Cognitive Heterogeneous Architecture for Industrial IoT

D1.3 v2- Specialized Static Analysis tools for more secure and safer IoT software development.

Bismon documentation.

Document Summary Information

<table>
<thead>
<tr>
<th>Grant Agreement</th>
<th>Acronym</th>
<th>CHARIOT</th>
</tr>
</thead>
<tbody>
<tr>
<td>780075</td>
<td>Cognitive Heterogeneous Architecture for Industrial IoT</td>
<td></td>
</tr>
</tbody>
</table>

Start date: 01/01/2018  
Duration: 36 months

Project URL: [www.chariotproject.eu](http://www.chariotproject.eu)

Deliverable: D1.3 Specialized Static Analysis tools for more secure and safer IoT software development.

Workpackage: WP1

Dissemination level: Public report

Contractual due date: 31/06/2020 (M30)  
Actual submission date: -

Nature: R (report)  
dissemination level: PU (public)

Lead beneficiary: CEA, LIST

Responsible author: Basile STARYNKEVITCH (CEA, LIST)

Contributions from: -

Revision: git commit a2a88e6e0a50bd13 on 2019-Aug-23

\LaTeX-generated: Fri 23 Aug 2019 03:04:17 PM MEST

This CHARIOT project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Grant Agreement No 780075.
Revision history (including peer reviewing & quality control)

<table>
<thead>
<tr>
<th>Version</th>
<th>Issue date</th>
<th>% Complete</th>
<th>Changes</th>
<th>Contributor(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Applicable</td>
<td>- see explanation below</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This partly generated document and the Bismon software itself are co-developed in an agile and incremental manner, and have exactly 3191 git commits on Fri 23 Aug 2019 03:04:17 PM MEST. 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 \footnote{The generated parts are clearly identified as such, and are extracted from the Bismon system.}, 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 a2a88e6e0a50bd13 (done on 2019-Aug-23) generated on Aug 23, 2019, exactly 3191 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 two commit-s:

commit a2a88e6e0a50bd13269f213909f9675a12d32e33
Author: Basile Starynkevitch <basile@starynkevitch.net>
Date: Fri Aug 23 15:03:27 2019 +0200
   chariot report: improve conclusion
M   doc/conclus-bm.tex

commit 220c69ff845394fb66105fa5c37e4feb4f1593d
Author: Basile Starynkevitch <basile@starynkevitch.net>
Date: Fri Aug 23 14:30:11 2019 +0200
    improve comment in test-jsonextract.sh
M   misc/test-jsonextract.sh

commit 3de6f938215ae8b816e41299645e6373359c16b
Author: Basile Starynkevitch <basile@starynkevitch.net>
Date: Fri Aug 23 13:58:16 2019 +0200
   chariot report: the index works again
M   doc/bismon-chariot-doc.tex

There is no notion of any identifiable “version” in bismon, so also in this D1.3 v2- deliverable. The work is incremental and the development is agile.

disclaimer

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. While the information contained in the documents is believed to be accurate, the authors(s) or any other participant in the CHARIOT consortium make no warranty of any kind with regard to this material including, but not limited to the implied warranties of merchantability and fitness for a particular purpose. Neither the CHARIOT Consortium nor any of its members, their officers, employees or agents shall be responsible or liable in negligence or otherwise howsoever in respect of any inaccuracy or omission herein. Without derogating from the generality of the foregoing neither the CHARIOT Consortium nor any of its members, their officers, employees or agents shall be liable for any direct or indirect or consequential loss or damage caused by or arising from any information advice or inaccuracy or omission herein.

\footnote{Obtained by the \texttt{git log -name-status -3} command running in bismon top source directory.}
Executive Summary

This D1.3\textsuperscript{v1} CHARIOT deliverable is (was) a first \textbf{draft} -and \textit{preliminary}- version (at M12) of D1.3\textsuperscript{v2} scheduled at M30 on “Specialized Static Analysis tools for more secure and safer IoT software development”.

But \textbf{incremental work has begun on D1.3\textsuperscript{v2} in early 2019}, hence the D1.3\textsuperscript{v2} in the header.

It describes the CHARIOT vision on static source code (mostly of C and C++ code for IoT firmware and application) analysis. It proposes a \textit{simple} static analysis \textit{framework} leveraging on the powerful recent \textit{GCC} [cross-]compiler. A \textit{persistent monitor} (tentatively named \textit{bismon}) is being designed and implemented as a GPLv3+ free software for Linux, using meta-programming techniques (leveraging on experience gained in the former GCC MELT project) to \textit{generate} GCC plugins, and able to keep some intermediate results (of compilation or static analysis) during the entire life of the IoT project, and giving to the IoT developers (thrue a web interface) a whole-program view of the source code (as digested by the GCC cross-compiler) and of its static analysis properties. That framework is configurable and scriptable by static analysis experts, hence permitting different IoT projects to address various concerns, while keeping the usual IoT development workflow (running as usual their GCC cross-compiler on Linux, with extra plugin-related compilation flags). The deliverable has been structured starting from the identification of the software and tool users and the document expected audience as well as the vision on specialized source code analysis towards more secure and safer IoT software development. The report then describes its strong alignment to adding capabilities to GCC as well as the driving principles of the tools. Data and their persistence character are also described including mutable and non-mutable values/types while persistence is considered to start by loading some previous persisted state, usually dumping its current state before termination and loading the next state on the next load-up. The framework for static code analysis is also defined as part of the GCC compilation process. The described work is also analysed in terms of contributing to other free software projects.

notice

This work is funded\textsuperscript{3} (from start of 2018 to end of 2020) thru the Chariot project (see its web site on \url{http://chariotproject.eu/}) which has received funding from the European Union’s Horizon 2020 research and innovation programme under the Grant Agreement No 780075.

\textbf{Deliverable created during the CHARIOT project.}

\textsuperscript{3}The development work of the \textit{bismon} free software prototype is also \textit{partly} funded -from 2019 to 2021- by the \textbf{DECODER} H2020 project, under its Grant Agreement 824231.
Contents

1 Introduction 4
  1.1 Mapping CHARIOT output .................................................. 4
  1.2 Deliverable Overview and Report Structure .......................... 4
  1.3 Expected audience .......................................................... 5
  1.4 The CHARIOT vision on specialized static source code analysis for more secure and safer IoT software development ........................................... 6
    1.4.1 About static source code analysis and IoT ...................... 6
    1.4.2 The power of an existing compiler: GCC ....................... 10
    1.4.3 Leveraging simple static source analysis on GCC .......... 16
  1.5 Lessons learned from GCC MELT ....................................... 18
  1.6 Driving principles for the Bismon persistent monitor .......... 18
    1.6.1 About Bismon ......................................................... 18
    1.6.2 About Bismon as a domain-specific language ................ 21
    1.6.3 About Bismon as a evolving software system ............... 22
    1.6.4 About Bismon as a static source code analyzer framework 24
  1.7 Multi-threaded and distributed aspects of Bismon ............. 25

2 Data and its persistence in Bismon 27
  2.1 Data processed in Bismon .................................................. 27
    2.1.1 Immutable values .................................................. 27
    2.1.2 Mutable objects .................................................. 28
  2.2 garbage collection of values and objects ......................... 30
  2.3 persistence in Bismon ................................................... 31
    2.3.1 file organization of the persistent state .................. 32
    2.3.2 persisting objects ............................................... 32

3 Static analysis of source code in Bismon 35
  3.1 static analysis of GCC code ............................................ 35
  3.2 static analysis of IoT firmware or application code .......... 35

4 Using Bismon 37
  4.1 How JSON is used by Bismon ............................................ 37
    4.1.1 The canonical JSON encoding of Bismon values ............. 37
    4.1.2 The nodal JSON decoding into Bismon values ............... 38
    4.1.3 JSON extraction with extract_json .......................... 38
  4.2 Web interface internal design ......................................... 39
  4.3 Using bismon for CHARIOT ............................................. 44

5 Miscellaneous work 45
  5.1 Contributions to other free software projects ................ 45
    5.1.1 Aborted contribution to libonion ............................. 45
    5.1.2 Contribution to GCC ............................................ 45
  5.2 Design and implementation of the compiler and linker extension 45

6 Conclusion 46

Index 47

References 47
List of Figures

1. the CHARIOT compilation of some IoT firmware or application (simplified) ........................................... 9
2. The Bismon monitor used by some IoT developer team following the CHARIOT approach. .................. 17
3. crude (soon deprecated) GTK interface oct. 22, 2018, git commit cbdcf1ec351c3f2a .......................... 23
4. generated dump example: first_test_module in file store2.bmon ......................................................... 33
5. syntax of values in dumped data files. ...................................................................................................... 33

List of Tables

1. adherence to CHARIOT’s GA deliverable and tasks descriptions .......................................................... 4
2. recursive inlining with constant folding in GCC (C source) ................................................................. 10
3. recursive inlining with constant folding in GCC (generated early Gimple) ....................................... 11
4. recursive inlining with constant folding in GCC (generated SSA form) ............................................... 12
5. recursive inlining with constant folding in GCC (generated optimized) ...................................................... 13
6. recursive inlining with constant folding in GCC (generated MIPS assembler) ....................................... 13
7. optimization around heap allocation by GCC (C source) ................................................................. 13
8. optimization around heap allocation by GCC (generated Gimple) .................................................. 15
9. optimization around heap allocation by GCC (generated SSA/optimized) ......................................... 15
10. optimization around heap allocation by GCC (generated x86-64 assembly) ..................................... 16
11. canonical JSON encoding $\text{json}[[v]]$ of a Bismon value $v$. ................................................................. 37
12. nodal JSON decoding $\text{nod}[[\text{js}]]$ of a JSON value $\text{js}$. ................................................................. 38
13. simple extraction from some JSON thing $\text{js}$; (see also table 14 below.) ............................................. 39
14. complex extraction from some JSON thing $\text{js}$; (see also table 13 above.) ............................................. 40
### Glossary of terms and abbreviations used

<table>
<thead>
<tr>
<th>Abbreviation / Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>binutils</code></td>
<td>GNU free software package containing assembler <code>as</code>, linker <code>ld</code> and other utilities to operate on object files or binary executables, etc... <a href="https://www.gnu.org/software/binutils/">https://www.gnu.org/software/binutils/</a></td>
</tr>
<tr>
<td><code>bismon</code></td>
<td>the free software framework and persistent monitor described here; source repository on <a href="http://github.com/bstarynk/bismon/">http://github.com/bstarynk/bismon/</a></td>
</tr>
<tr>
<td><code>Clang</code></td>
<td>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 <a href="http://clang.llvm.org/">http://clang.llvm.org/</a></td>
</tr>
<tr>
<td><code>GCC MELT</code></td>
<td>was a (GPLv3+-licensed) GCC plugin and framework providing a DSL to ease GCC extensions; it is archived on <a href="http://starynkevitch.net/Basile/gcc-melt/">http://starynkevitch.net/Basile/gcc-melt/</a></td>
</tr>
<tr>
<td><code>Generic</code></td>
<td>language-independent abstract syntax tree (internal representation) in GCC</td>
</tr>
<tr>
<td><code>Gimple</code></td>
<td>middle-end internal representation in GCC</td>
</tr>
<tr>
<td><code>GPL</code></td>
<td>Gnu General Public Licence (a copylefted free software license) <a href="https://www.gnu.org/licenses/gpl.html">https://www.gnu.org/licenses/gpl.html</a></td>
</tr>
<tr>
<td><code>IoT</code></td>
<td>Internet of Things</td>
</tr>
<tr>
<td><code>libonion</code></td>
<td>an HTTP server library <a href="https://www.coralbits.com/libonion/">https://www.coralbits.com/libonion/</a></td>
</tr>
<tr>
<td><code>LLVM</code></td>
<td>The LLVM Project is an open-source collection of modular and reusable compiler and toolchain technologies <a href="http://www.llvm.org/">http://www.llvm.org/</a></td>
</tr>
<tr>
<td><code>MELT</code></td>
<td>the Lisp-like domain specific language used in GCC MELT</td>
</tr>
<tr>
<td><code>Persistence</code></td>
<td>From Wikipedia: “In computer science, <strong>persistance</strong> 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”. <a href="https://en.wikipedia.org/wiki/Persistence_(computer_science)">https://en.wikipedia.org/wiki/Persistence_(computer_science)</a></td>
</tr>
<tr>
<td><code>RTL</code></td>
<td>(register transfer language) back-end internal representation in GCC</td>
</tr>
<tr>
<td>static code analysis</td>
<td>(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: <a href="https://en.wikipedia.org/wiki/Static_program_analysis">https://en.wikipedia.org/wiki/Static_program_analysis</a>). In this D1.3v1 report, it means static source code analysis, in practice analysis of C or C++ code for IoT fed to the GCC compiler.</td>
</tr>
<tr>
<td><code>SSA</code></td>
<td>Static Single Assignment (in GCC, a variant of Gimple)</td>
</tr>
</tbody>
</table>
1 Introduction

This D1.3 v1 CARIOT 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 CARIOT output

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

<table>
<thead>
<tr>
<th>CARIOT GA component title</th>
<th>CARIOT GA component outline</th>
<th>respective document chapter[s]</th>
<th>justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>deliverable</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D1.3</td>
<td>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 adaptation of existing libraries and software components, the persistent monitor outline and documentation, and 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.</td>
<td>this whole document</td>
<td>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 miscellaneous work. The conclusion is in §6.</td>
</tr>
<tr>
<td>tasks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T1.3</td>
<td>ST1.3.1 definition of data formats and protocols</td>
<td>§2; §5.2</td>
<td>The persistent monitor data and format are described in §2. The protocol to interact with CARIOT’s blockchain is related to §5.2 and chapter 6 of D1.2</td>
</tr>
<tr>
<td></td>
<td>ST1.3.2 significant patches to existing free software components</td>
<td>§5.1</td>
<td>Section §5.1 describe past work, and why future contributions to GCC could be needed.</td>
</tr>
<tr>
<td></td>
<td>ST1.3.3 design and implementation of the persistent monitor</td>
<td>$1.4; $1.6; $1.7; §2; $1.4 gives the CARIOT 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.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ST1.3.4 design and implementation of the compiler and linker extension</td>
<td>§3; §5.2</td>
<td>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 v2) drafted in §5.2</td>
</tr>
</tbody>
</table>

1.2 Deliverable Overview and Report Structure

This CARIOT 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

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\textsuperscript{4} (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 miscellaneous 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]) and/or C++ (Stroustrup [2014]) programming (notably on Linux 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\textsuperscript{5} 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\textsuperscript{6}) on the command line.

- be easily able, in principle, to compile\textsuperscript{7} 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\textsuperscript{8} on Linux.

- to be capable of building large free software projects (such as the GCC compiler (cf GCC Community [2018]\textsuperscript{9}), the Linux kernel, the Qt toolkit and other open source projects of perhaps millions of source code lines) and smaller ones (e.g. libonion\textsuperscript{10}) from their source form.

- have successfully downloaded and built the Bismon monitor from its source code available on \url{https://github.com/bstarynk/bismon}, on his/her Linux workstation.

\textsuperscript{4}See \url{http://blog.regehr.org/archives/1520} and \url{https://blog.regehr.org/archives/1520}

\textsuperscript{5}See \url{https://www.latex-project.org/}. Some knowledge of \LaTeX is useful to improve or contribute to this document.

\textsuperscript{6}When compiling IoT software such as firmware, it is usually of course some cross-compilation.

\textsuperscript{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).

\textsuperscript{8}See \url{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.

\textsuperscript{9}See \url{https://github.com/bstarynk/bismon}, on his/her Linux workstation.
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 [2016]; Nagle [2018]; Di Cosmo and Nora [1998]; Lerner and Tirole [2000]) implications, the practical work flow, the importance of developer communities and of business support.

be interested in static source code analysis, so have already tried some such tools like Frama-C (cf. Cuq et al. [2012]), Clang analyzer, ..., and be aware of compiler concepts and technologies (read Aho et al. [2006]).

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]).

be interested in various programming languages (cf. Abelson et al. [1996]; Scott [2007]; Queinnec [1996]), including domain specific ones.

is aware that most software projects fail (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 to any non-trivial software developed by humans (because of Braun et al. [1956], of "leaky abstractions" and of the Halting problem, etc...), and that formal methods approaches are still vulnerable to specification bugs. Agile and lean approaches could be effective for improving IoT software development processes (see Rodriguez et al. [2018]).

is understanding the notion of Technical Readiness Level (TRL) and its implication in innovative projects, notably H2020 ones (see Héder [2017]).

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]), 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. 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

11See 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?
12See https://frama-c.com/
13See https://clang-analyzer.llvm.org/
14look into man pages on http://man7.org/linux/man-pages/
15See https://www.geneca.com/why-up-to-75-of-software-projects-will-fail/
16IMHO, 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.
18A 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-178C etc... But in mid-2019 this is only a speculation (details are unknown to me).
19The 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.
warnings purposes (and that is why C compilers are monsters\textsuperscript{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\textsuperscript{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 COCCINELLE), very few Debian packages -coded in C or C++ for example- are conditionally “build-depending”\textsuperscript{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\textsuperscript{23} used in many non-critical embedded and connected IoT systems.

Static source code analysis tools can -generally speaking- be\textsuperscript{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 \textsuperscript{25}. For examples, FRAMA-C (cf Cuq 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\textsuperscript{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\textsuperscript{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\textsuperscript{28} or Clang-Analyzer, or even

\textsuperscript{20}See \url{https://softwareengineering.stackexchange.com/a/273711/40065} for more.
\textsuperscript{21}Se we can only hope an incremental progress in that area. Static source code analysis in CHARIOT won’t make miracles.
\textsuperscript{22}The Debian packaging system is sophisticated enough to just suggest a tool useful to build a package from its source code.
\textsuperscript{23}The Raspbian distribution is a typical example, see \url{https://raspbian.org/} for more. But look also into \url{https://www.automotivelinux.org/} or \url{https://www.genivi.org/} as another examples.
\textsuperscript{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).
\textsuperscript{25}There are cases where those static analyzers need weeks of computer time to give interesting results.
\textsuperscript{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.
\textsuperscript{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.
\textsuperscript{28}See \url{https://scan.coverity.com/}
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]) 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 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

---

\(^{29}\)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/](http://valgrind.org/), and be aware that valgrind is capable of runtime checking many other properties, such as some 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 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](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.

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 languages41: when a non-critical equipment 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 equipment 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 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. 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 some licensing issues on the obtained proprietary binary firmware blob, if it was compiled with the help of some hypothetical proprietary GCC plugin.

![Diagram of CHARIOT compilation process](image)

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

---

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


42 The actual version and the concrete configuration of GCC are important; we want to stick -when reasonably possible- to the latest GCC releases, e.g. to GCC 8 in autumn 2018. 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.
1.4.2 The power of an existing compiler: GCC

CHARIOT static analysis tools will leverage on the mainstream GCC 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 of source code (in gcc-8.2.0, measured by sloccount). This figure is an under-estimation, 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 target specific.

---

Table 2: recursive inlining with constant folding

```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);
}
```

---

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 \(\tau = g(x)\) then \(a = f(\tau, y)\), etc...), shown in table 3.

This is a textual (and quite incomplete since a partial view of some) dump of some in-memory 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. The 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
Table 3: recursive inlining with constant folding in GCC (generated early Gimple)

```c
fact12 ()
{
    int D.1490;
    D.1490 = fact (12);
    return D.1490;
}

fact (int n)
{
    int D.1494;
    if (n <= 1) goto <D.1492>; else goto <D.1493>;
    <D.1492>:
    D.1494 = 1;
    // predicted unlikely by early return (on trees) predictor.
    return D.1494;
    <D.1493>:
    _1 = n + -1;
    _2 = fact (_1);
    D.1494 = n * _2;
    // predicted unlikely by early return (on trees) predictor.
    return D.1494;
}
```

exit edges). Then special $\phi$ nodes introduce places where such a variable may come from two other ones (after branches).

At last, many other optimizations happen. And the optimized form in table 5 shows that \texttt{fact12} just returns the constant 479001600 (which happens to be the result of \texttt{fact(12)} computed at compile-time).
Table 4: recursive inlining with constant folding in GCC (generated SSA form)

```
;;; Function fact (fact, funcdef_no=0, decl_uid=1484, cgraph_uid=0, symbol_order=0)
fact (int n)
{
  int _1;
  int _2;
  int _3;
  int _6;
  int _9;
  <bb 2> :  
  if (n_5(D) <= 1)
    goto <bb 3>; [INV]
  else
    goto <bb 4>; [INV]
  <bb 3> :  
  _6 = 1;
  // predicted unlikely by early return (on trees) predictor.
  goto <bb 5>; [INV]
  <bb 4> :  
  _1 = n_5(D) + -1;
  _2 = fact (_1);
  _9 = n_5(D) * _2;
  // predicted unlikely by early return (on trees) predictor.
  <bb 5> :  
    # _3 = PHI <_6(3), _9(4)>
    return _3;
}

;;; Function fact12 (fact12, funcdef_no=1, decl_uid=1487, cgraph_uid=1, symbol_order=1)
fact12 ()
{
  int _3;
  <bb 2> :  
  _3 = fact {12};
  return _3;
}
```

*Static Single Assignment (SSA) code*

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

```c
;; Function fact12 (fact12, funcdef_no=1, decl_uid=1487, cgraph_uid=1, 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)

```c
### 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: (Debian 8.3.0-19) 8.3.0"
```

---

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)

```c
/// 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;
}
```

---

50Similar optimizations also happen with a GCC MIPS targeted cross-compiler.
The straight GCC compiler \(^5^0\) (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.
Table 8: optimization around heap allocation by GCC (generated Gimple)

```c
weirdsum (int x, int y)
|
int D.2566;
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.2566 = r;
return D.2566;
}

// malloc.c.004f.gimple generated by ...
// ... -O3 -S -fverbose-asn \n// ... -fdump-tree-gimple -fdump-tree-ssa -fdump-tree-optimized malloc.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)

```c
:: Function weirdsum (weirdsum, funcdef_no=11,
::  decl_uid=2561, cgraph_uid=11, symbol_order=11)
weirdsum (int x, int y)
|
int r;

<bb 2> [ local count: 1073741825]:
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 code this way). However, a similar example might happen in real life after preprocessor expansion and inlining in large header-mostly libraries. 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)

```
## 62 lines removed
.text
  .p2align 4,.15
  .globl weirdsum
  .type weirdsum, @function
weirdsum:
  .LFB11:
    .cfi_startproc
    # mallfree.c:7: int r = ar2[0] + ar2[1];
    leal (%rdi,%rsi), %eax #, r
    # mallfree.c:10: }
    ret
  .cfi_endproc
  .LFE11:
  .size weirdsum, .-weirdsum
  .ident "GCC: (Debian 8.3.0-19) 8.3.0"
  .section .note.GNU-stack,"",@progbits
```

### 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 mips-linux-gnu-gcc-8 or arm-linux-gnuabi-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.

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\(^{55}\).

---

\(^{51}\)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 responsibility 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, meson, cmake, 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 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.
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. 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.

---

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.
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.

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

\begin{itemize}
\item including of course the API related to internal GCC representations, such as Gimple or SSA.
\item This is the internal GCC garbage collector, a mark-and-sweep one which can run only between GCC passes.
\item Current GCC-8 have several thousands GTY-ed classes.
\item Notice bismon is a temporary name and could be changed, once we find a better name for it. Suggestions for better names are welcome.
\item The source code is unreleased but available, and continuously evolving, on https://github.com/bstarynk/bismon
\item 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
\item 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
\item 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
\end{itemize}
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** implementation, targetted to ease static source code analysis (above the **GCC** compiler), with the following features:

- **persistence**, 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 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 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**, like many scripting languages (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 better one, such as Ravenbrook MPS; something better, if good performance and scalability is wanted in *Bismon*. The hard point is the multi-threaded aspect of mutator threads in the 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** and reflection with introspection (cf Pitrat [1996, 1990, 2009a,b]): all the DSL code is explicitly represented as data which can be processed by *Bismon*, and the current state is accessible by the DSL.

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

- **bootstrapped** 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**[^1], used by a small team of trusted and well-behaving developers[^2]. The users of *bismon* are expected to trust each other, and to use the *bismon* tool responsibly[^3] (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)^[^4], 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) is out of scope and should be in charge of the entities and/or persons using and/or deploying *Bismon*. The login form template[^5] 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...).

- **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[^6]. 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). 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[^7] of data, web-based collaborative software tools, various multi-user web applications, declarative programming approaches, etc...

[^1]: See also the web application wikipage, but *bismon* has highly specific peculiarities, detailed more in §4.2 below.

[^2]: 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.

[^3]: 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.

[^4]: 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).

[^5]: on https://github.com/bstarynk/bismon/blob/master/login_ONIONBM.xhtml

[^6]: The entire analyzed data should fit in *Bismon* persistent store, so dozens of gigabytes, not terabytes.
1.6.2 About Bismon as a domain-specific language

Notice that Bismon is not even thought as a textual domain specific language (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. 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; 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 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 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 \( \alpha \) and the second component representing the right hand-side \( \beta \). 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 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.

---

81 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.
82 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].
83 In mid-2018, that Web interface was incomplete, and I still had to temporarily use some obsolete GTK-based interface that even I find so disgusting that I won’t explain it, and sometimes even to edit manually some store2.bmon data file, cf § 2.3.1.
84 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.
85 Technically the routine would be in the module’s shared object binary; the function is a Bismon object reifying the code of that
• 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)

• 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 89.

• 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 90 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

89 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.

90 See http://git-scm.com/
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_jstmt 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 91.

![GTK Interface Screenshot](image)

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

This crude GTK3 interface 92 is buggy and not satisfactory. It will and needs to be replaced by a Web interface. Several lessons have been gained with this experience:

- using GTK 93 is in practice incompatible with a multi-threaded precise garbage collector 94, like the one

---

91 Ian L. Taylor’s libbacktrace is used in bismon to provide symbolic backtraces.

92 It is implemented in 12.5KLOC of C code in gui_GTKBM.c, newgui_GTKBM.c and guicode_BM.c. We won’t debug that code - which crashes often - but will remove it once it can be replaced by a web interface.

93 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.

94 See also https://stackoverflow.com/q/43141659/841108 about GTK and Boehm’s conservative GC.

---
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 pioneered 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). At end of 2018 (or first quarter of 2019) it will be entirely scrapped, and the static analysis expert (and
other users of Bismon) will interact with bismon thru some Web interface.

Work on the future Web interface has significantly progressed 95. New users -called contributors - can be
voluntarily registered and unregistered on the command line 96 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 97 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 98 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 99 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 100 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 101 is less important), so transforming to A-normal form is not required at translation (to C++) time.

95With the hand-written web_ONIONBM.c using libbonion, and the bismon module webjs_module translated into the
modules/modbm_1zCsXG40TPr_8PwkDAWrl6S.c emitted C file of more than 6.5KLOC in october 2018.
96Use the option -contributor= to add them, -remove-contributor= to remove them and -add-passwords= to set
their encrypted bismon-specific password.
97See http://codemirror.net/ for more.
98For 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.
99Notice that GCC compute these call frame sizes (see the -fstack-usage option), and can detect excessively big call frame
with -Wstack-usage= option.
100Of course dynamic calls, e.g. call thru function pointers, make that much more complex and will require manual annotation.
101In practice, generated GCC plugins would simply “digest” some internal GCC representations and transmit their outcome to the
1.7 Multi-threaded and distributed aspects of Bismon

The Bismon monitor is by itself a multi-threaded process. It uses a fixed thread pool of worker threads (often active) 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 modules -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).

The worker threads of Bismon are implementing its agenda 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 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. 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.
- 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. ccl started from gcc, or cclplus 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

---

102 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.
103 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.
104 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.
105 Actually tasklets are objects (see §2.1.2 page 28 below), and to run them, the agenda is sending them a message with the predefined selector run_tasklet.
106 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.
107 So using your browser’s backward and forward navigation arrows won’t work well because in single-page applications they cannot work reliably
Specialized Static Analysis tools for more secure and safer IoT software development [draft] (Bismon)

CHARIOT D1.3 v2-

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 \(^\text{108}\) 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 by sometimes generating a GCC plugin \(^\text{109}\) to inspect the public headers of GCC\(^\text{110}\). Even when the end-user developer is targeting 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 \(^\text{111}\) generated C++ files) *.shar archives (obtained by Web requests, or perhaps some wget or curl command in some Makefile) for GNU sharutils \(^\text{112}\) 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 \(^\text{113}\) 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.

---

\(^\text{108}\)It is tempting to call such plugins cross-plugins, since they would be dlopen-ed by a cross-compiler.

\(^\text{109}\)It is tempting to call such plugins straight-plugins, since they would be dlopen-ed by a straight compiler, not a cross-compiler.

\(^\text{110}\)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.

\(^\text{111}\)By past experience in GCC MELT, we did generate C++ files totaling almost a million lines of C++ code, and compiling such a large generated C++ code base took dozens of minutes, and created a bottleneck.

\(^\text{112}\)See https://www.gnu.org/software/sharutils/ for more.

\(^\text{113}\)See https://ocsigen.org/
2 Data and its persistence in Bismon

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 114 (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 115 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 116 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.

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 117 numbers; but NaN is never boxed and becomes reified as the nan_double object; hence boxed doubles stay comparable (with the convention 118 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.

- **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.

- **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.

The nil value is generally not considered as a value, but as the absence of some value. We might (later)

---

114To 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.

115Their hash is a non-zero 32 bits unsigned number. Only the nil pointer has an hash of 0.

116For an example, look into the ongoing DECODER H2020 project, ICT-16 call, grant agreement number 82423.

117The 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.

118The intuition behind such a convention would be that \(-0.0\) is a very (or infinitely) small negative number, so less that +0.0 which
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 119 (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 120 or _0xbmmxnN8E8_02uEeqMqMNH. 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 the € objid abbreviation 121, 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 122. So locking (or unlocking) an object really means using that mutex to lock the object.

119Each object has its mutex initialized with pthread_mutex_init(3p) with the PTHREAD_MUTEX_RECURSIVE attribute, and lockable with pthread_mutex_lock(3p) etc...

120That 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 0x31a5cb0767997fd31e5183916681468.

121This € objid abbreviation is for documentation purposes; it is not acceptable in persistent store files.

122We 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.
lock on pthread mutex operations.  

- 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** pointer to an object (usually another one) describing its class, as understood by Bismon. It is allowed to change dynamically 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**.
  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 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 `objcrtout_sigBM` corresponding to the predefined object `function_sig`. For an object of `oid` that signature its routine address corresponds to the C name `crout:oid:BM`. For instance, to initialize (at load time) the object of `oid _9CG8SKN6Q1_4PiHd8cnyd` the initial loader (or the module loader) would `dlsym` the `crout_09Hug4WGnPK_7PpZby8pz84_BM` C function name.

- objects may also have some (nearly arbitrary) **payload** - which can contain anything that don’t fit elsewhere. That payload 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**.

---

123 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**

124 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.

125 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.

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

127 The generated C code of modules also contains an array of constant objids, and another array of routine objids.

128 Some weird payloads, in particular for web exchanges and web sessions, cannot be created programmatically by public functions 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.
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

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, 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, and no tasklet is active.

The garbage collection roots include:

- all the tasklets queued in the (several queues) of the agenda
- all the predefined objects
- all the constants (objects 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.

---

Footnotes:

129 That is, json_t* from the Jansson library. It is practically useful for WebSocket messages.

130 See also previous footnote 71 on page 19 for possible improvements of the GC.

131 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.

132 The object of objid _1FEnnpEkGdI_5DAcVDL5XHG should be designed as BMK_1FEnnpEkGdI_5DAcVDL5XHG in hand-
The naive *Bismon* monitor garbage collection\(^{133}\) 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. 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. 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\(^{134}\).

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\(^{135}\) or MySQL\(^{136}\), 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\(^{137}\) 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 algorithm used to dump a persistent state, and classical copying 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 should be backed-up and probably version-controlled\(^{138}\) at every site using the *Bismon* monitor.

Notice that in *Bismon* only the heap is persisted, but “continuations” or “call stacks” or threads\(^{139}\) are *not persisted*\(^{140}\) by themselves.

---

\(^{133}\)Some previous experimentation with Boehm’s GC in multithreaded settings has been unsatisfactory.

\(^{134}\)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.

\(^{135}\)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.

\(^{136}\)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.

\(^{137}\)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.

\(^{138}\)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).
### 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**¹⁴¹ 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 files 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 bootstrapped¹⁴², so all its files should be generated (including what is still in 2018 the hand-coded “runtime” part of Bismon such as our `_*_.c` files, notably the load and dump machinery in `load_BM.c` and `dump_BM.c`, the agenda mechanism in `agenda_BM.c`, miscellaneous routines including the support of module loading in `misc_BM.cc`, etc...). However, we are still quite far from that ideal. Existing bootstrapped systems¹⁴³ such as CAIA (see Pitrat [2000, 2009a,b], Pitrat [2013-2019]¹⁴⁴, OCAML (cf Leroy et al. [2018]; Leroy [2000] and many other papers by Xavier Leroy and the Gallium team at INRIA), CHICKEN/SCHEx¹⁴⁵, SELF (cf Ungar and Smith [1987]), SBCL¹⁴⁶ 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 refactoring¹⁴⁷. 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).

### 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.

The figure 4 shows an example of the textual dump for some object (named `first_test_module`) of objid `_9oXtCgAbkqv_4ylxhhF5Nhz` extracted from the data file `store2.bmon`.

The lines starting with « or !( and with » or !) are delimiting the object’s persistent representation. Comments¹⁴⁸ can start with a bar | till the following bar, or with two slashes // till the end of line. 1 ~ with

---

¹⁴¹Conceptually, the `dlopen-ed` shared object files, such as our `modubin/modbm_1zCaXG40TPR_r8PwKDAx165.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 `dlopen` them at initial load time.

¹⁴²This 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.

¹⁴³Observe 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/](http://www.linuxfromscratch.org/) for guidance.

¹⁴⁴That [blog](http://starynkevitch.net/Basile/guile-tutorial-1.html) explains that all the 500KLOC of the C code of CAIA and its persistent data are generated.

¹⁴⁵Notice that semantically Bismon is quite close to Scheme and shares many features with that language; see [http://starynkevitch.net/Basile/guile-tutorial-1.html](http://starynkevitch.net/Basile/guile-tutorial-1.html) for more. The bootstrapped CHICKEN system is on [https://www.call-cc.org/](https://www.call-cc.org/) and is, like Bismon, translated to C.


¹⁴⁷In 2018, if we decide painfully to change the representation of attribute associations in objects, we have to modify a lot of handwritten 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.

¹⁴⁸Once the persistence code - loading and dumping of the state - is mature enough, we will disable generation of comments in data files.
Specialized Static Analysis tools for more secure and safer IoT software development [draft] (Bismon)

```
«9oXtCgAbkvq_4y1xhhF5Nhz |=first_test_module |
± name (~ first_test_module ~)
± 1546368464.488
∈ _5bP4nozCtp0_100PPTK398ml |=basiclo_dumpable_module |
→ _3dmecF21dtxI_1beIF14qge3 |=also|
% _5DDSY1YwVzr_6dOU4tiBldk { | _6SSe2uyt8Cn_4QNhKM5hsDA _9O2lgu1TweO_0mVlpTwrBG1 }}
→ _4rZ14weGz_8Y035uB1eKL |=see|
| _3dmcFZldtxI_1bEIFl4jqe3 |=also|
% _5DDSY1YwVzr_6dOU4tiBldk ( { _6SSe2uyt8Cn_4QNhKM5hsDA _9O2lgu1TweO_0mVlpTwrBG1 } )
Ω _4rz1Bi4weGz_8Y035uB1eKL |=see|
{ _43Y25VLmh6s_3JRpErevcR4 _6SSe2uyt8Cn_4QNhKM5hsDA _9O2lgu1TweO_0mVlpTwrBG1 }
!^ 3
\_43Y25VLmh6s_3JRpErevcR4
\_9O2lgu1TweO_0mVlpTwrBG1
\_0quxQEfintmp_1Wp2YuKhnJ3
// emitted persistent module modbm_9oXtCgAbkvq_4y1xhhF5Nhz.c
»_9oXtCgAbkvq_4y1xhhF5Nhz
```

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

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 ! 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_object 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-objs are prefixed with !^ or with μ. Then the collection of objects (similar to figure 4 each) follows.

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 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 to a scanned object having some payload during the scanning phase. The dump_value

\footnote{That temporary_attribute_object class has objid _23vPTNrGyB.}
\footnote{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.}
\footnote{The dumper object is the additional argument to dump_scan.}
message is sent\textsuperscript{152} is to dumped values during the emission phase. The \texttt{dump_data} message is sent\textsuperscript{153} 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 \texttt{temporary_attribute_object} are not dumped.

\textsuperscript{152}Additio\textsuperscript{n}nal arguments to to \texttt{dump_value} are a string buffer object, the \texttt{dumper} object, and the depth as a tagged integer. Notice that dumping of integers is implemented with the \texttt{dump_value} method of class \texttt{int}, dumping of tuples is implemented with the \texttt{dump_value} method of class \texttt{tuple}, dumping of object references as \texttt{objids} is implemented with the \texttt{dump_value} method of class \texttt{object}, etc....

\textsuperscript{153}Extr\textsuperscript{a} arguments to \texttt{dump_value} are: the \texttt{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).

The rest of this chapter will be written in the final D1.3v2 version.

A significant part of this chapter should be generated (like GCC MELT generated its documentation, see Starynkevitch [2008-2016]) from the persistent state of Bismon. Perhaps this chapter should be put after the “using Bismon” chapter (§4).

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 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 of GCC API is practically relevant (most of the API of gcc–8.3 should stay in the next gcc–9.0 version).

Describe 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.

Additional content of this §3.1 will be written for the final D1.3v2.

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 doesn’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 developed. 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 which is, in 2019, understood by most free software source code editors running on Linux, including emacs, or vim, or VSCode. It might even later use the new Sarif protocol, designed for communication between static source code analyser.

We probably would also take as an example the analysis of some MQTT library. The insight is to trust some existing

---

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

155 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.

156 See https://langserver.org/ for more.

157 See http://docs.oasis-open.org/sarif/sarif/v2.0/csprd01/sarif-v2.0-csprd01.html for more.
MQTT implementation\(^{158}\), 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, 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\(^{159}\) are available on Linux. In principle, such an approach might be used in \texttt{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\(^{160}\) 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.

\[\text{Additional content of this §3.2 will be written for the final D1.3}^{162}\].

\(^{158}\)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.

\(^{159}\)See TENSORFLOW on \url{https://www.tensorflow.org/}, and GUDHI on \url{http://gudhi.gforge.inria.fr/}, and many other similar libraries.

\(^{160}\)Technical Readiness Level and the TRLwikipage for more.
4 Using Bismon

This section §4 should become somehow a user manual, and will be written for the final D1.3². 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 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 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.

![Table 11: canonical JSON encoding of a Bismon value v.](http://json.org/)

The canonical JSON encoding is implemented as the canonjsonifyvalue_BM function.

---

161 See [http://json.org/](http://json.org/) for more
162 Adding immutable JSON values as a new kind of Bismon value could be considered in the future.
163 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_1USbaS25hx1"), or abbreviated objids (e.g. JSON "ε_9Z2BqJbf4"), or named objects (e.g. JSON "arguments"), related to Bismon object arguments, i.e. ε_0jFqaPPHg are really decoded.

Table 12: nodal JSON decoding \( \langle js \rangle_{\text{nod}} \) of a JSON value \( js \).

4.1.3 JSON extraction with extract_json

A Bismon statement of class basiclo_extractjsonstmt, obtained with the \texttt{\textasciitilde extract_json \{ objson extrac-tornode sub-statements... \}} readmacro, so having the attributes \texttt{json_object : objson} and \texttt{extract_json : extrac}tornode, can extract data from the JSON in a payload (of some object \texttt{objson}, usually of class \texttt{json_object}).
Specialized Static Analysis tools for more secure and safer IoT software development [draft]  

** extractor node | success condition | side effects and explanation **
--- | --- | ---
| `json_null ()` | `js ≡ null` | succeeds when `js` is the null JSON |
| `json_false ()` | `js ≡ false` | succeeds when `js` is the false JSON |
| `json_true ()` | `js ≡ true` | succeeds when `js` is the true JSON |
| `int (v^{int})` | `js ≡ some JSON integer i` | succeeds when `js` is a JSON integer `i`, and assign to integer variable `v` its integral value `v ← i` |
| `double_float (v^{real})` | `js ≡ some JSON real δ` | succeeds when `js` is a JSON real `δ`, and assign to double variable `v` its boxed double value `v ← boxed-double(δ)` |
| `string (v^{real})` | `js ≡ some JSON string σ` | succeeds when `js` is a JSON string `σ`, and assign to value variable `v` its boxed string value `v ← boxed-string(σ)` |
| `id (v^{obj})` | `js ≡ some JSON string σ` | 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 ← ω` |
| `name (v^{obj})` | `js ≡ some JSON string σ` | 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 ← ω` |
| `member (x^{val} v^{obj})` | `js ≡ some JSON string σ` | 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 ← ω` |
| `put (ω)` | always succeeds | then put in that given Bismon object `ω` a JSON payload with `js` inside |

The `χ, χ', χ...` are extractors; the `js, js', js...` are JSON values; `χ ∋ js` means that the extraction using extractor `χ` on JSON `js` was successful; the `v, v', v'', v, w,...` are Bismon local variables whose type is explained with blue annotations like 166; 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.)

That extraction is driven by the given extractor node 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 thing165 like `{ "do": "foo", "obj": "_3d9rq9TD6PH_6qUv4Hao767", "targets": [ "_3dmcFZldtxI_1bEIP143qei", "_6vMEDC0qxdp_7shiuw0EZEqx" ] }` could be extracted with the composite extractor `+json_object("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 { `_3dmcFZldtxI_1bEIP143qei` } that is to `{also and}`.

### 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 166 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,

164Our JSON extraction is inspired by some pattern matching constructs on linear patterns with semi-unification.

165We may call JSON things what the JSON standard call JSON values; and that example could be inside some AJAX or REST POST HTTP request.

166Such as Firefox 60.7 or later, or Google Chrome 75.0 or later, both exist in 2019Q2.
extractor node

* set (v<sup>val</sup>)

success condition

\[
js \equiv \text{some JSON array} \\
[ j_{s_1}, \ldots, j_{s_m} ] \\
\text{of strings, so} \\
\begin{align*}
  j_{s_1} &= \text{string } \sigma_1 \\
  \ldots &\quad \land \\
  j_{s_n} &= \text{string } \sigma_n \\
\end{align*} \\
\begin{align*}
  \forall i, \exists \text{ object } \omega_i, \\
  ( \text{objname}(\omega_i) = \sigma_i \\
  \lor \text{objid}(\omega_i) = \sigma_i )
\end{align*}
\]

side effects and explanation

succeeds when \( js \) is some JSON array of JSON strings \( j_{s_i} = \text{string } \sigma_i \), then builds a Bismon set of \( \text{existing objects } v_{\text{set}} = \{ \ldots, \omega_i, \ldots \} \) from these, with \( \sigma_i \rightarrow \omega_i \), such that \( \text{objname}(\omega_i) = \sigma_i \) or \( \text{objid}(\omega_i) = \sigma_i \).

* json_array (\( \chi_1 \ldots \chi_n \))

\[
js \equiv \text{some JSON array} \\
[ j_{s_1}, \ldots, j_{s_n} ] \\
\land \chi_1 \triangleright j_{s_1} \land \ldots \land \chi_n \triangleright j_{s_n}
\]

succeeds when \( js \) is a JSON array of \textit{exactly} \( n \) elements : \( j_{s_1}, \ldots, j_{s_n} \), and extracts them in sequence, using \( \chi_1 \) on \( j_{s_1} \), then \( \chi_2 \) on \( j_{s_2} \), etc... and at last \( \chi_n \) on \( j_{s_n} \).

* json_entry (\( \alpha \chi \))

\[
js \equiv \text{some JSON object having} \\
\exists i, \{ \ldots \alpha : j_{s_i}, \ldots \} \land \chi \triangleright j_{s_i}
\]

succeeds when \( js \) is a JSON object having an attribute string \( \alpha \) associated to some JSON value \( j_{s_i} \), which can be extracted using \( \chi \).

* json_entry_object (\( \omega \chi \))

\[
js \equiv \text{some JSON object having} \\
\exists i, \exists \text{ some JSON string } \alpha, \\
( \text{objid}(\omega) = \alpha \lor \text{objname}(\omega) = \alpha ) \\
\exists i, \{ \ldots \alpha : j_{s_i}, \ldots \} \land \chi \triangleright j_{s_i}
\]

succeeds when \( js \) is a JSON object having an attribute string \( \alpha \) which is the name or objid of the given Bismon object \( \omega \), associated to some JSON value \( j_{s_i} \) which can be extracted using \( \chi \).

* json_object (\( \chi_1 \ldots \chi_n \))

\[
js \equiv \text{some JSON object of length} \\
\text{exactly } n \text{ such that} \\
\chi_1 \triangleright js \land \ldots \land \chi_n \triangleright js
\]

succeeds when \( js \) is a JSON object of \textit{exactly} \( n \) members : \( \{ \alpha_1 : j_{s_1}, \ldots, \alpha_n : j_{s_n} \} \), and that same object \( js \) can be extracted by each of the \( \chi_1 \ldots \chi_n \) in sequence; the sub-extractors \( \chi_i \) are often but not always of form * json_entry (\( \alpha_i \chi_i \)), ...

* json_value (v<sup>val</sup>)

always succeeds

assign \( v \leftarrow \langle j_{s} \rangle_{\text{nod}} \) to value variable \( v \) the \textit{nodal-encoding} of \( js \)

* value (v<sup>val</sup>)

\[
\exists x \text{ Bismon value, } j_{s} \equiv [x]_{\text{json}}
\]

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 (\( \epsilon \text{ stmt}_1 \ldots \text{ stmt}_n \))

The testing expression \( \epsilon \) evaluates as true (non-nil value, or non-zero integer)

succeeds when expression \( \epsilon \) is true (it usually involves some variables computed by previous extractors), then executes in sequence the statements \( \text{stmt}_1 \ldots \text{stmt}_n \) for their side effects

* and_then (\( \chi_1 \ldots \chi_n \))

sequentially and lazily

\[
( \chi_1 \triangleright js \land \\
\ldots \\
\land \chi_n \triangleright js )
\]

succeeds when the same \( js \) can be successfully extracted by \textit{each} of the \( \chi_1 \ldots \chi_n \) in sequence from left to right (failing lazily as soon as some \( \chi_j \) fails), and combine their side effects (visible from \( \chi_j \) to the next \( \chi_{j+1} \)).

* or_else (\( \chi_1 \ldots \chi_n \))

sequentially and lazily

\[
( \chi_1 \triangleright js \lor \\
\ldots \\
\lor \chi_n \triangleright js )
\]

succeeds as soon as the same \( js \) can be successfully extracted -from left to right- by some of the \( \chi_i \) (continuing lazily to the next \( \chi_{j+1} \) when it fails), and combine their side effects (visible from \( \chi_j \) to the next \( \chi_{j+1} \)).

Notation:

Share notation with table 13 above. In addition, the \( \epsilon \) are Bismon expressions. The \( \alpha \) are JSON strings in attribute position. The \( \sigma \) are strings (usually JSON strings). The \( \text{stmt} \) are Bismon statements.

Table 14: complex extraction from some JSON thing \( js \); (see also table 13 above.)
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. Compatibility with other browsers 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 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 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 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 web exchange payload contains not only a string buffer, 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 sessionexchange_object. 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 associated to key "bismontab" identifies and gives the tab number in the browser. An inactive web session expires in about an hour.

Of course, web request objects or web session objects are transient and are not and should not be persisted at dump

---

167So 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 peculiarities or deficiencies. Likewise, scalability to thousands of simultaneous HTTP connections is out of scope in bismon, but it is essential in most web applications.


169For 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!

170In particular, any HTTP request containing .. is rejected.

171A 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.

172There is no programmatic way to create such a web exchange payload. It can only be created by processing such dynamic HTTP requests.

173Since 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.

174So the only way to create a web session payload is thru the login form. There is no programmatic way to create it.


176See the USER_WEBSESSION_EXPIRATION_DELAY constant in web_ONIONBM.c
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 177 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_root178 dictionary 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 179 used. If finally no web handler closure is found, an 404 Not found status is returned. The the_web_sessions predefined object stores the dictionary of transient web session objects and associates a cookie string to its web session object. That dictionary 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 dictionary 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 180 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 181. That emit_xhtml, which get as arguments: the value 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 182 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 &lt; 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 emit_xhtml_open routine 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 emit_xhtml_open returning the tag string like before (e.g. "span") for a <span class='somecl'></span> emission, then > is output to end the opening tag, then the components, then the end tag is emitted, using the returned

177 See the WEBEXCHANGE_DELAY_BM constant in file web_ONIONBM.c ...
178 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 dictionary Whdict is associated to some other object ωwh, of class webhandler_dict_object, that dictionary object ωwh is recursively explored with the rest of that URL path (e.g. "status" in our example). If Whdict has some and_then attribute associated to an object wwhthen which is a web dictionary object, that wwhthen is explored with an incremented depth.
179 Since dictionary objects map non-empty strings to non-nil values (cf. §2.1.2).
180 For instance, within a <script> HTML5 element containing JavaScript, it is not even allowed in HTML5 to have if (x &lt; 5) even if ordinary HTML rules suggest to use &lt; instead of < in textual content... That makes compositional generation of mixture of HTML and JavaScript emitting HTML much harder.
181 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).
182 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.
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 $\omega_{conn}$, it is emitted per the following rules:

- if $\omega_{conn}$ is one of int, hexa, octa and v_html is an unary node with an integer son $n$, that integer $n$ is emitted in decimal, hexadecimal, octal respectively.
- if $\omega_{conn}$ is id and v_html is an unary node with an object son $\omega_{son}$ its objid is emitted.
- if $\omega_{conn}$ is buffer and v_html is an unary node with an object son $\omega_{son}$ which has a string buffer payload, that string is emitted HTML-encoded.
- if $\omega_{conn}$ is object and v_html is an unary node with an object son $\omega_{son}$ its name or objid is emitted.
- if $\omega_{conn}$ is name and v_html is an unary node with an object son $\omega_{son}$ its name is emitted, and when $\omega_{son}$ is not a named object, we have a failure. If $\omega_{conn}$ is name and v_html is an binary node with an object first son $\omega_{son}$, and some arbitrary non-nil second son $\epsilon$ its name is emitted, and when $\omega_{son}$ is not a named object, the $\epsilon$ is recursively emitted
- if $\omega_{conn}$ is sequence, every son of v_html is emitted in sequence, with an incremented depth. Nothing is additionally emitted between them.
- if $\omega_{conn}$ 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 $\omega_{conn}$ as connective, we extract its emit_xhtml_node attribute $v_{emit}$ node and its emit_xhtml_connection 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 $\omega_{conn}$ 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 $\omega_{conn}$ 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. 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 in Bismon by its homoioticity, and used by the static analysis expert, won’t handle lines of tokens in an editor widget, 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.

---

183 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.

184 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.

185 See also end of 1.6.1 above, and also https://stackoverflow.com/a/47116008/841108 for more.

186 Of course, using https://codemirror.net/ will be useful to show analysed C or C++ source code, but not for our Bismon’s
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.
5 Miscellaneous 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 unsatisfactory. 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\(^\text{187}\) (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\(^\text{188}\) reifiable into transient objects, having as payload a linked-list sequence of call frames.

The final D1.3\(^\text{v2}\) 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\(^\text{v2}\) document will contain a longer conclusion.

\(^{187}\) So there are no released stable versions of this software, but snapshots.

\(^{188}\) 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.
References


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.


Specialized Static Analysis tools for more secure and safer IoT software development [draft]  

Bismon

CHARIOT D1.3 v2-

F. P. Brooks, Jr. No silver bullet - essence and accidents of software engineering.  
ISSN 0018-9162.  
URL https://doi.org/10.1109/MC.1987.1663532.

F. P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*.  


F. Chazal and B. Michel. An introduction to Topological Data Analysis: fundamental and practical aspects for data scientists.  
URL https://ui.adsabs.harvard.edu/abs/2017arXiv171004019C.

URL http://doi.acm.org/10.1145/2029932.2029934.

C. J. Cheney. A nonrecursive list compacting algorithm.  
doi: 10.1145/362790.362798.  
URL http://doi.acm.org/10.1145/362790.362798.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints.  

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.  
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.  
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.


URL https://doi.org/10.1007/978-3-642-14681-7_1.


URL http://doi.acm.org/10.1145/2480362.2480593.

URL http://www.dicosmo.org/HoldUp/.  
[French book, out of print, but downloadable].

URL https://hal.inria.fr/inria-00076535.

Free Software Foundation. GCC runtime library exception, 2009.  

Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler.  
*Higher Order Symbol.*
Specialized Static Analysis tools for more secure and safer IoT software development [draft] (Bismon)

Chariot D1.3 v2


Specialized Static Analysis tools for more secure and safer
IoT software development [draft]  

Bismon  

C. Hariot  

D1.3 v2-  


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** on your Linux computer, then run **make doc** 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](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/](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 189 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.

This entire document is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

---

189 See Debian documentation project on https://www.debian.org/doc/ddd etc...