wide IoT networks, and so does avalanche detection and forecasting in European mountains. Smart cities networks and smart grids (see Belarbi [2004]; McLaren and Agyeman [2015]; Delons et al. [2008]; Bakken [2014] and the GRID4EU project) also need a large grid of many communicating IoT devices.

In *some* even critical IoT devices (e.g. medical devices such as a Covid-19 breathing ventilator), losing power or time for a few seconds is acceptable, as long as there is some hardware alarm (e.g. electronic bell) informing professional users (e.g. medical nurses).

Most non-critical IoT devices could *sometimes* access by the network a remote database , such as some POSTGRESQL server, or some MONGODB document database. Losing the connection to such a remote database is generally affordable, if the connection loss don't last too long. On the other hand, a posteriori adding database checkpoint facilities to some existing large long-running scientific code (think of oil industry simulation software, or digital twins for automotive crash simulations, both running for months of super computer time) requires some significant development efforts or code refactoring. Of course European particle accelerators such as CERN ELENA or LHC installations are deploying wide networks of heterogenous IoT devices, but probably can afford partial failure of some of them. Cyber-attacks on smart grids (e.g. Lee et al. [2016]) could justify an increase of European research funding on long-term mixed static analysis and dynamic machine learning based techniques, but require funding of projects with a time span above three years and involving tight cooperation thru open source projects (see Brooks [1995]; Lerner and Tirole [2000]; Tirole [2018]; Hashem et al. [2015]). See also the related SOFTWAREHERITAGE project. See Brook's law and observation -in 1975- that "while it takes one woman nine months to make one baby, "nine women can't make a baby in one month"" (to be scaled in 2020 by a factor of 10x and generalized to software developers of both genders, given the complexity of current software intensive systems).

Notice that exact static prediction of call stack depth is in practice impossible, because of Rice's theorem -and usage in C code of the `alloca(3)` stack allocation primitive, or of variable-length arrays - and since compilers may put automatic variables in processor registers. For C or C++ compilers handling the `asm` keyword (in recent GCC compilers, it provides even powerful language extensions to C or C++), register allocation becomes a nightmare for the compiler writer, since the `register` keyword tend to become obsolete or ill-defined (like the `volatile` one). So it is strongly tied to the register allocation issue, which is a difficult sub-problem inside most optimizing compilers (see Eisl et al. [2016]; Chaitin et al. [1981]; Aho et al. [2006]). . In recent C++ language dialects, the `auto` and `decltype` keywords are enabling some *type inference* (see Pierce [2002]; Stroustrup [2014, 2020]; ISO [2011b]). In practice, industrial compilers don't even try to find the best register allocation possible, but just a good enough one (otherwise compilation time could be prohibitive). Be aware that flexible array members are also a C language feature, which, combined with pointer casts (or equivalently used in `union` of pointers), permit compilers to do arbitrarily sophisticated optimizations.

memory leaks, and that observation could explain why rebooting them weekly improve the user experience.

4 Using *Bismon*

This section §4 should become somehow a user manual, and will be written for the final D1.3^{v2}. It is both for the ordinary IoT developer just using *bismon* for static analysis of IoT source code, and for the static analysis expert configuring and programming it.

Most of that should be generated from data persisted inside *bismon*. Perhaps should be exchanged with the "static analysis" chapter (§3).

4.1 How JSON is used by Bismon

The JSON[174] textual format is a convenient, common and compact structured textual format. It is used in *Bismon*, in particular because of its web interface, and supported as a payload (but not directly[175] as an immutable value) for objects of class `json_object`.

Conceptually, the JSON model is close, but not identical to, the Bismon persistent model: it provides structured and compositional constructs, but JSON objects have *strings* as attributes, while Bismon objects have arbitrary object references as attributes, and also components and some optional payload.

4.1.1 The canonical *JSON* encoding of *Bismon* values

Therefore, there is some way to encode any *Bismon* value into a JSON value; this is the *canonical JSON encoding of values*, given in table 11 below.

$[\![\mathrm{nil}]\!]_{json}$	$\rightarrow$	**`null`**	The Bismon nil is encoded as the JSON null
$[\![\mathrm{unspecified}]\!]_{json}$	$\rightarrow$	**`false`**	The Bismon *unspecified* is encoded as the JSON false
$[\![\mathrm{integer}\ i]\!]_{json}$	$\rightarrow$	i (JSON integer)	tagged integers encoded as is
$[\![\mathrm{boxed\ float}\ f]\!]_{json}$	$\rightarrow$	f (JSON float)	boxed doubles encoded as is, with decimal point
$[\![\mathrm{string}\ s]\!]_{json}$	$\rightarrow$	s (JSON string)	Bismon strings encoded as is
$[\![\mathrm{object}\ \omega\ \text{of objid}\ oid]\!]_{json}$	$\rightarrow$	**`{ "!oid" :`** *oid* **`}`**	Bismon objects encoded with `"!oid"` JSON attribute giving the objid as a JSON string
$[\![\mathrm{tuple}\ [\omega_1 \ldots \omega_n]\ ]\!]_{json}$	$\rightarrow$	**`{ "!tup" :`** `[` $oid_1 \ldots oid_n$ `]` **`}`**	Bismon tuples encoded with `"!tup"` JSON attribute giving the JSON array of corresponding objid JSON strings : $oid_i = \mathrm{objid}(\omega_i)$
$[\![\mathrm{set}\ \{\omega_1 \ldots \omega_n\}\ ]\!]_{json}$	$\rightarrow$	**`{ "!set" :`** `[` $oid_1 \ldots oid_n$ `]` **`}`**	Bismon sets encoded with `"!set"` JSON attribute giving the JSON array of corresponding objid JSON strings : $oid_i = \mathrm{objid}(\omega_i)$
$[\![\mathrm{node}\ {*}\ \omega_{conn}(\sigma_1 \ldots \sigma_n)\ ]\!]_{json}$	$\rightarrow$	**`{ "!node" :`** oid_{conn} **`,`** **`"!sons" :`** `[` $[\![\sigma_1]\!]_{json} \ldots [\![\sigma_n]\!]_{json}$ `]` **`}`**	Bismon nodes encoded with `"!node"` JSON attribute giving the objid $oid_{conn} = \mathrm{objid}(\omega_{conn})$ of the connective oid_{conn}, and with `"!sons"` JSON attribute associated to the array of encodings of that node's sons σ_i
$[\![\mathrm{closure}\ \%\ \omega_{rout}(\kappa_1 \ldots \kappa_n)\ ]\!]_{json}$	$\rightarrow$	**`{ "!clos" :`** oid_{rout} **`,`** **`"!cval" :`** `[` $[\![\kappa_1]\!]_{json} \ldots [\![\kappa_n]\!]_{json}$ `]` **`}`**	Bismon closures encoded with `"!clos"` JSON attribute giving the objid $oid_{rout} = \mathrm{objid}(\omega_{conn})$ of the closure's routine, and with `"!cval"` JSON attribute associated to the array of encodings of that closure's closed values κ_i

Table 11: canonical JSON encoding $[\![v]\!]_{json}$ of a Bismon value v.

The canonical JSON encoding is implemented[176] as the `canonjsonifyvalue_BM` function.

[174] See **`http://json.org/`** for more
[175] Adding immutable JSON values as a new kind of Bismon value could be considered in the future.
[176] Coded in C, in file `jsonjansson_BM.c`

4.1.2 The nodal *JSON* decoding into *Bismon* values

Since JSON is a structured and compositional, tree-like, representation, and because nodes are the only kind of structured immutable *Bismon* values, any JSON value can obviously be decoded into a *Bismon* values, using mostly nodes for structuring data, following the rules listed in table 12 below. Actually, there are several variants of nodal decodings, depending on how JSON strings looking like full objids (e.g. JSON `"_756o00yB7Zs_1USbaS25hxl"`), or abbreviated objids (e.g. JSON `"_9Z2BgJbf4"`), or named objects (e.g. JSON `"arguments"`, related to Bismon object `arguments`, i.e. `_0jFqaPPHg`) are really decoded.

$\langle \texttt{null} \rangle^{nod}$	$\rightarrow$	**`json_null`**	The JSON `null` is nodal-decoded as the `json_null` Bismon object `_6WOSg1mpN`
$\langle \texttt{false} \rangle^{nod}$	$\rightarrow$	**`json_false`**	The JSON `false` is nodal-decoded as the `json_false` Bismon object `_1h1MMlmQi`
$\langle \texttt{true} \rangle^{nod}$	$\rightarrow$	**`json_true`**	The JSON `true` is nodal-decoded as the `json_true` Bismon object `_0ekuRPtKaI`
$\langle \text{integer } \iota \rangle^{nod}$	$\rightarrow$	tagged integer ι	The JSON integers are nodal-decoded as the corresponding *Bismon* integer
$\langle \text{real } \delta \rangle^{nod}$	$\rightarrow$	boxed double δ	The JSON reals are nodal-decoded as the corresponding *Bismon* boxed double
$\langle \text{objid-looking string } \sigma \rangle^{nod}$	$\rightarrow$	object ω, when $\omega \notin$ {**`json_null`**, **`json_false`**, **`json_true`**, **`json_array`**, **`json_object`**}	an objid-looking string σ, starting with an underscore _ and matching the objid of an existing object, *may* be nodal-decoded as *the existing anonymous* object ω such as $\text{objid}(\omega) = \sigma$ when that ω is not a special object mentioned here.
$\langle \text{name-looking string } \sigma \rangle^{nod}$	$\rightarrow$	object ω, with $\omega \notin$ {**`json_null`**, **`json_false`**, **`json_true`**, **`json_array`**, **`json_object`**}	a name-looking string σ, starting with a letter and naming an existing object, may be nodal-decoded as the *existing named* object ω such as $\text{name}(\omega) = \sigma$ when that object is not special.
$\langle \text{any string } \sigma \rangle^{nod}$	$\rightarrow$	*Bismon* string σ	a string σ *would* otherwise be nodal-decoded as is into the same *Bismon string* σ
$\langle [js_1, \ldots js_n] \rangle^{nod}$	$\rightarrow$	$\star$**`json_array`** $(\langle js_1 \rangle^{nod} \ldots \langle js_n \rangle^{nod})$	A JSON array is compositionally nodal-decoded into a node of connective `json_array` and sons given by nodal-decoding the components of that array
$\langle \{\alpha_1 \mathbf{:} js_1, \ldots \alpha_n \mathbf{:} js_n\} \rangle^{nod}$	$\rightarrow$	$\star$**`json_object`**($\star$**`json_entry`**$(\langle \alpha_1 \rangle^{nod}$, $\langle js_1 \rangle^{nod})$ $\vdots$ $\star$**`json_entry`**$(\langle \alpha_n \rangle^{nod}$, $\langle js_n \rangle^{nod})$)	A JSON object is compositionally nodal-decoded into a node of connective `json_objects` and sons given by `json_entries` subnodes whose first son is a string α_i or object ω_i or node $\star$ **`id`**(ω_i) where $\alpha_i = \text{objid}(\omega_i)$ or node $\star$ **`name`**(ω_i) where $\alpha_i = \text{objname}(\omega_i)$

Table 12: nodal JSON decoding $\langle js \rangle^{nod}$ of a JSON value js.

[177] Our JSON extraction is inspired by some pattern matching constructs on linear patterns with semi-unification.

extractor node	success condition	side effects and explanation
`* json_null ()`	$js \equiv$ `null`	succeeds when *js* is the null JSON
`* json_false ()`	$js \equiv$ `false`	succeeds when *js* is the false JSON
`* json_true ()`	$js \equiv$ `true`	succeeds when *js* is the true JSON
`* int` (v^{int})	$js \equiv$ some JSON integer i	succeeds when *js* is a JSON integer i, and assign to integer variable v its integral value $v \leftarrow i$
`* double_float` (v^{val})	$js \equiv$ some JSON real δ	succeeds when *js* is a JSON real δ, and assign to double variable v its *boxed* double value $v \leftarrow$ boxed-double(δ)
`* string` (v^{val})	$js \equiv$ some JSON string σ	succeeds when *js* is a JSON string σ, and assign to value variable v its *boxed* string value $v \leftarrow$ boxed-string(σ)
`* id` (v^{obj})	$js \equiv$ some JSON string σ $\wedge\ \exists$ object $\omega, \text{objid}(\omega) = \sigma$	succeeds when *js* is some JSON string σ looking like an objid, and that string σ is the *objid* of some *existing* object ω, then assign to object variable v that object ω, so $v \leftarrow \omega$
`* name` (v^{obj})	$js \equiv$ some JSON string σ $\wedge\ \exists$ object $\omega, \text{objname}(\omega) = \sigma$	succeeds when *js* is some JSON string σ looking like a name, and that string σ is the *objname* of some *existing* object ω, then assign to object variable v that named object ω, so $v \leftarrow \omega$
`* member` $(x^{\text{val}}\ v^{\text{obj}})$ with x being some Bismon set	$js \equiv$ some JSON string σ $\wedge\ \exists$ object $\omega \in x$, $(\text{objname}(\omega) = \sigma$ $\vee \text{objid}(\omega) = \sigma)$	succeeds when *js* is some JSON string σ looking like a name or an id, and that string σ is the *objname* or *objid* of some *existing* object ω member of set x, then assign to object variable v that object ω, so $v \leftarrow \omega$
`* put` (ω)	*always succeeds*	then put in that given Bismon object ω a JSON payload with *js* inside

Notation: The χ, χ', χ_i... are extractors; the js, js', js_i... are JSON values; $\chi \rhd js$ means that the extraction using extractor χ on JSON *js* was successful; the v, v', v'', v_i, w,... are *Bismon* local variables whose type is explained with blue annotations like $^{\text{val}}$; the x, y... are Bismon values, the ω,... are Bismon objects.

Table 13: simple extraction from some JSON thing *js*; (see also table 14 below.)

4.1.3 *JSON* extraction with `extract_json`

A *Bismon* statement of class `basiclo_extractjsonstmt`, obtained with the `^extract_json (` *objson extractornode sub-statements...* `)` readmacro, so having the attributes `json_object :` *objson* and `extract_json :` *extractornode*, can extract data from the JSON in a payload (of some object *objson*, usually of class `json_object`). That extraction[177] is driven by the given *extractornode* in the statement, which should be a node, as explained in tables 13 (for simple extractions) and 14 (for more complex extractions) below.

Such a JSON extraction is compositional, has side-effects (e.g. could set local variables), and could fail. When the extraction has succeeded, the given *sub-statements...* are executed (they are the components of that JSON extraction statement). No backtracking occurs during extraction.

For example, a JSON thing[178] like `{ "do": "foo" , "obj" : "_3d9rq9TD6PH_6qUv4Hao767", "targets" : [ "_3dmcFZ1dtxI_1bEIF14jqe3", "_6vMEdC0qxdp_7shHwvOZEQx" ] }` could be extracted with the composite extractor `*json_object(*json_entry("do" *string(v_str)) *json_entry("obj" *id(o_comp)) *json_entry("targets" *set(v_set)))` successfully extracting the value variable `v_str` assigned (as a side effect) to the boxed string `"foo"`, the object variable `o_comp` assigned to object `_3d9rq9TD` (that is, the object named `a_to_c`) and the value variable `v_set` assigned to set `{ _3dmcFZ1dt _6vMEdC0qxd }` that is to `{also and}`.

[178] We may call JSON things what the JSON standard call JSON values; and that example could be inside some AJAX or REST POST HTTP request.

extractor node	**success condition**	**side effects and explanation**
`* set` (v^{val})	$js \equiv$ some JSON array `[` js_1`,` $\ldots js_n$ `]` of strings, so $js_1 =$ string $\sigma_1 \wedge \ldots \wedge js_n =$ string $\sigma_n \wedge \forall i, \exists$ object ω_i, (objname$(\omega_i) = \sigma_i \vee$ objid$(\omega_i) = \sigma_i$)	succeeds when js is some JSON array of JSON strings $js_i =$ string σ_i, then builds a Bismon set of *existing* objects $v_{set} = \{\ldots\omega_i\ldots\}$ from these, with $\sigma_i \rightarrow \omega_i$ such that objname$(\omega_i) = \sigma_i$ or objid$(\omega_i) = \sigma_i$
`* json_array` $(\chi_1 \ldots \chi_n)$	$js \equiv$ some JSON array `[` js_1`,` $\ldots js_n$ `]` $\wedge\ \chi_1 \rhd js_1\ \wedge\ \ldots\ \chi_n \rhd js_n$	succeeds when js is a JSON array of ***exactly*** n *elements* : $js_1, \ldots js_n$, and extracts them in sequence, using χ_1 on js_1, then χ_2 on js_2, etc... and at last χ_n on js_n
`* json_entry` $(\alpha\ \ \chi)$	$js \equiv$ some JSON object having $\exists i,$`{` $\ldots \alpha$ `:` $js_i \ldots$ `}` $\wedge\ \chi \rhd js_i$	succeeds when js is a JSON object having an attribute string α associated to some JSON value js_i which can be extracted using χ.
`* json_entry_object` $(\omega\ \ \chi)$	$js \equiv$ some JSON object having $\exists$ some Bismon object ω $\exists$ some JSON string α, (objid$(\omega) = \alpha \vee$ objname$(\omega) = \alpha$) $\exists i,$`{` $\ldots \alpha$ `:` $js_i \ldots$ `}` $\wedge\ \chi \rhd js_i$	succeeds when js is a JSON object having an attribute string α which is the name or objid of the given Bismon object ω, associated to some JSON value js_i which can be extracted using χ.
`* json_object` $(\chi_1 \ldots \chi_n)$	$js \equiv$ some JSON object of length exactly n such that $\chi_1 \rhd js\ \wedge \ldots\ \wedge\ \chi_n \rhd js$	succeeds when js is a JSON object of ***exactly*** n *members* : `{` α_1 `:` $js_1, \ldots\ \alpha_n$ `:` js_n `}`, and that same object js can be extracted by each of the $\chi_1 \ldots \chi_n$ in sequence; the sub-extractors χ_i are often but not always of form `* json_entry` $(\alpha_i\ \ \chi'_i)$, ...
`* json_value` (v^{val})	always succeeds	assign $v \leftarrow \langle js \rangle^{nod}$ to value variable v the *nodal-encoding* of js
`* value` (v^{val})	$\exists x$ Bismon value, $js \equiv [\![x]\!]_{json}$	succeeds when js is the minimal canonical encoding of some value x; then assign $v \leftarrow x$ to value variable v that Bismon value x
`* when` $(\epsilon\ \ stmt_1\ \ \ldots\ \ stmt_n)$	The testing expression ϵ evaluates as true (non-nil value, or non-zero integer)	succeeds when expression ϵ is true (it usually involves some variables computed by previous extractors), then executes in sequence the statements $stmt_1 \ldots stmt_n$ for their side effects
`* and_then` $(\chi_1\ \ \ldots\ \ \chi_n)$	sequentially and lazily ($\chi_1 \rhd js\ \wedge$ $\vdots$ $\wedge\ \chi_n \rhd js$)	succeeds when the same js can be successfully extracted by *each* of the $\chi_1 \ldots \chi_n$ in sequence from left to right (failing lazily as soon as some χ_j fails), and combine their side effects (visible from χ_j to the next χ_{j+1}).
`* or_else` $(\chi_1\ \ \ldots\ \ \chi_n)$	sequentially and lazily ($\chi_1 \rhd js\ \vee$ $\vdots$ $\vee\ \chi_n \rhd js$)	succeeds as soon as the same js can be successfully extracted -from left to right- by *some* of the χ_i (continuing lazily to the next χ_{i+1} when it fails), and combine their side effects (visible from χ_j to the next χ_{j+1}).

Notation: Share notation with table 13 above. In addition, the ϵ are Bismon expressions. The α are JSON strings in attribute position. The σ are strings (usually JSON strings). The $stmt$ are Bismon statements.

Table 14: complex extraction from some JSON thing js; (see also table 13 above.)

4.2 Web interface internal design

The Web interface of *bismon* is supposed to be used without malice (see §1.4.3 and §1.6.1), with a *recent* graphical web browser [179] using HTTP/1.1. In particular, *bismon* does not take any measure against denial-of-service attacks, since it is supposed to be used on a trusted and friendly corporate intranet or local area network, not directly on the wild Internet. The network administrator running *bismon* could deploy usual relevant techniques (firewalls, `iptables`, HTTP proxying, DMZ, etc ...) to avoid such attacks. In practice, there are few web browsers - so few HTTP clients - interacting with *bismon* simultaneously : only a dozen of people in some IoT development team, and each uses his/her graphical browser - a *recent* Firefox or Chrome [180]. Compatibility with other browsers[181] is not yet a concern, given the low TRL ambitioned. Each *Bismon* user is expected to have one, or only a few, browser tab[s] interacting with the `bismon` server, and these tabs, if there are more than one, are handled as different web browsers so have different web sessions. They are physically and geographically located on the same local area network as the machine running the *bismon* monitor. So, from web technologies perspective, *bismon* is making different trade-offs [182] than "traditional" web servers or web applications : the web browser ↔ *bismon* web server round-trip transmission time is supposed to be very small so frequent AJAX requests are possible, the bandwidth is expected to be quite large so voluminous HTTP responses are acceptable, the number of simultaneous web connections or of web sessions is tiny. Therefore most web optimizations are practically unneeded.

With its initial (and current, in mid-2019) naive stop-the-world garbage collector, the interactive performance and user experience (i.e. user look-and-feel) of *Bismon* is expected to be unsatisfactory (since that GC could "block" the `bismon` monitor and web service for more than half a second - during which the web interface stays unresponsive, if running the GC on a large enough heap; but see footnote 71 suggesting an improvement). With significant work, that could be improved.

Each HTTP request either corresponds to a "static" file path under `webroot/` (for a `GET` or `HEAD` HTTP request) or else it is handled dynamically. For a static file path, that file is served directly by the routine `onion_handler_export_local_new` with a `Content-Type` corresponding to its suffix; for example an HTTP `GET` request of `favicon.ico` is answered with the content of `webroot/favicon.ico` file, and an HTTP `GET` request of `jscript/jquery.js` is served by the content of `webroot/jscript/jscript.js`. Care is taken [183] to avoid serving any static file outside of `webroot/`. So the `webroot/` directory contains static content such as images, external JavaScript libraries, CSS stylesheets, etc... Static content requests are always handled the same, so they work even without any cookies.

Any HTTP request which cannot be handled as a static resource like above, because it has no corresponding file under `webroot/`, is considered as a request for dynamic content and is called a *dynamic request*. Dynamic content requires a web session cookie named `BISMONCOOKIE` which contains [184] a cryptographic quality hash (in practice unforgeable) and mentions the objid (cf §2.1.2) of some web session object. If there is no cookie, or if the cookie is invalid or wrong (e.g. forged), a login form is returned. So any HTTP request for a dynamic content (that is which is not handled as a static resource like above) is rejected (with HTTP status `403 Forbidden`) if the user (a contributor in *bismon* parlance, cf. §1.6.3) is not logged in.

Dynamic requests are reified as very temporary *bismon* objects of class `webexchange_object`. Their

[179] Such as Firefox 60.7 or later, or Google Chrome 75.0 or later, both exist in 2019Q2.

[180] So no particular effort is even taken to support a variety of old browsers: we don't have any code to e.g. support Internet Explorer pecularities or deficiencies. Likewise, scalability to thousands of simultaneous HTTP connections is out of scope in *bismon*, but it is essential in most web applications.

[181] See **https://caniuse.com/** and **https://developer.mozilla.org/** and elsewhere for the many subtle but present compatibility issues of current Web technologies, and notably **https://stackoverflow.com/q/33540051/841108** or **https://softwareengineering.stackexchange.com/q/393837/40065** for some tricky questions directly relevant to our *Bismon* development.

[182] For example, we could accept making some HTTP exchange - e.g. with AJAX - on *every* keystroke on the keyboard, but such practice won't be acceptable in usual web services. Also, we don't care much about minimizing the HTTP exchanges - no "minification" needed in practice!

[183] In particular, any HTTP request containing `..` is rejected.

[184] A practical example of `BISMONCOOKIE` value might be `n000041R970099188t330716425o_6IHYL1fOROi_58xJPnBLCTe`: 41 is the serial number counting web sessions in the running *bismon* process, 970099188 and 330716425 are two random numbers, `_6IHYL1fOROi_58xJPnBLCTe` is the randomly-generated objid of the web session object.

web exchange payload [185] contains not only a string buffer [186], to be filled with the HTTP response content, but also mentions the web request (as processed by libonion) and the web session object computed from the BISMONCOOKIE and may contain some arbitrary data value. The web exchange object is supposed to be filled -like string buffers are- and at last given some integer HTTP status and immediately sent back to the browser. Their web session object is created at web login time and is of class websession_object [187]. It knows the contributor who is logged in, the expiration time of the session, some session data (an arbitrary *bismon* value; of course more data can sit in attributes or in components of that web session object), and the web socket connection (if any) to the browser using that session. The session storage [188] associated to key "bismontab" identifies and gives the tab number in the browser. An inactive web session expires in about an hour [189].

Of course, web request objects or web session objects are transient and are not and should not be persisted at dump time (cf. §2.3). So after each restart of the *bismon* monitor, its web users (i.e. contributors) should login again.

A dynamic request is handled by some closure and should be answered in a couple [190] of seconds; otherwise a web timeout occurs. That web handler closure is applied to the remaining URL path string and to the web exchange object created in the libonion-specific thread dealing with the HTTP request, so outside of the agenda machinery (cf §1.7), and usually would add some tasklet into the agenda. Most of the time, a fraction of a second later, some other tasklet would complete the filling the web request object and give some HTTP status code such as 200 OK, then an HTTP reply is sent back to the browser. If a timeout occurs because the web request object has not been taken care of quickly enough, an HTTP 500 Internal Server Error is given back to the browser and that web request object is cleared.

The mapping between URL paths (or prefixes) and web handler closures handling dynamic requests for them is given by the webdict_root[191] dictionnary predefined object of class webhandler_dict_object; for an empty path in the URL (such as http://localhost:8086/ for example), its web_empty_handler attribute is [192] used. If finally no web handler closure is found, an 404 Not found status is returned. The the_web_sessions predefined object stores the dictionnary of transient web session objects and associates a cookie string to its web session object. That dictionnary is forcibly cleared at start of the web server inside *bismon*, but it should be loaded empty, since web session objects are created and should remain transient. A class temporary_webhandler_dict_object, sub-class of the webhandler_dict_object class, also exists and have *transient dictionnary entries* which are not dumped.

In practice, dynamic requests are usually generating the HTML5 content very dynamically. For generated HTML, it is much easier to produce XHTML5, the XML variant of HTML5, because its textual syntax is [193] much more regular and easier to generate than with plain HTML5.

The webxhtml_module in *bismon* has code to ease the emission of XHTML5. And XHTML5 fragments are emitted by the emit_xhtml routine object [194]. That emit_xhtml, which get as arguments: the value

[185] There is no programmatic way to create such a web exchange payload. It can only be created by processing such dynamic HTTP requests.

[186] Since a string buffer should contain valid UTF-8 string content without nul bytes, this restriction forbids binary contents in HTTP replies to dynamic requests. Hence, dynamically computed image contents are not possible, unless they use a textual format like SVG.

[187] So the only way to create a web session payload is thru the login form. There is no programmatic way to create it.

[188] See https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage for more.

[189] See the USER_WEBSESSION_EXPIRATION_DELAY constant in web_ONIONBM.c

[190] See the WEBEXCHANGE_DELAY_BM constant in file web_ONIONBM.c ...

[191] For example: an URL like http://localhost:8086/show/status is handled by some *bismon* monitor listening HTTP requests on port 8086 and with webdict_root associating the string "show" to some closure κ, that web handler closure κ would be applied to the suffix string "status" and to the web exchange object ω created for that HTTP request. The result of that application is ignored, only side effects -often adding some tasklets into the agenda, and/or filling the web exchange object with some XHTML5, etc...- are useful. If the string in such a web handling dictionnary w_{dict} is associated to some other object ω_{wh} of class webhandler_dict_object, that dictionnary object ω_{wh} is recursively explored with the rest of that URL path (e.g. "status" in our example). If w_{dict} has some and_then attribute associated to an object $\omega_{andthen}$ which is a web dictionnary object, that $\omega_{andthen}$ is explored with an incremented depth.

[192] Since dictionnary objects map *non-empty* strings to non-nil values (cf. §2.1.2).

[193] For instance, within a <script HTML5 element containing JavaScript, it is not even allowed in HTML5 to have if (x < 5) even if ordinary HTML rules suggest to use < instead of < in textual content... That makes compositional generation of mixture of HTML and JavaScript emitting HTML much harder.

[194] So that emit_xhtml is, like PHP, a machinery to emit arbitrary XHTML. However, we want to avoid thinking -like PHP was originally designed- in terms of emitting a stream of characters, and emit_xhtml is supposed to emit structured XHTML from some structured, tree-like, internal representation. That internal representation is a DAG (directed acyclic graph).

v_html to emit; an arbitrary web context object o_emitctx which might be in simple cases just some web session object; a string buffer object o_strbuf which is often the web exchange object; the tagged integer recursion depth v_depth, which is in general *not* [195] the emitted indentation. When the string buffer is too full or the recursion depth is too deep, that emit_xhtml fails. When the emitted HTML-reifying value v_html is nil, nothing happens. When it is a scalar, it is emitted trivially: a string is emitted HTML-encoded (so < for <, etc...); a tagged integer is emitted in decimal notation; When v_html is an object ω_{html}, it is emitted per the following rules:

- the newline object emits an indented newline.
- the nlsp object emits a newline when the current line is long enough, or else a space.
- the space object emits a space character.
- instances of html_void_element_object emit some void element like e.g. <hr class='foo'/> using the _0FRLxSGQ1Z routine. The emit_xhtml_open selector should emit -as a side effect- the opening tag <hr class='foo' without the ending />, and returns the string naming the tag, e.g. "hr".
- instances of html_element_object emit recursively using _5NH940lCHYJ some nested XHTML element starting with a start tag like <div but ending with an end tag like </div>; the components of that objects are emitted recursively (with an incremented recursion depth). The spacing style is first determined by sending html_spacing with the o_emitctx and the depth. It can be newline for indented, newline separated, content; or nlsp for space or newline separated, unindented content; or space for space separated content; any other spacing style -notably nil- don't emit any separators in the content. The start tag is emitted with emit_xhtml_open returning the tag string like before (e.g. "span" for a <span class='somecl' emission, then > is output to end the opening tag, then the components, then the end tag is emitted, using the returned tag string from emit_xhtml_open.
- instances of html_sequence_object emit recursively their components but without surrounding tags.
- instances of html_active_object emit recursively HTML stuff thru a message of selector emit_xhtml sent to them.
- any other object is emitted by its name, if it has some, or by its objid. This is mostly intended to represent common repeated names or words by a single and shared object.

When v_html is a node of connective ω_{conn}, it is emitted per the following rules:

- if ω_{conn} is one of int, hexa, octa and v_html is an unary node with a integer son n, that integer n is emitted in decimal, hexadecimal, octal respectively.
- if ω_{conn} is id [196] and v_html is an unary node with an object son ω_{son} its objid is emitted.
- if ω_{conn} is buffer and v_html is an unary node with an object son ω_{son} which has a string buffer payload, that string is emitted HTML-encoded.
- if ω_{conn} is object and v_html is an unary node with an object son ω_{son} its name or objid is emitted.
- if ω_{conn} is name and v_html is an unary node with an object son ω_{son} its name is emitted, and when ω_{son} is not a named object, we have a failure. If ω_{conn} is name and v_html is an binary node with an object first son ω_{son}, and some arbirary non-nil second son ϵ its name is emitted, and when ω_{son} is not a named object, the ϵ is recursively emitted

[195] In very simple cases, without closures or sequences in the DAG of emitted values, the depth could be the depth of XHTML elements, so could be the indentation. In general, it is not.

[196] So nodes of connective id, object or name can be used to emit objects of class html_void_element_object, html_element_object, html_sequence_object, html_active_object which would be handled specially otherwise.

- if ω_{conn} is `sequence`, every son of `v_html` is emitted in sequence, with an incremented $depth$. Nothing is additionally emitted between them.
- if ω_{conn} is `space`, or `newline`, or `nlsp`, every son of `v_html` is emitted in sequence, with an incremented $depth$. Between each son, a space (respectively, a newline, or a smart space of newline) is emitted.
- for any other object ω_{conn} as connective, we extract its `emit_xhtml_node` attribute $v_{emit\ node}$ and its `emit_xhtml_connective_open` attribute $v_{emit\ open}$. Only one of them should be present (non-nil value) and it should be a closure. If $v_{emit\ node}$ is given, it is applied to ω_{conn} `o_emitctx o_strbuf` $depth + 1$ `v_html`, else if $v_{emit\ open}$ is present, we apply it (like the `emit_xhtml_open` selector above) to ω_{conn} `o_emitctx o_strbuf` $depth + 1$ `v_html` to obtain an XHTML element tag. We also extract and use its `html_spacing` attribute. Then proceed like for `html_element_object` using the sons as components....
- @@ to be completed a lot.

When `v_html` is a closure, it is applied @@ to be completed and the result of that application is recursively emitted. When `v_html` is a sequence (set or tuple), its components are emitted recursively.

If no rule is applicable, `emit_xhtml` fails.

The web session objects are also used for *WebSockets* with the following additional conventions. The `bismon` server uses WebSockets only for asynchronous communication from that `bismon` server to Web browsers [197]. The WebSocket messages from `bismon` to web browsers are arbitrary *JSON* values.

The *syntax-oriented editor* (inspired by MENTOR: Donzeau-Gouge et al. [1980]) needed[198] in *Bismon* by its homoiconicity , and used by the static analysis expert, won't handle lines of tokens in an editor widget[199], but should enable that expert user to enter and *conveniently* manipulate the *abstract syntax tree* (or AST) of our *Bismon* DSL. That AST should appear in the web browser as some "graphical" tree (e.g. displayed as SVG elements or in some Canvas element) and/or more probably as nested, but with visible borders, HTML5 markup elements updated thru the DOM. AST manipulation should be easy, e.g. using contextual menus and/or keyboard function keys like F1, F2 and/or control keyboard sequences -with e.g. the Ctrl or Esc keys- inspired by those of `emacs` or `vim`, etc... The set of elementary AST user-doable manipulations -including copy and pasting of AST sub-trees- should be carefully designed, but small. The design insight could represent these ASTs both in user's browser as DOM elements and inside *Bismon* as AST transient objects and/or nodes.

4.3 Using `bismon` for CHARIOT

To run the `bismon` monitor for CHARIOT related activities, that monitor should initialize its state for these activities. So you need to pass `-i init_chariotdemo` as a program argument when running `bismon` in that case.

[197] So web browsers don't communicate *asynchronously* with the `bismon` server. For such communications from browser to `bismon`, Web browser always use synchronous HTTP requests, e.g. using AJAX techniques.

[198] See also end of 1.6.1 above, and also **https://stackoverflow.com/a/47116008/841108** for more.

[199] Of course, using **https://codemirror.net/** will be useful to show analysed C or C++ source code, but not for our *Bismon*'s DSL.

5 Miscellanous work

5.1 Contributions to other free software projects

This is related to subtask ST1.3.2 of CHARIOT GA.

5.1.1 Aborted contribution to `libonion`

The `libonion` library is a free software HTTP server library (LGPLv3 licensed) that is used in `bismon` for its web service feature. See its web site **https://www.coralbits.com/libonion/** for a description, and its source repository **https://github.com/davidmoreno/onion** for more.

The handling of `SIGTERM` signal (and others) is deemed unsatifactory. See the opened issue 229 (on **https://github.com/davidmoreno/onion/issues/229**) in `libonion`. We discussed that issue on google group with the *libonion* community, and came to a disagreement (our design was considered too complex, but we believe that such a complexity is needed to avoid bugs in the rare cases of a multi "`onion`" application, which `bismon` is not).

Independently of that issue, we improved our `bismon` to avoid needing or depending on that `SIGTERM` feature in `libonion` (by using `signalfd` Linux specific facilities in `bismon` itself and passing the `O_NO_SIGTERM` flag to `onion_new`...).

So the effort on improving `SIGTERM` handling in `libonion` was concluded.

5.1.2 Contribution to *GCC*

There is no contribution yet to *GCC*, because it is not yet needed in october 2018. We reserve some effort for future such contributions, when our *GCC* plugin generator would require them. In the lucky case where no adaptation of *GCC* plugin infrastructure is necessary, the effort could be moved to other work in T1.3 (notably ST1.3.3).

5.2 Design and implementation of the compiler and linker extension

This is related to subtask ST1.3.4 (and also ST1.3.1) of CHARIOT GA

The compiler extensions will be *generated* GCC plugins.

The linker extension will compute some "cryptographic quality" hash code of the C or C++ translation units of the IoT software. Then it will interact with the blockchain, according to the *§6 API for Private key related transactions* of the *D1.2 Method for coupling preprogrammed private keys on IoT devices with a Blockchain system*. That API is a Web API and a C or C++ compatible plain API or library should be developed, following the tutorial code example of D1.2.

This chapter will be updated and completed in the upcoming and final version (in D1.3 ^{v2}).

6 Conclusion

The bismon free software is developed in an agile and incremental manner [200] (required by its bootstrapping approach), with continuous updates to **https://github.com/bstarynk/bismon/**.

In october 2018, the persistence machinery is working and daily used to enhance bismon. The agenda mechanism is working. A naive stop-the-world mark-and-sweep precise garbage collector is implemented. The generation of internal C code is done (by hand-written routines, still coded in C), this enables the meta-programming approach. The web interface is worked upon: a libonion based infrastructure is already handling HTTP requests, and a GDPR-compliant login form is presented on web browsers. Our jsweb_module contains the functions related to Javascript (nearly complete) and HTML generation (work in progress). The syntactical editor (replacing the crude GTK interface) and then the GCC plugin generation should be worked on.

In august 2019, the web machinery is mostly working. More generated C code is available. The JSON handling is incomplete. Bismon continuations are almost[201] reifiable into transient objects, having as payload a linked-list sequence of call frames.

The final D1.3 ^{v2} version (scheduled for M30) of this deliverable will explain the Web interfaces (both for the ordinary user, i.e. the IoT developer; and for the static analysis expert) and the generation of C++ code for GCC plugins, with some examples of simple, IoT focused, whole-program static source code analysis performed by *bismon*. So the final D1.3 ^{v2} document will contain a longer conclusion.

Within the timeframe allocated for CHARIOT it was not realistically possible in May 2020 to partly or fully generate GCC plugins C++ code (like past GCC MELT did), in particular because BISMON has no usable documentation, and understanding its persistent heap of 3485 objects is not realistic without such a documentation. A garbage collection Jones et al. [2016] design bug (and its subtle interaction with the powerful but complex GTK graphical toolkit) makes the current BISMON (of git id cb1c4ccfe3802fa33....) extremely brittle, to the point of being barely usable. The original insight was to generate most parts of such *Bismon* documentation, per the Unix philosophy and decade of related practice (from the original troff to prior practice in GCC MELT, or to DOXYGEN or OCAML ...) but such a documentation, even if it is quite reasonably easy to generate from an orthogonally persistent semantic network such as BISMON's heap, would largely overflow the 70 pages hard limit (CHARIOT consortium defined) of this report: notice that the generated GCC MELT past documentation had hundreds of A4 pages....

Machine learning techniques inspired by Zhang and Huang [2019] could be relevant in BISMON. See also the REFPERSYS research[202] project, inspired by Pitrat [1996, 2009a,b]; Starynkevitch [1990]; Bordini et al. [2020]. The deeplearning4j.org infrastructure (used as a web service), or opensource C++ machine learning libraries such as mlpack.org or TENSORFLOW machine learning, or topological data analysis libraries such as GUDHI (see The GUDHI Project [2020]; Maria [2020]; Maria et al. [2020]) or TTK (see Bin Masood et al. [2019]) be coupled to BISMON. Such future work would however require further funding for at least a year of qualified developer work (see also Maglogiannis [2007] and the ai4eu.eu project, but don't forget the empirical Hofstadter's law).

[200] So there are no released stable versions of this software, but snapshots.

[201] Thanks to generated invocations of the LOCALFRAME_BM *C* macro, which provides 90% of the development work: full transient reification of partial continuations, that is of call stack segments, is just a matter of clustering emitted stack-local struct stackframe_stBM-based linked-lists of Bismon call frames.

[202] Also related: talks in the memory of J.Pitrat, AFIA, March 6th, 2020, Paris.

Appendix A Building `bismon` from its source code

We focus here on how to build `bismon` from its source code on Debian-like distributions running on x86-64 computers, such as Debian Buster 10.6, or Ubuntu 20.04, or Linux Mint 20. Familiarity with the command line is required[203], with `root` access (e.g. using `sudo`). Fluency with `git` is expected, and it is strongly advised to `git commit` every few hours (including your persistent store, when `bismon` is *not running*).

The reader is expected to be authorized (by his/her management, if that build is done professionally) to build `bismon` from its source code and probably also some recent GCC cross-compiler on his/her Linux workstation and should budget several days of work for that.

Be aware that **`bismon` requires specifically some GCC 10 compiler** and won't work with e.g. a GCC 9 compiler.

The build procedure happens in two phases:

- a **configuration step**, to be run only once in a while, or when your Linux distribution has changed or upgraded, or when you have added extra useful libraries, or have upgraded your GCC compiler.
- a **compilation step**, to be run more frequently (e.g. every night using `crontab(1)`....)

A.1 Prerequisites for building `bismon`

The `bismon` source code is on **`github.com/bstarynk/bismon/`** and the reader is expected to be capable of getting that source code on his/her Linux workstation. A possible command to retrieve that code might be `git clone https://github.com/bstarynk/bismon.git` ; you'll then obtain a *fresh* `bismon/` subdirectory containing the source code. About 100Mbytes of disk space (for less than 2000 inodes) is required.

A recent `libonion` library[204] (version 0.8 at least) is required. Fetch `libonion`'s source code from **`github.com/davidmoreno/onion/`** and follow its build instructions: probably `mkdir _build` then `cd _build` then `cmake ..` then `make` and at last `sudo make install`. That `libonion` library needs less than 25Mbytes of disk space, `cmake` and several libraries (in particular support for `openssl`, `gcrypt`, `systemd`, `sqlite3`, `lzma`, `libicu`, `libpam`) to be built. Check and inspect your `onion/version.h` header file[205], it should have some `ONION_VERSION` close to `0.8.150` at least.

The GNU readline (GPLv3+ licensed) library is required, at version 8. It is useful for autocompletion abilities in interactive situations.

Ian Lance Taylor's libbacktrace library is needed for backtraces on error and in warnings, and possibly for future (generated) introspective code. This library takes advantage of DWARF debugging metadata in object files and executable, so it is advised to compile every *Bismon* source file (either handwritten or generated) with `-g` (and possibly also `-O2` for optimization) flag to `gcc` or `g++`.

The *Bismon* project uses internally GNU `make`, version 4.2 at least. Our hand-written `GNUmakefile` is driving it.

A.2 File naming conventions in `bismon`

By our conventions, files whose base name[206] start with a single underscore (that is, a _ character) are generated: for example `_bismon-config.mk` and `_bm_config.h`, etc... However, some of them need to be kept, backed-up and version controlled but would be regenerated by running `make redump`.

File names whose base name start with two underscores, such as `__timestamp.c`, are temporary and can be removed. They would be removed by running `make clean` or `make distclean`. Of course, object files (suffixed `.o`) and shared libraries (suffixed `.so`, see Drepper [2011]) are also temporary, and could be removed then regenerated. Some of these (in particular under `modubin/` directory) are `dlopen(3)`-ed.

[203] For example, the reader is expected of being able to build GNU make or GNU bash from their source code.

[204] This is an open source library for web HTTP and HTTPS service. It is LGPL licensed.

[205] You might use `locate(1)` or `find(1)` to find files on your Linux box. On *my* Linux machine, that header file is in `/usr/local/include/onion/version.h` and comes from *libonion* git commit `43128b031995`....

[206] In the sense of the `basename(1)` command applied to the file path.

The main executable is named `bismon`. But `BM_makeconst` and `BISMON-config` are auxiliary metaprograms (generating C or C++ code). All three are ELF executables.

A.3 Naming conventions and source files organization for `bismon`

naming and coding conventions in hand-written *C* code

- **All** public ELF **names of hand-written functions or global variables** (as known to `nm(1)`, `objdump(1)` or to `dlsym(3)` **are conventionally suffixed by `_BM`**. For example, we have some `prime_above_BM` function giving some prime number above a given reasonable positive integer.
- **We have conventional suffixes:** Our public `struct`-s are generally tagged with a name ending with **`_stBM`**; Our `typedef`-ed types are suffixed with **`_tyBM`**; usually their field names is globally unique and share a common prefix (e.g. in `struct parstoken_stBM` field names all start with `tok`). Public signatures (useful for C function pointers) are suffixed with **`_sigBM`** (for example, the initialization of generated modules is a C function of signature `moduleinit_sigBM`). Most public `enum`-s have their name ending with **`_enBM`** e.g. `space_enBM` for space numbers.
- Preprocessor symbols or macros are in all capital ending with **`_BM`**, notably the important `LOCALFRAME_BM` variadic macro for local roots known to our garbage collector.

Hand-written *C* code files

- The header file `bismon.h` is our only public header file, and is `#include`d everywhere. It includes system headers (e.g. `<unistd.h>` or `<pthread.h>`, and the following "internal" headers:
  1. `cmacros_BM.h` is `#define`-ing important global preprocessor macros, including `FATAL_BM` for fatal errors, `LOCALFRAME_BM` variadic macro for local roots, `DBGPRINTF_BM` for debugging output, `WARNPRINTF_BM` for warning messages, `INFOPRINTF_BM` for informational messages, etc... The `ROUTINEOBJNAME_BM` macro is giving the routine name of a given *objid*.
  2. `id_BM.h` is implementing our object ids.
  3. `types_BM.h` is defining our global types, `struct`-s, etc... Notice the `value_tyBM` opaque type (a `void*` pointer) for Bismon values.
  4. `global_BM.h` is declaring our `extern`al global data, some of which is generated.
  5. `fundecl_BM.h` is declaring our global hand-written functions. Some of them are `static inline` for efficiency reasons (for example `elapsedtime_BM` returning the elapsed clock time as a `double` number in seconds, or `valhash_BM` to compute the hash code of a Bismon value.
  6. `inline_BM.h` is implementing our global `static inline` functions.
- `agenda_BM.c` is implementing our agenda with tasklets (see §1.7 above).
- `allocgc_BM.c` is implementing low-level memory allocation and garbage collector (see §2.2 above).
- `assoc_BM.c` is implementing associative lists and tables, in particular for object attributes.
- The `code_BM.c` file contains many Bismon routines for closures.
- The `dump_BM.c` file is implementing the dump of the persistent store. See § 2.3.
- The `emitcode_BM.c` file contains many Bismon routines for emission or of C code in modules.
- The `engine_BM.c` file is related to tasklets in the agenda (see §1.7 above).
- `gencode_BM.c` is related to C code generation.

- `id_BM.c` implements objid support.
- `jsonjansson_BM.c` is for JSON support. values.
- `list_BM.c` is for list values.
- `load_BM.c` is for loading the persistent store .
- `main_BM.c` has the `main` function and support functions (fatal errors, etc...).
- `node_BM.c` implements node values.
- `object_BM.c` implements objects.
- `parser_BM.c` implements the parser with callbacks
- `primes_BM.c` contains an array of prime numbers, and related utilities. They could be useful in hash tables and in some hash functions. In several cases, flexible array members inside BISMON are allocated with a prime number size.
- `scalar_BM.c` implements scalar values numbers. (strings, boxed doubles).
- `sequence_BM.c` implements sets and tuples.
- `user_BM.c` relates to reifications of contributors and users.
- `misc_BM.cc` is a **C++** file, to take advantage of some standard C++ containers.

The persistent store

The persistent data (see §2 above) sits in files `store*BISMON.bmon` (using `glob(7)` notation); more precisely

- `store1-BISMON.bmon` is for predefined objects. The header file `genbm_predef.h` is generated from them at dump time.
- file `store2-BISMON.bmon` contains the global object space. Several global objects describe modules whose C code is generated (e.g. at dump time) under sub-directory **`modules/`**.
- other `store`***`i`***`-BISMON.bmon` textual files contain[207] objects in space ranked i .... Notice that all these files are both loaded and dumped, and should be backed-up (like the **`modules/`** directory) regularily.

These *generated* textual `store`***`i`***`-BISMON.bmon` *files* should be *version controlled* by the `git` tool. You might use the `make redump` command to regenerate the persistent store and the modules, and it is advised[208] to run it daily.

Users and contributors related files

The BISMON system does need some minimal data about users. The reader of this report is expected to verify (perhaps with the help of lawyers) that such data is compliant and compatible with regulations like the European GDPR.

- the textual file `contributors_BM` describes the known contributors to the BISMON software. That file has comments (or ignored lines) starting with the `#` character. Non-comment lines contain three or four fields, separated by semi-colons (i.e. `;`):
  1. the user or contributor name, as known to the system. It could be some pseudo.
  2. the unique objid of the BISMON object describing that user or contributor
  3. the email of that user.

[207] So there cannot be any `store0-BISMON.bmon` file, since space 0 is for transient objects which are never dumped.

[208] Once `make redump` fails, the persistent store is inconsistent and corrupted. This should not happen, but when it does, use `git`

4. an optional email alias of the same user

- the textual file `passwords_BM` associates an objid with some encrypted password. known This is used for the login form of the web interface, and should not be readable by group or others. Lines inside this files are either comments (or ignored lines) starting with the # character, or entry lines with a user name, then a semicolon, then the objid, then a semi-colon, then the encrypted password.

Since the main author of this draft report is known to BISMON and reified in the object of objid `_6UYrSn7piPM_3eYhL` the file `contributors_BM` should at least contain a line like perhaps `Basile Starynkevitch;_6UYrSn7piPM_3eYhLtoXlmL;bs`

It is preferable to run the `./bismon` software with specific command lines argument to update the contributors file and the passwords file.

A.4 Generators and meta-programs in bismon

Generating code is one of the core ideas of BISMON. Such code generation happens both at build time and at run time. The generated code is usually some C file[209].

At build time, two meta-programs and some shell or GNU awk scripts are involved; each of these two metaprograms has a single handwritten C++ source code file:

- `BISMON-config` is querying some parameters from the user (that is the Linux sysadmin installing `bismon`) and generates some C++ files.
- `BM_makeconst` is usually scanning some handwritten C file (for example, our `engine_BM.c` file, etc...) and producing some headers or utility files.
- `timestamp-emit.sh` is a shell script (using internally the `emit-git-sources.gawk` GNU gawk script) which emits a simple C file (containing only data) with the timestamp and some metadata information about the build.

But once the `bismon` ELF executable exists, the above metaprograms are not useful anymore. However, they are needed to recompile `bismon` (which you might want to do periodically, i.e. every evening).

At run time, the `bismon` executable is routinely generating C or C++ code. Some C code (under the **`modules/`** directory) is generated to extend the behavior of `bismon` itself : the generated C code is compiled, and the resulting shared object is `dlopen(3)`-ed but never[210] `dlclose(3)`-ed.

Conventionally, we want the generated persistent files to contain the `$GENERATED_PERSISTENT$` string, and if possible to have a `$` inside the file path. But generated temporary (or transient) files should contain the `¤GENERATED¤` string, and if possible have a starting underscore (that is, a _ character) in their file name.

@@TO BE COMPLETED

Appendix B Configuring bismon from its source code

Warning: This configuration step has to be done again as soon as your GCC compiler or cross-compiler has changed versions, or when you have added new important libraries on the LINUX workstation running *Bismon*.

First, **inspect, and *improve if needed*, the `Configure` shell script** for `/bin/bash`. **Then run that script** using the `./Configure` command.

Appendix C Building bismon from its source code

Warning: This building step should probably be run every evening, e.g. using a `crontab(1)` job. It should be done only after the `./Configure` script has been successfully run.

First, **inspect, and *improve if needed*, the `Build` shell script** for `/bin/bash`. **Then run that script**

`bisect` to find the latest consistent state of your *Bismon* repository.

[209] With additional funding and more time, we could have used `libgccjit` to generate directly some `*.so` shared object.

[210] Not `dlclose`-ing is of course some kind of memory leak, since the virtual address space of the process running `bismon` is

using the `./Build` command.

C.1 Checking the version of **bismon**

Once the `./Build` script did work correctly, there should be some `bismon` executable file. Check first using the `file ./bismon` command, it should give you something similar to:

```
./bismon: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linke
 interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=b4e74f225dff6115a01764abfe361a1b7757a208, for GNU/Linux 3.2.0,
with debug_info, not stripped
```

Then use `ldd ./bismon` to verify that all dependencies are present. On some computer, I am possibly getting the following output:

```
 % ldd ./bismon
linux-vdso.so.1 (0x00007ffee532a000)
libonion.so.0 => /usr/local/lib/libonion.so.0 (0x00007f5ee9ec2000)
libglib-2.0.so.0 => /lib/x86_64-linux-gnu/libglib-2.0.so.0 (0x00007f5ee9d99000)
libjansson.so.4 => /usr/local/lib/libjansson.so.4 (0x00007f5ee9d8a000)
libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1 (0x00007f5ee9d4f000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f5ee9d2c000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f5ee9d24000)
libstdc++.so.6 => /lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f5ee9b43000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f5ee99f4000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f5ee99d9000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5ee97e7000)
libxml2.so.2 => /lib/x86_64-linux-gnu/libxml2.so.2 (0x00007f5ee962d000)
libpam.so.0 => /lib/x86_64-linux-gnu/libpam.so.0 (0x00007f5ee961b000)
libgcrypt.so.20 => /lib/x86_64-linux-gnu/libgcrypt.so.20 (0x00007f5ee94fb000)
libgnutls.so.30 => /lib/x86_64-linux-gnu/libgnutls.so.30 (0x00007f5ee9325000)
libsqlite3.so.0 => /lib/x86_64-linux-gnu/libsqlite3.so.0 (0x00007f5ee91fc000)
libhiredis.so.0.14 => /lib/x86_64-linux-gnu/libhiredis.so.0.14 (0x00007f5ee91e9000)
libsystemd.so.0 => /lib/x86_64-linux-gnu/libsystemd.so.0 (0x00007f5ee913a000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007f5ee90c7000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5eea0b9000)
libicuuc.so.66 => /lib/x86_64-linux-gnu/libicuuc.so.66 (0x00007f5ee8edf000)
libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f5ee8ec3000)
liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x00007f5ee8e9a000)
libaudit.so.1 => /lib/x86_64-linux-gnu/libaudit.so.1 (0x00007f5ee8e6e000)
libgpg-error.so.0 => /lib/x86_64-linux-gnu/libgpg-error.so.0 (0x00007f5ee8e4b000)
libp11-kit.so.0 => /lib/x86_64-linux-gnu/libp11-kit.so.0 (0x00007f5ee8d13000)
libidn2.so.0 => /lib/x86_64-linux-gnu/libidn2.so.0 (0x00007f5ee8cf2000)
libunistring.so.2 => /lib/x86_64-linux-gnu/libunistring.so.2 (0x00007f5ee8b70000)
libtasn1.so.6 => /lib/x86_64-linux-gnu/libtasn1.so.6 (0x00007f5ee8b5a000)
libnettle.so.7 => /lib/x86_64-linux-gnu/libnettle.so.7 (0x00007f5ee8b20000)
libhogweed.so.5 => /lib/x86_64-linux-gnu/libhogweed.so.5 (0x00007f5ee8ae8000)
libgmp.so.10 => /lib/x86_64-linux-gnu/libgmp.so.10 (0x00007f5ee8a62000)
librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x00007f5ee8a57000)
liblz4.so.1 => /lib/x86_64-linux-gnu/liblz4.so.1 (0x00007f5ee8a36000)
libicudata.so.66 => /lib/x86_64-linux-gnu/libicudata.so.66 (0x00007f5ee6f75000)
libcap-ng.so.0 => /lib/x86_64-linux-gnu/libcap-ng.so.0 (0x00007f5ee6f6d000)
libffi.so.7 => /lib/x86_64-linux-gnu/libffi.so.7 (0x00007f5ee6f5f000)
```

At last, you can ask `bismon` to give its version information using `./bismon -version` which might output something like:

```
% ./bismon --version
./bismon: version information
 timestamp: jeu. 28 janv. 2021 13:32:53
 git id: d62d320623ad2a7e73019e736870039508c36d02
 last git commit: d62d320623ad tell about the Build script in end of BISMON-config.cc
 last git tag: heads/master
 source checksum: d929d9edebcd6a76618901c7ebbaed71
 source dir: /home/basile/bismon-master
 GNUmakefile: /home/basile/bismon-master/GNUmakefile
########
run
   ./bismon --help
to get help.
```

Please report the output of `./bismon -version` before any question on BISMON.

The output of `./bismon -help` is giving up to date information about invoking `./bismon` (e.g. in some `cron` script). For example:

```
Usage:
  bismon [OPTION?] * the BISMON static source code analyzer *

BISMON is a static source code analyzer, using GCC.
see github.com/bstarynk/bismon commit 7b3f952207eb46a6...
WITHOUT WARRANTY, since GPLv3+ licensed

Help Options:
  -h, --help                                      Show help options

Application Options:
  -l, --load=DIR                                  load persistent heap from directory DIR (default is .)
  -d, --dump=DIR                                  dump persistent heap into directory DIR
  --dump-after-load=DIR                           dump after load the persistent heap into directory DIR
  --contributors-file=PATH                        use PATH as the contributors file;
 .. default is contributors_BM or $HOME/contributors_BM
  --passwords-file=PATH                           use PATH as the password file;
 .. default is passwords_BM or $HOME/passwords_BM
  --contact-file=PATH                             use PATH as the master contact file;
 .. default is contact_BM or $HOME/contact_BM
  --add-passwords=PASSWORDENTRIES                 use the given PASSWORDENTRIES file (if it is -, stdin) containi
  --emit-has-predef=NB                            emit NB 'HAS_PREDEF_BM'
  -j, --job=NBJOBS                                number of worker threads NBJOBS (>=2, <16)
  --random-seed=SEED                              set the initial PRNG seed for g_random_int to given SEED
  --pid-file=PATH                                 use PATH as the pid file;
 .. default is _bismon.pid
  --run-command=CMD                               run the command CMD
  -i, --init-after-load=<closure> (or object)     do the <closure> after loading, as initialization
  -c, --chdir-after-load=DIR                      change directory after load to DIR (perhaps making it)
  --parse-value=EXPR                              parse (after loading) the value EXPR
  --test-plugin=PLUGINAME                         run the drafts/testplugin_PLUGINAME.so (after loading) the test
plugin PLUGINAME
  --comment-predef=COMM                           set comment of next predefined to COMM
  --add-predef=PREDEFNAME                         add new predefined named PREDEFNAME
  --contributor=CONTRIBUTOR                       add or change contributor CONTRIBUTOR,
 like 'First Lastname <email@example.com>'
 or 'First Lastname;email@example.com;aliasmail@example.org'
 (this puts personal information relevant to European GDPR in file contributors_BM)
  --print-contributor-of-oid=CONTRIBOID           print tab-separated: (full name, objid, email, alias) of contri
  --remove-contributor=CONTRIBUTOR                remove existing contributor CONTRIBUTOR,
 like 'First Lastname'
 or email@example.com
 or some existing contributor oid similar to _2PFRochKb3N_3e8RFFAUi9K
 (this should remove personal information relevant to European GDPR in file contributors_BM)
  --cleanup                                       cleanup memory at end (for valgrind)
 ... (see valgrind.org for more).
  --final-gc                                      forcibly run a final garbage collection (after any dump or ever
  --batch                                         run in batch mode without user interface
```

never shrinking.. This explains why the `bismon` process should be restarted at least daily. Our `manydl.c` program demonstrates that `dlopen(3)`-ing many thousands of times is practically possible on modern LINUX workstations.

```
  -D, --debug                                    gives lots of debug messages
  --debug-after-load                             enable debug messages after load
  --emit-module=MODULEOBJ                        emit module MODULEOBJ
  --mailhtml-file=FILE                            FILE is the file, in HTML, to be sent to contributor
  --mailhtml-subject=SUBJECT                      SUBJECT is the subject of the email to be sent to contributor
  --mailhtml-contributor=CONTRIBUTOR              CONTRIBUTOR is to whom the email will be sent
  --mailhtml-attachment=ATTACHEDFILE              ATTACHEDFILE is the attached file of the email
  --version                                      gives version information
  --ssl-certificate=FILEPREFIX                   Uses FILEPREFIX.pem & FILEPREFIX.key for SSL certificate to lib
  --web-base=WEB_BASE                            A string like <host>:<port>, default is localhost:8086, describ
  --anon-web-session=COOKIEFILE                  Create an anonymous web session, and write its cookie in the gi
```

Appendix D Dumping and restoring the `bismon` persistent heap

It is absolutely essential that the `make redump` command works well., and you need to run that command regularily. See also §2.3

It is practically important to use the periodically `git` version control system on the repository, and this includes the persistent heap dump files `store*BISMON.bmon` - when `bismon` is **not running**. It is recommended to `git commit` these files twice a day at least, when the `bismon` executable is not running.

Index

! (, 35
!), 35
«, 35
», 35
Σ, 35
objid abbreviation, 31

A
abstract syntax tree, 22
abstraction, 7
address, 38
agenda, 27, 46, 52
agenda_BM.c file, 52
AI4EU project, 50
algorithm, 33
alias
    pointer, 38
alloca stack-allocation primitive, 40
allocgc_BM.c file, 52
analysis, 4
    static, 8
and_then, 46
A-normal form, 12
application, 22
    web, 22
ARDUINO platform, 38, 39
ARM architecture, 8
artificial intelligence, 22
as, 4
as-if rule, 15
asm keyword, 40
ASN-1, 8
assoc_BM.c file, 52
association, 52
associative table, 32
AST, 22, 29, 48
atomic, 31
atomic variable, 22
attribute, 31, 36, 41, 52
auto keyword, 40
automotive crash, 40
avalanche, 40
AVR microcontroller, 38

B
backup
    remote, 40
BARELFISH operating system, 29
basic block, 13
basiclo_assign, 23
basiclo_cexpansion, 23
basiclo_cond, 23
basiclo_exit, 23
basiclo_extractjsonstmt, 43
basiclo_fail, 23
basiclo_function, 23
basiclo_lockobj, 23
basiclo_loop, 23
basiclo_minifunction, 23
basiclo_return, 23
basiclo_run, 23
basiclo_when, 23
basiclo_while, 23
big data, 38
binutils, 4
Bismon, 6, 11, 20
bismon, 4
_bismon-config.mk, 51, 54
bismon.h header, 52
_bm_config.h, 51
BM_makeconst metaprogram, 54
bootstrap, 21, 24, 34
browser, 18, 21, 27, 45
buffer, 47
buffer overflow, 39
Build script, 55
    automation, 51
build automation, 17
building
    bismon, 55

C
C++, 53
C++11, 6, 16
C11, 6
CAIA, 34
call frame, 50
call stack, 39, 50
callback
    parser, 53
canonical JSON encoding, 41
canonjsonifyvalue
    canonjsonifyvalue_BM, 41
cast
    pointer, 40
CENTAUR system, 22
checkpointing
    application, 40
CHICKEN, 34
CHICKEN SCHEME, 34
city
    smart, 40

CLANG, 4
CLASP, 34
class, 31, 32, 35
clean, 51
clock
    real-time, 39
closure, 30, 41, 46, 52
cmacros_BM.h header, 52
code, 39
    generation, 52
    refactoring, 40
    scientific, 40
code analysis, 4
code chunk, 23
code refactoring, 40
code review, 7, 38
code smell, 38
code_BM.c file, 52
collaborative sofware, 22
collision, 30
comment
    comment, 24
compiler, 7, 11
compiler sanitizer, 9
component, 31, 41
computing grids, 40
concurrency, 40
condition variable, 22
configuration, 54
configuration file, 39
Configure script, 54
container
    C++, 16, 53
contination
    reification, 50
contributor, 25, 45, 53
contributors_BM file, 53
control
    keyboard sequence, 48
control flow graph, 37
cookie, 45
copy/paste, 48
copying
    garbage collection, 33
copy-paste, 25
corruption
    file system, 39
COVID 19 illness, 40
critical software, 10
crontab(5), 40
cross-compiler, 11
cryptographic protocol, 39
current_closure, 24
current_closure_size, 24
current_module, 24
current_routine, 24

D

data, 29
    global, 52
data analysis
    topological, 38
data mining, 22, 38
database, 33, 40
DBGPRINTF_BM macro, 52
decision
    algorithm, 33
declarative programming, 22
decltype keyword, 40
DECODER, 29, 38
decoding
    JSON nodal, 42
DEEPLEARNING4J framework, 50
defect
    software, 8
denial-of-service, 45
depth-first, 33
development
    software, 6
dictionary, 32
distclean, 51
distributed computing, 28
DOM, 48
Domain-Specific Language
    DSL, 20
double, 29, 41
    boxed, 53
doxygen documentation generator, 7
DSL, 4, 7, 20, 22, 48
    domain specific language, 7
dump, 20, 33, 46, 52, 53
dump_data, 36
dump_scan, 36
dump_value, 36
dump_BM.c file, 52
dumper, 36
dumper_object, 36
dynamic request, 45

E

editor
    syntactical, 22, 48
elapsedtime_BM.h function, 52
ELF, 52
email, 39
email classification, 22
emission, 52

emit_xhtml, 46
emit_xhtml_open, 47
emitcode_BM.c file, 52
emit-git-sources.gawk internal GAWK script, 54
endianness, 8
engine_BM.c file, 52
error condition, 39
/etc/ configuration directory, 39
ETHERNET protocol, 8
ethics, 22
expert
 IoT, 6
 software engineering, 6, 9
 static analysis, 22, 48
exploration
 depth-first, 33
extract_json, 43
extraction
 from JSON, 43
extractor
 from JSON, 43

F

face recognition, 22
failure
 of standard C functions, 39
FATAL_BM macro, 52
fault injection, 39
file, 34, 51
 generated, 54
file system, 39
flexible array member, 53
floating point, 29
 precision, 39
FLTK, 6
forecast
 avalanche, 40
forge, 5
formal methods, 9
FRAMA-C, 4
framework, 22
fread function, 39
FREEBSD operating system, 38
FREERTOS kernel, 38, 39
fscanf function, 39
function, 52
 inline, 52
 key, 48
fundecl_BM.h header, 52

G

garbage collection, 20, 26, 33
garbage collector, 32, 52
GCC, 4
GCC MELT, 4, 50
GDPR, 21, 26, 53
genbm-predef file, 53
gencode_BM.c file, 52
¤GENERATED¤, 54
§GENERATED_PERSISTENT§, 54
generation, 52
 of code, 52
 of files, 54
Generic, 4
Gimple, 9
Gimple, 4, 12
git, 22, 24, 53
GLIBC library, 39
global_BM.h header, 52
GPL, 4
grid
 smart, 40
GRID4EU project, 40
GTK, 24
GUDHI topological data analysis framework, 50

H

H2020, 2, 7, 10, 29
hash
 of values, 52
hash map, 32
hash set, 32
hash table, 53
hexa, 47
homoiconicity, 48
html_active_object, 47
html_element_object, 47
html_sequence_object, 47
html_spacing, 47
html_void_element_object, 47
HTML5, 46, 48
HTTP (web protocol), 39
HTTPS protocol, 8

I

I/O error, 39
IA-32 architecture, 8
id, 47
id_BM.h header, 52, 53
IDE
 integrated development environment, 6
idle queue, 27
IEEE 802.11 protocol, 8
IMAP (email retrieval protocol), 39
immutable, 29
implementation
 of programming languages, 7

INFOPRINTF_BM macro, 52
inline_BM.h header, 52
int, 47
integer, 29, 41
integrated development environment, 6
IoT, 4, 6, 8, 20
IP protocol, 8
IPSE, 10

J
JPEG image encoding, 8
JSON, 32, 41, 53
    extractor, 43
json_object, 41
jsonjansson_BM.c file, 53

K
kernel
    operating system, 38
keyboard, 48

L
language
    domain specific (DSL), 7
    dynamic, 22
    programming, 7
LaTeX formatter, 7
ld, 4
libonion, 4, 6
library
    shared, 51
LI-FI protocol, 8
linked list, 32
LINUX operating system, 6, 38, 39
list
    associative, 52
list_BM.c file, 53
LLVM, 4
load, 33, 53
load_BM.c file, 53
LOCALFRAME_BM, 50
LOCALFRAME_BM macro, 52
login, 21, 45, 46
login template, 21
lout document formatter, 7

M
machine learning, 9, 40, 50
main_BM.c file, 53
make, 51
malloc, 39
manipulation
    of AST, 48
markup
    element, 48
medical device, 40
MELT, 4
member
    flexible array, 53
memory, 39
    code, 39
    shared, 22
    virtual, 22
MENTOR system, 22, 48
meta-data, 39
meteorological
    computing grids, 40
methods
    formal, 9
MIPS architecture, 38
misc_BM.cc file, 53
MISRA C, 38
MLPACK machine learning framework, 50
modubin/, 51
module, 26, 33, 52, 53
    webxhtml_module, 46
modules/ directory, 53
mtime, 31, 35
multi-threading, 22
multi-tier programming, 28
MUSL LIBC library, 39
mutable, 30
mutator thread, 21
mutex, 22, 29

N
name, 32, 35, 47
naming
    conventions, 51
nb_conds, 23
newline, 47, 48
nil, 30, 41
nlsp, 47, 48
nodal
    JSON decoding, 42
node, 30, 41, 42, 48
    transient, 30
node_BM.c file, 53
non-critical system, 10
NTP - network time protocol, 39
NULL in C, 38, 42
    JSON, 42
null_object, 24
null_value, 24
nullptr in C++, 38
nuweb system, 7

O
object, 29, 30, 41, 47, 48, 52, 53

file, 51
object_BM.c file, 53
objid, 30, 36, 41, 42, 45, 52, 53
OCAML, 34
ocamldoc documentation generator, 7
octa, 47
oil industry, 40
OPENWRT router, 39
operating system, 38
optimization, 13
orthogonal persistence, 33
out of memory, 39

P
parser_BM.c file, 53
partial evaluation, 34
pass
optimization, 13
passwords_BM file, 54
payload, 32, 36, 41
persistence, 4, 20, 33, 52, 53
orthogonal, 33
persistent, 27
monitor, 11
store, 53
persistent monitor, 11
persistent state, 33
persistent store, 22
personal data, 21
personel, 22
photography
digital, 9
plugin, 21
pointer, 38
power
grid, 40
precision, 39
predefined, 31, 33
prime, 53
primes_BM.c file, 53
priority, 27
processor
multi-core, 22
program
analysis, 4
programming
declarative, 22
programming languages, 7
projection, 34
PTHREAD library, 39

Q
QT, 6

R
race condition, 10, 22
RASPBERRYPI system, 8, 39
read-only memory, 39
redump, 51, 57
refactoring
code, 40
REFPERSYS project, 33, 50
register keyword, 40
register allocation, 40
regulation
compliance to, 21
reification
of continuations, 50
remote
database, 40
remote backup, 40
repository, 5
reproducing
faults, 39
request, 45
dynamic, 45
static, 45
RFB - remote framebuffer protocol, 8
Rice's theorem, 40
Rice's theorem, 38
RISC-V architecture, 38
routine, 30, 31, 52
routine address, 31
routine signature, 31
ROUTINEOBJNAME_BM macro, 52
rsync(1), 40

S
sanitizer
compiler, 9
SATA protocol, 8
SBCL, 34
scalar_BM.c file, 53
Scheme, 34
scientific code, 40
SDK
software development kit, 6
search
depth-first, 33
SELF, 34
semantics, 34
sequence, 30, 48
sequence_BM.c file, 53
session
web session, 45
set, 29, 41
shared memory, 22
signature

function, 35
silver bullet, 8
simulation software, 40
size, 30
smart cities, 40
smell
code, 38
SMTP (mail sending protocol), 39
social credit system, 22
software development, 6
software development kit, 6
SOFTWAREHERITAGE project, 40
son, 30
source code, 6, 22
Bismon, 6
`space`, 31, 47, 48
specificaton, 7
SQLITE database library, 39
SSA, 4
SSH secure shell, 8
stack overflow, 37, 39
`stackframe_stBM`, 50
STANCE project, 39
state, 33
static analysis, 8, 22
tool, 9
static code analysis, 4
static request, 45
store, 52
persistent, 53
`store*BISMON.bmon` files, 53
`storei-BISMON.bmon` files, 53
string, 29, 41, 53
string buffer, 32
suffix
file, 52
super-computers, 40
surveillance, 22
SVG, 48
synchronization, 22
syntax-oriented
editor, 22, 48

T
tasklet, 27, 33, 46, 52
`temporary_attribute_object`, 35
`temporary_webhandler_dict_object`, 46
test, 39
thread, 39
mutator, 21
thread pool, 26
threading
multi-, 22
time, 39
elapsed, 52
`time(7)`, 39
timeout
web, 46
`timestamp-emit.sh` shell script, 54
timing, 39
TLS cryptographic protocol, 39
topological data analysis, 38, 50
touch, 31
tracing
garbage collection, 33
transcompiler, 22
transient, 20, 27, 31, 36, 46, 48
node, 30
object for continuations, 50
`transient_connective_object`, 30, 33
transpiler, 22
tree
JSON, 42
TRL, 7, 38, 45
TTK topological data analysis toolkit, 50
tuple, 29, 41
twins
digital, 40
type inference, 40
`types_BM.h` header, 52

U
undefined behavior, 6, 10
`union`, 40
unspecified, 30, 41
USB protocol, 8
USB storage, 39
`user_BM.c` file, 53
UUID, 30

V
`valhash_BM` function, 52
value, 29
double, 53
list, 53
node, 53
object, 53
string, 53
`value_tyBM` opaque types, 52
vector, 32
version control, 24
VESSEDIA, 10
virtual memory, 22
`volatile` keyword, 40
vulnerability, 7
security, 8

W
`WARNPRINTF_BM` macro, 52

WAYLAND display server, 8
weather prediction, 40
web application, 22
web cookie, 45
web handler, 46
web interface, 18, 21, 45
web request, 45
web service, 39
web session, 45
web socket, 46
web timeout, 46
webdict_root, 46
webhandler_dict_object, 46
WebSocket, 48
wifi, 8
worker thread, 26
-Wstack-usage warning option, 39

X

x86 architecture, 8
X.509 protocol, 8
XHTML5, 46
XML, 46
X11 window system, 8

References

H. Abelson, G. J. Sussman, and with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd edition, 1996. ISBN 0-262-01153-0. URL https://mitpress.mit.edu/sites/default/files/sicp/index.html.

N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised 5th report on the algorithmic language Scheme. *SIGPLAN Not.*, 33(9):26–76, Sept. 1998. ISSN 0362-1340. URL https://schemers.org/Documents/Standards/R5RS/.

A. Ahmad, F. Bouquet, E. Fourneret, and B. Legear. Model-based testing for internet of things systems. In A. M. Memon, editor, *Advances in Computers*, volume 108, chapter 1, pages 1–58. Academic Press, 2018. ISBN 978-0-12-815119-8. URL http://doi.org/10.1016/bs.adcom.2017.11.002.

A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006. ISBN 0321486811.

S. Amershi, D. Weld, M. Vorvoreanu, A. Fourney, B. Nushi, P. Collisson, J. Suh, S. Iqbal, P. N. Bennett, K. Inkpen, et al. Guidelines for human-ai interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 3. ACM, 2019. URL http://saleemaamershi.com/papers/amershi.HAI.Guidelines.CHI.2019.pdf.

N. Amin and T. Rompf. LMS-Verify: Abstraction without regret for verified systems programming. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 859–873, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009867. URL http://doi.acm.org/10.1145/3009837.3009867.

E. S. Andreasen, A. Møller, and B. B. Nielsen. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, pages 31–36, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5072-3. doi: 10.1145/3088515.3088521. URL http://doi.acm.org/10.1145/3088515.3088521.

K. R. Apt and M. Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2006.

R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015. URL http://pages.cs.wisc.edu/~remzi/OSTEP/.

K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, University of California at Berkeley, 2006. URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

L. Atkinson. answer to: "how can a web server know all the tabs served by it..". *Software Engineering* forum, 2018. URL https://softwareengineering.stackexchange.com/a/378508/40065.

I. Attarzadeh and O. Siew Hock. Project management practices: Success versus failure. In *2008 International Symposium on Information Technology*, volume 1, pages 1–8, Aug 2008. doi: 10.1109/ITSIM.2008.4631634.

H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

D. Bakken. *Smart Grids: Clouds, Communications, Open Source, and Automation*. CRC Press, 2014. ISBN 978-1-4822-0611-1.

T. Barrelfish. Barrelfish architecture overview, barrelfish technical note 000. Technical report, Technical report, ETH Zurich, 2013.

P. Barry. Abstract syntax notation-one (asn. 1). In *IEE Tutorial Colloquium on Formal Methods and Notations Applicable to Telecommunications*, pages 2–1. IET, 1992.

P. Baudin, A. Pacalet, J. Raguideau, D. Schoen, and N. Williams. Caveat: a tool for software validation. *Proceedings International Conference on Dependable Systems and Networks*, pages 537–, 2002.

P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevesto. ACSL: ANSI/ISO C specification language (version 1.13). Technical report, CEA, INRIA, LRI, 2018. URL https://frama-c.com/download/acsl.pdf.

B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software: The* KEY *approach*. Springer-Verlag, 2007.

F. Belarbi. *Vehicle to infrastructure communication based systems: their contribution to road traffic management. Case of AIDA: the on-board traffic information system for motorways*. PhD thesis, 2004.

B. Berkes, M. Kellil, D. Pariente, G. Everhardt, V. László, and N. Zilio. VESSEDIA approach for security evaluation. Technical report, SLAB, 2018. URL https://vessedia.eu/downloads/VESSEDIA_D4.2_approach_security_evaluation_PU_M18.pdf. deliverable of Vessedia H2020 project, grant agreement No 731453.

K. Bhargavan, A. Delignat-Lavaud, C. Fournet, C. Hritcu, J. Protzenko, T. Ramananandro, A. Rastogi, N. Swamy, P. Wang, S. Z. Béguelin, and J. K. Zinzindohoué. Verified low-level programming embedded in F. *CoRR*, abs/1703.00053, 2017. URL http://arxiv.org/abs/1703.00053.

T. Bin Masood, J. Budin, M. Falk, G. Favelier, C. Garth, C. Gueunet, P. Guillou, L. Hofmann, P. Hristov, A. Kamakshidasan, C. Kappe, P. Klacansky, P. Laurin, J. Levine, J. Lukasczyk, D. Sakurai, M. Soler, P. Steneteg, J. Tierny, W. Usher, J. Vidal, and M. Wozniak. An Overview of the Topology ToolKit. In *TopoInVis*, 2019.

D. Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. doi: 10.1109/FOSE.2007.27. URL https://doi.org/10.1109/FOSE.2007.27.

P. Biswas, A. Di Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer. Venerable variadic vulnerabilities vanquished. In *Proceedings of the 26th* USENIX *Conference on Security Symposium*, SEC'17, pages 183–198, Berkeley, CA, USA, 2017. USENIX Association. ISBN 978-1-931971-40-9. URL http://dl.acm.org/citation.cfm?id=3241189.3241205.

D. G. Bobrow and T. Winograd. An overview of krl, a knowledge representation language. *Cognitive Science*, 1 (1):3–46, 1977. doi: 10.1207/s15516709cog0101_2. URL https://onlinelibrary.wiley.com/doi/abs/10.1207/s15516709cog0101_2.

R. H. Bordini, A. E. F. Seghrouchni, K. Hindriks, B. Logan, and A. Ricci. Agent programming in the cognitive era. *Autonomous Agents and Multi-Agent Systems*, 34, 2020. URL https://link.springer.com/article/10.1007/s10458-020-09453-y.

J. Braun, C. Koch, J. L. Davis, and G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(2), March 1956.

F. P. Brooks, Jr. No silver bullet - essence and accidents of software engineering. *Computer*, 20(4):10–19, Apr. 1987. ISSN 0018-9162. doi: 10.1109/MC.1987.1663532. URL https://doi.org/10.1109/MC.1987.1663532.

F. P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-83595-9.

P. Carle. MERING IV *: un langage d'acteurs pour l'intelligence artificielle distribuée integrant objets et agents par réflexivité compilatoire*. PhD thesis, 1992. URL http://www.theses.fr/1992PA066429. Thèse de doctorat dirigée par Ferber, Jacques Sciences appliquées Paris 6 1992.

X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik. Reconciling remote attestation and safety-critical operation on simple iot devices. In *Proceedings of the 55th Annual Design Automation*

Conference, DAC '18, pages 90:1–90:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3199853. URL http://doi.acm.org/10.1145/3195970.3199853.

V. G. Cerf and R. E. Icahn. A protocol for packet network intercommunication. *ACM SIGCOMM Computer Communication Review*, 35(2):71–82, 2005.

G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins, and P. W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981. ISSN 0096-0551. doi: https://doi.org/10.1016/0096-0551(81)90048-5. URL http://www.sciencedirect.com/science/article/pii/0096055181900485.

S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi. A survey on malware analysis and mitigation techniques. *Computer Science Review*, 32:1 – 23, 2019. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2019.01.002. URL http://www.sciencedirect.com/science/article/pii/S1574013718301114.

F. Chazal and B. Michel. An introduction to Topological Data Analysis: fundamental and practical aspects for data scientists. *arXiv e-prints*, art. arXiv:1710.04019, Oct 2017. URL https://ui.adsabs.harvard.edu/abs/2017arXiv171004019C.

C. Chen and S. Helal. A device-centric approach to a safer internet of things. In *Proceedings of the 2011 International Workshop on Networking and Object Memories for the Internet of Things*, NoME-IoT '11, pages 1–6, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0929-5. doi: 10.1145/2029932.2029934. URL http://doi.acm.org/10.1145/2029932.2029934.

C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov. 1970. ISSN 0001-0782. doi: 10.1145/362790.362798. URL http://doi.acm.org/10.1145/362790.362798.

B. Christian and T. Griffiths. *Algorithms to Live By: The Computer Science of Human Decisions*. Picador, 2017. ISBN 978-1627790369. URL http://algorithmstoliveby.com/.

R. Clarke. Big data, big risks. *Information Systems Journal*, 26(1):77–90, 2016. URL https://doi.org/10.1111/isj.12088.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

P. Cousot and R. Cousot. Abstract interpretation: Past, present and future. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 2:1–2:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2886-9. doi: 10.1145/2603088.2603165. URL http://doi.acm.org/10.1145/2603088.2603165.

P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10^{th} International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-33825-0. doi: 10.1007/978-3-642-33826-7_16. URL http://dx.doi.org/10.1007/978-3-642-33826-7_16.

C. J. Date. *Database in Depth: Relational Theory for Practitioners*. O'Reilly Media, Inc., 2005. ISBN 0596100124.

C. J. Date. *SQL and Relational Theory: How to Write Accurate SQL Code*. O'Reilly Media, Inc., 2011. ISBN 1449316409, 9781449316402.

T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48, 2013.

D. J. Dean. *The Visual Development of GCC Plug-ins with GDE*. PhD thesis, The Graduate School, Stony Brook University: Stony Brook, NY., 2009.

A. Dearle, G. N. C. Kirby, and R. Morrison. Orthogonal persistence revisited. In *ICOODB*. LNCS 5936, 2009. URL https://doi.org/10.1007/978-3-642-14681-7_1.

A. Dearle, G. N. C. Kirby, and R. Morrison. Orthogonal persistence revisited. *CoRR*, abs/1006.3448, 2010. URL http://arxiv.org/abs/1006.3448.

M. Delahaye, N. Kosmatov, and J. Signoles. Common specification language for static and dynamic analysis of C programs. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1230–1235, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1656-9. doi: 10.1145/2480362.2480593. URL http://doi.acm.org/10.1145/2480362.2480593.

J. Delons, N. Coulombel, and F. Leurent. PIRANDELLO an integrated transport and land-use model for the paris area. Technical report, 2008.

R. Di Cosmo and D. Nora. *Le Hold-up Planétaire: la face cachée de MicroSoft [the planetary hold-up: the hidden face of MicroSoft]*. Calman-Levy, 1998. ISBN 2-7021-2923-4. URL http://www.dicosmo.org/HoldUp/. [French book, out of print, but downloadable].

M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali. Cloud computing: Distributed internet computing for it and scientific research. *IEEE Internet computing*, 13(5):10–13, 2009.

V. Donzeau-Gouge, G. Huet, B. Lang, and G. Kahn. Programming environments based on structured editors : the MENTOR experience. Research Report RR-0026, INRIA, 1980. URL https://hal.inria.fr/inria-00076535.

F. Doucet, S. Shukla, and R. Gupta. Introspection in system-level language frameworks: Meta-level vs. integrated. In *Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1870-2. URL http://dl.acm.org/citation.cfm?id=789083.1022756.

J. Doyle. Expert systems and the" myth" of symbolic reasoning. *IEEE Transactions on Software Engineering*, 11:1386–1390, 1985.

P. Dragicevic, Y. Jansen, A. Sarma, M. Kay, and F. Chevalier. Increasing the transparency of research papers with explorable multiverse analyses. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI 19, pages 65:1–65:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300295. URL http://doi.acm.org/10.1145/3290605.3300295.

U. Drepper. How to write shared libraries. Technical report, Intel?, December 2011. URL https://software.intel.com/sites/default/files/m/a/1/e/dsohowto.pdf.

V. Echeverria, R. Martinez-Maldonado, and S. Buckingham Shum. Towards collaboration translucence: Giving meaning to multimodal group data. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 39. ACM, 2019. URL http://simon.buckinghamshum.net/wp-content/uploads/2019/02/CollabTranslucence_CHI2019.pdf.

J. Eisl, M. Grimmer, D. Simon, T. Würthinger, and H. Mössenböck. Trace-based register allocation in a jit compiler. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ 16, New York, NY, USA, 2016.

Association for Computing Machinery. ISBN 9781450341356. doi: 10.1145/2972206.2972211. URL https://doi.org/10.1145/2972206.2972211.

T. Fitz, M. Theiler, and K. Smarsly. A metamodel for cyber-physical systems. *Advanced Engineering Informatics*, 41:100930, 2019. ISSN 1474-0346. doi: https://doi.org/10.1016/j.aei.2019.100930. URL http://www.sciencedirect.com/science/article/pii/S1474034618306803.

P. Flach. *Machine Learning: The Art and Science of Algorithms that Make Sense of Data*. Cambridge University Press, 2012. doi: 10.1017/CBO9780511973000.

J.-M. Fouet. *Utilisation de connaissances pour améliorer l'utilisation de connaissances : la machine Gosseyn*. PhD thesis, Université Paris 6, 1987. URL http://www.theses.fr/1987PA066378. [using knowledge to improve usage of knowledge: the GOSSEYN machine].

J.-M. Fouet and B. Starynkevitch. Describing control. Technical Report FR8801379/CEA-CONF–9239, CEA, august 1987. URL https://inis.iaea.org/collection/NCLCollectionStore/_Public/19/059/19059867.pdf.

Free Software Foundation. GCC runtime library exception, 2009. URL https://www.gnu.org/licenses/gcc-exception-3.1.en.html.

Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 12(4):381–391, Dec. 1999. ISSN 1388-3690. doi: 10.1023/A:1010095604496. URL https://doi.org/10.1023/A:1010095604496.

D. M. Gabbay and P. Smets. *Handbook of defeasible reasoning and uncertainty management systems: algorithms for uncertainty and defeasible reasoning*, volume 5. Springer Science & Business Media, 2013. ISBN 0-7923-6672-7.

GCC Community. GCC internals, 2018. URL http://gcc.gnu.org/onlinedocs/gccint/.

S. Gerber. *Authorization, Protection, and Allocation of Memory in a Large System*. PhD thesis, ETH Zurich, 2018.

J. Giceva, G. Zellweger, G. Alonso, and T. Rosco. Customized OS support for data-processing. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, pages 1–6, 2016.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0-201-11371-6.

I. Gomes, P. Morgado, T. Gomes, and R. B. T. Moreira. An overview on the static code analysis approach in software development. 2009. URL https://www.semanticscholar.org/paper/An-overview-on-the-Static-Code-Analysis-approach-in-Gomes-Morgado/ce3c584c906eea668954f6a1a0ddbb295c6ec5a2.

K. Goseva-Popstojanova and A. Perhinschi. On the capability of static code analysis to detect security vulnerabilities. *Information and Software Technology*, 68:18–33, 2015.

E. Goubault and E. Haucourt. A practical application of geometric semantics to static analysis of concurrent programs. In *International Conference on Concurrency Theory*, pages 503–517. Springer, 2005.

E. Goubault and S. Putot. Static analysis of finite precision computations. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 232–247. Springer, 2011.

D. Graeber and A. Cerutti. *Bullshit jobs*. Simon & Schuster New York, NY, 2018. URL https://en.wikipedia.org/wiki/Bullshit_Jobs.

P. Graunke, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Modeling web interactions. In P. Degano, editor, *Programming Languages and Systems*, pages 238–252, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36575-4.

D. Greenaway, J. Lim, J. Andronick, and G. Klein. Don't sweat the small stuff: Formal verification of C code without the pain. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 429–439, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594296. URL http://doi.acm.org/10.1145/2594291.2594296.

R. Guerraoui and P. Kuznetsov. *Algorithms for Concurrent Systems*. EPFL press, 2018. ISBN 978-2-88915-283-4.

D. Guilbaud, E. Goubault, A. Pacalet, B. Starynkévitch, and F. Védrine. A simple abstract interpreter for threat detection and test case generation. In *WAPATV workshop*. ICSE, 2001. URL http://www.lix.polytechnique.fr/~goubault/papers/icse01.pdf.

E. Gulay and A. Lucero. Integrated workflows: Generating feedback between digital and physical realms. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI 19, pages 60:1–60:15, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-5970-2. doi: 10.1145/3290605.3300290. URL http://doi.acm.org/10.1145/3290605.3300290.

J. Gustedt. *MODERN C*. Manning Publications Company, 2019. URL https://modernc.gforge.inria.fr/.

H.-J. Happel and S. Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, pages 5–9. Citeseer, 2006.

I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan. The rise of big data on cloud computing: Review and open research issues. *Information systems*, 47:98–115, 2015.

D. Helbing, B. S. Frey, G. Gigerenzer, E. Hafen, M. Hagner, Y. Hofstetter, J. Van Den Hoven, R. V. Zicari, and A. Zwitter. Will democracy survive big data and artificial intelligence? In *Towards Digital Enlightenment*, pages 73–98. Springer, 2019. URL https://link.springer.com/chapter/10.1007/978-3-319-90869-4_7.

D. R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979. ISBN 0465026850.

G. J. Holzmann. The power of 10: rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006. URL https://ieeexplore.ieee.org/document/1642624.

U. Huws. iCapitalism and the Cybertariat: Contradictions of the digital economy. *Monthly review*, 66(8):42, 2015. URL http://monthlyreview.org/2015/01/01/icapitalism-and-the-cybertariat/.

M. Héder. From NASA to EU: the evolution of the TRL scale in public sector innovation. *Innovation Journal*, 22(2):1–23, 2017. URL https://core.ac.uk/download/pdf/94310086.pdf.

ISO. *C11 Standard*. 2011a. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf. ISO/IEC 9899:2011.

ISO. *C++11 Standard*. 2011b. URL https://github.com/cplusplus/draft/blob/master/papers/n3337.pdf. ISO/IEC 9899:2011.

I. Jacobs and L. Rideau-Gallot. a CENTAUR tutorial. Technical Report RT-140, INRIA Sophia-Antipolis, july 1992. URL ftp://www.inria.fr/pub/rapports/RT-140.ps.

A. K. Jain and S. Z. Li. *Handbook of face recognition*. Springer, 2011.

R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, Oct. 2009. ISSN 0360-0300. doi: 10.1145/1592434.1592438. URL http://doi.acm.org/10.1145/1592434.1592438.

R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 2nd edition, 2016. ISBN 1420082795, 9781420082791. URL http://gchandbook.org/.

T. Karvinen, K. Karvinen, and V. Valtokari. *Make: sensors: a hands-on primer for monitoring the real world with* ARDUINO *and* RASPBERRY PI. Maker Media, Inc., 2014.

B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988. ISBN 0131103709.

M. Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010. ISBN 1593272200, 9781593272203.

A. Khalilian, A. Baraani-Dastjerdi, and B. Zamani. CGenProg: Adaptation of cartesian genetic programming with migration and opposite guesses for automatic repair of software regression faults. *Expert Systems with Applications*, 169:114503, 2021. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2020.114503. URL http://www.sciencedirect.com/science/article/pii/S0957417420311477.

A. A. Khan, J. Keung, M. Niazi, S. Hussain, and M. Shameem. Gsepim: A roadmap for software process assessment and improvement in the domain of global software development. *Journal of Software: Evolution and Process*, 31(1), 2019. doi: 10.1002/smr.1988. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1988. e1988 JSME-18-0098.R1.

D. Khandal and S. Jain. Li-fi (light fidelity): The future technology in wireless communication. *International Journal of Information & Computation Technology*, 4(16):1–7, 2014.

B. Kiss, N. Kosmatov, D. Pariente, and A. Puccetti. Combining static and dynamic analyses for vulnerability detection: illustration on HEARTBLEED. In *Haifa Verification Conference*, pages 39–50. Springer, 2015. URL http://nikolai.kosmatov.free.fr/publications/kiss_kpp_hvc_2015.pdf.

B. Klimt and Y. Yang. The ENRON corpus: A new dataset for email classification research. In *European Conference on Machine Learning*, pages 217–226. Springer, 2004.

M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, et al. IMPALA: A modern, open-source sql engine for hadoop. In *Cidr*, volume 1, page 9, 2015.

Y. Kou and C. M. Gray. A practice-led account of the conceptual evolution of UX knowledge. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 49. ACM, 2019. URL https://www.researchgate.net/profile/Yubo_Kou/publication/330026000_A_Practice-Led_Account_of_the_Conceptual_Evolution_of_UX_Knowledge/links/5c2a7990a6fdccfc7074f0e2/A-Practice-Led-Account-of-the-Conceptual-Evolution-of-UX-Knowledge.pdf.

R. Kumar and R. Goyal. On cloud security requirements, threats, vulnerabilities and countermeasures: A survey. *Computer Science Review*, 33:1 – 48, 2019. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2019.05.002. URL http://www.sciencedirect.com/science/article/pii/S1574013718302065.

D. Lee, H. Moon, S. Oh, and D. Park. mIoT: Metamorphic iot platform for on-demand hardware replacement in large-scaled iot applications. *Sensors*, 20(12):3337, 2020. URL https://www.mdpi.com/1424-8220/20/12/3337.

R. Lee, M. Assante, and T. Conway. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 388, 2016. URL https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf.

D. B. Lenat. Eurisko: A program that learns new heuristics and domain concepts. *Artif. Intell.*, 21(1-2):61–98, Mar. 1983. ISSN 0004-3702. doi: 10.1016/S0004-3702(83)80005-8. URL http://dx.doi.org/10.1016/S0004-3702(83)80005-8.

D. B. Lenat and R. V. Guha. The evolution of cycl, the cyc representation language. *SIGART Bull.*, 2(3):84–87, June 1991. ISSN 0163-5719. doi: 10.1145/122296.122308. URL http://doi.acm.org/10.1145/122296.122308.

J. Lerner and J. Tirole. The simple economics of open source. Working Paper 7600, National Bureau of Economic Research, March 2000. URL http://www.nber.org/papers/w7600.

X. Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

X. Leroy et al. Ocaml site, 2018. URL http://ocaml.org/.

G. N. Levine. Defining defects, errors, and service degradations. *SIGSOFT Softw. Eng. Notes*, 34(2):1–14, Feb. 2009. ISSN 0163-5948. doi: 10.1145/1507195.1507205. URL http://doi.acm.org/10.1145/1507195.1507205.

H. Lipford, T. Thomas, B. Chu, and E. Murphy-Hill. Interactive code annotation for security vulnerability detection. In *Proceedings of the 2014 ACM Workshop on Security Information Workers*, SIW '14, pages 17–22, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3152-4. doi: 10.1145/2663887.2663901. URL http://doi.acm.org/10.1145/2663887.2663901.

V. László, G. Eberhardt, J. Burghardt, J. Gerlach, F. Stornig, E. Querrec, I. Greiro Santos, and A. M. Blanchar. Security requirements for connected medium security-critical applications (Vessedia D1.1). Technical report, SLAB, 2017. URL https://vessedia.eu/downloads/VESSEDIA-D1.1-Security-requirements-PU-M09.pdf. deliverable of Vessedia H2020 project, grant agreement No 731453.

I. G. Maglogiannis. *Emerging artificial intelligence applications in computer engineering: real world AI systems with applications in ehealth, hci, information retrieval and pervasive technologies*, volume 160. Ios Press, 2007. ISBN 978-1-58603-780-2.

C. Maria. Persistent cohomology. In *GUDHI User and Reference Manual*. GUDHI Editorial Board, 3.2.0 edition, 2020. URL https://gudhi.inria.fr/doc/3.2.0/group__persistent__cohomology.html.

C. Maria, P. Dlotko, V. Rouvreau, and M. Glisse. Rips complex. In *GUDHI User and Reference Manual*. GUDHI Editorial Board, 3.2.0 edition, 2020. URL https://gudhi.inria.fr/doc/3.2.0/group__rips__complex.html.

D. McLaren and J. Agyeman. *Sharing cities: a case for truly smart and sustainable cities*. MIT press, 2015.

M. Medwed. Iot security challenges and ways forward. In *Proceedings of the 6th International Workshop on Trustworthy Embedded Devices*, TrustED '16, pages 55–55, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4567-5. doi: 10.1145/2995289.2995298. URL http://doi.acm.org/10.1145/2995289.2995298.

K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of C: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 1–15, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. URL http://doi.acm.org/10.1145/2908080.2908081.

A. Miné and D. Delmas. Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. In *Proceedings of the 12th International Conference on Embedded Software*, EMSOFT '15, pages 65–74, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4673-8079-9. URL http://dl.acm.org/citation.cfm?id=2830865.2830873.

A. Miné, A. Ouadjaout, and M. Journault. Design of a modular platform for static analysis. In *Proc. 9th Workshop on Tools for Automatic Program Analysis*, august 2018. URL https://hal.archives-ouvertes.fr/hal-01870001.

M. Mitchell, J. Oldham, and A. Samuel. *Advanced Linux Programming*. New Riders Publishing, 2001. ISBN 0-7357-1043-0. URL http://www.makelinux.net/alp/.

B. Myers, S. E. Hudson, R. Pausch, and R. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, Mar. 2000. ISSN 1073-0516. doi: 10.1145/344949.344959. URL http://www.cs.cmu.edu/~amulet/papers/futureofhciACM.pdf.

M. Nadeem, B. J. Williams, and E. B. Allen. High false positive detection of security vulnerabilities: A case study. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 359–360, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1203-5. doi: 10.1145/2184512.2184604. URL http://doi.acm.org/10.1145/2184512.2184604.

F. Nagle. Learning by contributing: Gaining competitive advantage through contribution to crowdsourced public goods. *Organization Science*, 29(4):569–587, 2018. URL https://doi.org/10.1287/orsc.2018.1202.

A. D. Nicola, M. Missikoff, and R. Navigli. A software engineering approach to ontology building. *Information Systems*, 34(2):258 – 275, 2009. ISSN 0306-4379. doi: https://doi.org/10.1016/j.is.2008.07.002. URL http://www.sciencedirect.com/science/article/pii/S0306437908000628.

R. Nouira and J.-M. Fouet. A knowledge based tool for the incremental construction, validation and refinement of large knowledge bases. In *Workshop proceedings ECAI96*. Citeseer, 1996. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.8230&rep=rep1&type=pdf.

C. O'Neil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Broadway Books, 2016. ISBN 9780553418811. URL https://en.wikipedia.org/wiki/Weapons_of_Math_Destruction.

M. Payer. How memory safety violations enable exploitation of programs. In P. Larsen and A.-R. Sadeghi, editors, *The Continuing Arms Race*, pages 1–23. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2018. ISBN 978-1-97000-183-9. doi: 10.1145/3129743.3129745. URL https://doi.org/10.1145/3129743.3129745.

D. Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5:1–1, 2000.

H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 426–437, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813604. URL http://doi.acm.org/10.1145/2810103.2813604.

B. C. Pierce. *Types and programming languages*. MIT press, 2002. ISBN 0-262-16209-1.

J. Pitrat. *Méta-connaissances: futur de l'intelligence artificielle [meta-knowledge, future of artificial intelligence]*. Hermès, 1990. ISBN 9782866012472. [French book].

J. Pitrat. Implementation of a reflective system. *Future Generation Computer Systems*, 12(2):235 – 242, 1996. ISSN 0167-739X. doi: https://doi.org/10.1016/0167-739X(96)00011-8. URL http://www.sciencedirect.com/science/article/pii/0167739X96000118.

J. Pitrat. *De la machine à l'intelligence [from machine to intelligence]*. Hermès, 2000. ISBN 978-2866014742.

J. Pitrat. A step toward an artificial artificial intelligence scientist. Technical report, CNRS and LIP6 Université Paris, 2009a. URL https://pdfs.semanticscholar.org/2117/9600b3f05c0af399f9acbfc6e7b6d24daf03.pdf.

J. Pitrat. *Artificial Beings: The Conscience of a Conscious Machine*. Wiley ISTE, 2009b. ISBN 978-1-848-21101-8.

J. Pitrat. My view on artificial intelligence. blog, 2013-2019. URL http://bootstrappingartificialintelligence.fr/WordPress3/.

G. Polito, S. Ducasse, L. Fabresse, N. Bouraqadi, and B. van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Science of Computer Programming*, 96:141 – 155, 2014. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2013.10.008. URL http://www.sciencedirect.com/science/article/pii/S0167642313002797. Special issue on Advances in Smalltalk based Systems.

S. Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. Thèse de doctorat de l'École des mines de paris, École Nationale Supérieure des Mines de Paris, Dec. 2006. URL https://pastel.archives-ouvertes.fr/pastel-00002281.

F. Pérez, T. Ziad, and C. Cetina. Utilizing automatic query reformulations as genetic operations to improve feature location in software models. *IEEE Transactions on Software Engineering*, 2020. URL https://hal.sorbonne-universite.fr/hal-02852488/.

C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-56247-3. URL https://pages.lip6.fr/Christian.Queinnec/WWW/LiSP.html.

C. Queinnec. Continuations and web servers. *Higher Order Symbol. Comput.*, 17(4):277–295, Dec. 2004. ISSN 1388-3690. doi: 10.1007/s10990-004-4866-z. URL https://core.ac.uk/download/pdf/81910787.pdf.

P. Raj. Chapter one - a detailed analysis of *NoSQL* and *NewSQL* databases for bigdata analytics and distributed computing. In P. Raj and G. C. Deka, editors, *A Deep Dive into* NoSQL *Databases: The Use Cases and Applications*, volume 109 of *Advances in Computers*, pages 1 – 48. Elsevier, 2018. doi: https://doi.org/10.1016/bs.adcom.2018.01.002. URL http://www.sciencedirect.com/science/article/pii/S0065245818300020.

E. S. Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001. ISBN 0596001088.

J. C. Reynolds. The discoveries of continuations. *Lisp & Symbolic Computation*, 6(3-4):233–248, Nov. 1993. ISSN 0892-4635. doi: 10.1007/BF01019459. URL http://dx.doi.org/10.1007/BF01019459.

C. Rich and R. C. Waters. *Readings in artificial intelligence and software engineering*. Morgan Kaufmann, 2014.

P. Rodriguez, M. Mäntylä, M. Oivo, L. E. Lwakaratare, P. Seppänen, and P. Kuvaja. Advances in using agile and lean processes for software development. In A. M. Memon, editor, *Advances in Computers*, volume 113, chapter 4, pages 135–221. Academic Press, 2018. ISBN 978-0-12-816070-1. URL https://doi.org/10.1016/bs.adcom.2018.03.014.

I. Rus, M. Lindvall, and S. Sinha. Knowledge management in software engineering. *IEEE software*, 19(3): 26–38, 2002.

D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge university press, 2003. ISBN 0-521-7787177-9.

S. Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Springer Nature, 2020. ISBN 978-1-4842-4932-1.

C. E. Schafmeister. CLASP - a Common Lisp that interoperates with C++ and uses the LLVM backend. In *Proceedings of the 8th European Lisp Symposium on European Lisp Symposium*, ELS2015, 2015.

C. E. Schafmeister. CANDO: A compiled programming language for computer-aided nanomaterial design and optimization based on Clasp Common Lisp. In *Proceedings of the 9th European Lisp Symposium on European Lisp Symposium*, ELS2016, pages 9:75–9:82. European Lisp Scientific Activities Association, 2016. ISBN 978-2-9557474-0-7. URL http://dl.acm.org/citation.cfm?id=3005729.3005738.

B. Schlich. Model checking of software for microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4): 36:1–36:27, Apr. 2010. ISSN 1539-9087. doi: 10.1145/1721695.1721702. URL http://doi.acm.org/10.1145/1721695.1721702.

A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27, 2008.

M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 4th edition, 2007. ISBN 978-0124104099.

K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2342821.2342849.

S. Shalev-Shwartz and S. Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014. ISBN 978-1107-05713-5.

J. H. Siddiqui, A. Rauf, and M. A. Ghafoor. Advances in software model checking. In A. M. Memon, editor, *Advances in Computers*, volume 108, chapter 2, pages 59–89. Academic Press, 2018. ISBN 978-0-12-815119-8.

N. Silva and M. Vieira. Software for embedded systems: A quality assessment based on improved odc taxonomy. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 1780–1783, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3739-7. doi: 10.1145/2851613.2851908. URL http://doi.acm.org/10.1145/2851613.2851908.

B. Starynkevitch. *Expliciter et utiliser les données et le contrôle pour les connaissances par des metaconnaissances : systèmes* EUM1 *et* EUM2. PhD thesis, Université Paris 6 - LIP6, december 1990. URL http://www.theses.fr/1990PA066799. [explicitation and use of data and control for knowledge with metaknowledge : the EUM1 and EUM2 systems].

B. Starynkevitch. Multi-stage construction of a global static analyzer. In *GCC summit*, pages 143–152, Ottawa, Canada, july 2007. GCC. URL http://starynkevitch.net/basile/gccsummit2007-starynkevitch.pdf.

B. Starynkevitch. GCC MELT website (archive), 2008-2016. URL http://starynkevitch.net/Basile/gcc-melt/. (archive of the old gcc-melt.org site).

B. Starynkevitch. A middle end lisp translator for GCC. In *GCC Research Opportunities workshop*, 2009. URL http://www.doc.ic.ac.uk/~phjk/GROW09/papers/01-MELT-Starynkevitch.pdf.

B. Starynkevitch. MELT - a translated domain specific language embedded in the GCC compiler. In *DSL2011 IFIP conf.*, Bordeaux (France), Sept. 2011. URL http://adsabs.harvard.edu/abs/2011arXiv1109.0779S.

G. L. Steele, Jr. *Common LISP: The Language*. Digital Press, Newton, MA, USA, 2nd edition, 1990. ISBN 1-55558-041-6.

B. Stroustrup. *Programming: Principles and Practice Using C++*. Addison-Wesley Professional, 2nd edition, 2014. ISBN 0321992784, 9780321992789.

B. Stroustrup. Thriving in a crowded and changing world: C++ 2006–2020. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–168, 2020. URL https://dl.acm.org/doi/pdf/10.1145/3386320.

A. S. Tanenbaum. *Modern operating systems*. Prentice Hall, Englewood Cliffs, N.J., 1992.

D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the CONDOR experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 3.2.0 edition, 2020. URL https://gudhi.inria.fr/doc/3.2.0/.

J. Tirole. *Économie du bien commun*. Presses Universitaires de France, 2018. ISBN 978-2-13-080766-7. in French: *economics of common goods*.

D. Ungar and R. B. Smith. SELF: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, Dec. 1987. ISSN 0362-1340. doi: 10.1145/38807.38828. URL http://doi.acm.org/10.1145/38807.38828.

R. Van De Riet. Linguistic instruments in knowledge engineering. In *Proc.1991 Workshop on Linguistic Instruments in Knowledge Engineering*. North Holland, 1992. ISBN 978 0444883940.

R. Vedala and S. A. Kumar. Automatic detection of printf format string vulnerabilities in software applications using static analysis. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 379–384, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1185-4. doi: 10.1145/2381716.2381787. URL http://doi.acm.org/10.1145/2381716.2381787.

P. Voigt and A. Von dem Bussche. The *EU* GENERAL DATA PROTECTION REGULATION. *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 2017.

S. Wagner. Defect classification and defect types revisited. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, DEFECTS '08, pages 39–40, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-051-7. doi: 10.1145/1390817.1390829. URL http://doi.acm.org/10.1145/1390817.1390829.

A. S. Waterman. *Design of the RISC-V instruction set architecture*. PhD thesis, UC Berkeley, 2016.

S. Weber. *The Success of Open Source*. Harvard University Press, Cambridge, MA, USA, 2004. ISBN 0674012925.

X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1):97–107, 2013.

Y. Zhang and Y. Huang. "learned" operating systems. *ACM SIGOPS Operating Systems Review*, 53(1):40–45, 2019.

S. Zuboff. Big other: surveillance capitalism and the prospects of an information civilization. *Journal of Information Technology*, 30(1):75–89, 2015. URL https://cryptome.org/2015/07/big-other.pdf.

For books in French, I have provided a *tentative* translation into English of their title in brackets.

About this document

To produce this document, both in PDF and HTML forms : build *bismon*[a] on your Linux computer, then run `make doc`[b] or just `make latexdoc` to get only its PDF form.

Feedback and improvements on this document can be suggested by email (to **basile@starynkevitch.net** or **basile.starynkevitch@cea.fr**) or by submitting patches to *Bismon* thru its **https://github.com/bstarynk/bismon** repository (or directly by email, with your permission to include it). Notice that this document may contain generated documentation, and will contain more and more generated parts in the future.

[a]See the README.md file on **https://github.com/bstarynk/bismon/** for building instructions.
[b]That uses LaTeX and *HeVeA*. HTML generation might not work in summer 2019.

Acknowledgements

Thanks to my colleague Franck Védrine, to several members of the CHARIOT consortium who have proofread this report, and to Niklas Rosencratz (from Sweden, outside of the consortium) who voluntarily found mistakes in it and proposed, in the repository on **https://github.com/montao/bismon-docker/**, a `Dockerfile` for *bismon*.

Thanks also to Jérémie Salvucci (France) and Abhishek Chakravarti (India) for many valuable questions, suggestions, and discussions -in numerous audioconference (or face to face) exchanges, or in private emails-, about reflexive persistent systems in general, and more specifically about *Bismon*. Both did suggest several improvements to this report and to the *Bismon* software.

The CC-BY-SA license below for this (CHARIOT D1.3^{v2}) deliverable is required to enable major parts of this report to be later incorporated into a proper *bismon* free software documentation. For example, the Debian Linux distribution has a policy [211] strongly recommending a specific set of licenses (notably CC-BY-SA) for documentation. Using other (deemed proprietary) licenses in free software documentation is decreasing the future chances of such documentation being later incorporated in Linux distributions.

(git commit `61e424abbb482e7e++`; generated on *Mon 01 Feb 2021 03:26:24 PM MET*)

Some variant of this draft report is downloadable from **http://starynkevitch.net/Basile/bismon-doc.pdf** and elsewhere.

[211]See Debian documentation project on **https://www.debian.org/doc/ddp** etc...