# Hardware enforced fine grained data labeling on RISC-V using RUST

Master of Science Thesis

ing. X. van Rijnsoever

TU Delft Delft University of Technology

Challenge the future

# Hardware enforced fine grained data labeling on RISC-V using RUST

by

ing. X. van Rijnsoever

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday September 18, 2020 at 2:00 PM.


| | | |
|---|---|---|
| Student number: | 1006908 | |
| Thesis committee: | Dr. ir. J.S.S.M. Wong, | TU Delft, supervisor |
| | Dr. J. Hoekstra, | TU Delft |
| | Dr. ir. A.J. van Genderen, | TU Delft |
| | Ir. S. Woutersen, | Technolution, daily supervisor |




TU Delft

# Abstract

Software bugs in many different variants can potentially leak sensitive data to an attacker. Implementing a separation mechanism for security domains can prevent incorrect or malicious code to leak sensitive data from one security domain to another. This work presents a separation mechanism based on labeling security domains with a label in tagged memory, at word-level granularity, called *color labeling*.

Utilizing a tagged architecture based on the RISC-V architecture, color labeling assigns colors (denoting a security domain) to individual memory words, cache lines, registers and peripherals. Using a simple set of hardware enforced policies, data protection is ensured. Control flow integrity is maintained with the help of additional tag bits that denote code and valid jump addresses. New instructions have been added for functions that handle data residing in multiple security domains.

Software support is implemented in the Rust compiler. The compiler is enhanced with macros to support the coloring concept via source level annotations. Incorrect use of labels is reported during compilation. An external tool is used to generate tag information and generate a security report with information on variable coloring and special function use and construction. Using the external tool keeps the changes to the compiler minimal, thereby reducing the maintenance burden and the required trust in the compiler as well. The report can be used in a security audit.

The concept is implemented on an instruction set architecture simulator. The toolchain modifications and the concept itself have been tested on this simulator. Testing showed the concept can prevent cross-security domain information leaks under several common attack patterns. The overhead due to the execution of additional instructions in the executable code depends on the actual code. Tests with the typical target application OpenVPN-NL showed a less than 5% increase in instruction count for the most commonly called functions.

By designing or redesigning software specifically for color labeling, this overhead can possibly be further reduced. Further testing, specifically on an actual hardware implementation is recommended.

Due to timing constraints, the concept has not been implemented in hardware. However, the hardware performance costs are estimated to be negligible. The area requirements are substantial: implementing the concept in the RISC-V softcore requires double the external memory capacity and FPGA resource utilization is estimated to require 14% more ALMs and 74% more internal memory blocks.

**Keywords:** coloring, labeling, tagged memory, security domain separation, hardware enforced, Rust, compiler, toolchain, RISC-V, ISA simulator, security

# Contents

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

*"If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology"*

*– Bruce Schneier*

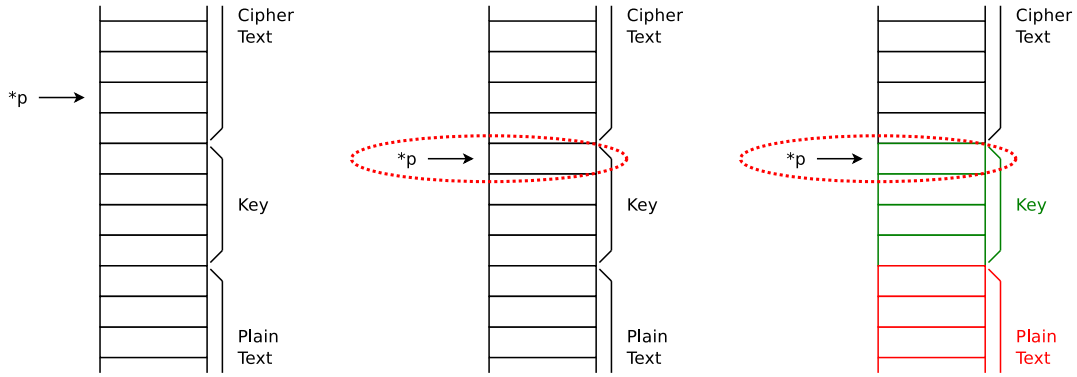
On April 7, 2014 the Heartbleed bug in the widely used OpenSSL implementation of the Transport Layer Security (TLS) protocol was publically disclosed [1], [2]. Heartbleed could disclose decrypted and potentially sensitive data, including private keys. About 500,000 SSL certificates are estimated to have been affected [3], including those of Amazon Web Services, GitHub and Wikipedia. Subsequent abuse of the bug led to, amongst others, theft of Social Insurance Numbers from the Canada Revenue Agency [4] and the compromised confidentiality of 4.5 million patient records in the United States [5]. The costs of the bandwidth required for distributing the certificate revocation records alone is estimated to be over $400.000 [6].

As this example shows, in our highly connected and ICT dependent world, problems like Heartbleed can have a very high impact and associated price tag. Not only accidental bugs, but also malicious attacks, both by nation states or private parties are increasingly threatening our ICT infrastructure, privacy, and security. Identity and information theft, monetary fraud, and espionage are becoming more common and the impact of security breaches is ever increasing. Many such attacks abuse common software bugs like Heartbleed. For this reason software has a bad reputation in the field of security, despite many years of software security research.

## 1.1 Buffer Over-read

The underlying problem of the Heartbleed bug turned out to be a variant of a buffer over-read caused by improper input validation [7] (see also Appendix B). In a buffer over-read, the boundaries of a buffer are not respected and more data is read out than should be allowed. In order to better illustrate this problem, let's start with a very simple, but relevant, example. In a software application that handles encryption and decryption, at least the following information is, at some point, kept in memory:

**Cipher text** is the encrypted representation of the data;




**(a)** Initial layout **(b)** Buffer over-read **(c)** With color labels

**Figure 1.1:** Buffer over-read crossing security domains

**Plain text** is the unencrypted, readable representation of the data;

**Key** is the secret key data that is required to encrypt or decrypt the data.

These different data elements may be present in the memory according to the memory layout shown in Figure 1.1a. The pointer `p` is used here to read out the cipher text data and send it over an unsecure network. If, due to a bug or malicious intent, the pointer `p` points beyond the boundery of the cipher text buffer (see Figure 1.1b), then data that logically belongs to another security domain, in this example the key data, could be send out over the unsecure network. In the running program there is no notion of the security domain of the data currently pointed to, so this error cannot be detected.

These types of bugs are not uncommon: searching the Common Vulnerabilities and Exposures (CVE) list [8] for terms related to buffer overflows, over-reads, etc, reveils that these errors cause over 13% of the reported vulnerabilities in the period 1999–2018, and over 16.5% in 2018 alone (see Appendix A). The actual number is probably even higher, since the CVE database only contains publically available reports. For many closed source applications the exact cause and remedy are not published.

For a system with hardware peripherals, it is also possible that the memory read-out happens via e.g., a misconfigured DMA controller, which makes it difficult to mitigate this problem with a software-only solution.

In order to prevent data of one security domain to be accessible from another security domain, the data can be tagged with a color label. This is depicted in Figure 1.1c, here the cipher text data is shown in black, key data is shown in green, and plain text data in red. Now the pointer `p` belongs to the security domain denoted with the color black. If this black pointer is then used to read from the green key data, this can be detected and prevented, even in the presence of a hardware peripheral reading out the memory.

## 1.2 Technolution

Technolution [9] is a high-tech electrical and computer engineering company specialized in integrating state-of-the-art technology in the areas of infrastructure, energy, industry, and public safety and security. One of the products in the security branch is the PrimeLink, see Figure 1.2. The PrimeLink can setup Virtual Private Network (VPN) connections between private networks over the (unsecure) internet. The


**Figure 1.2:** PrimeLink box

different networks are conventionally denoted with different colors: red for the secure (unencrypted) private network, black for the unsecure (encrypted) public network (e.g., the internet), as is indicated by the coloring of the network connectors on the PrimeLink box.

The PrimeLink combines dedicated hardware with software running on a RISC-V softcore. High assurance applications are preferably implemented in hardware, since exploiting the problems that make software unsecure, such as buffer overflows, buffer over-reads, code injection, etc, is much harder in hardware. However, budgetary constraints as well as time-to-market and maintainability concerns, might necessitate the use of software.

## 1.3 Research Question

In order to prevent software bugs to escalate into security breaches, a domain separation mechanism is required. Several approaches can be used in practice, such as separate CPUs for each security domain or compartmentalization of application components in a special framework such as Genode [10]. An alternative approach would be to annotate data with a color label that indicates the security domain. Similar to the colored network interfaces, encrypted data could be labeled black and unencrypted data could be labeled red. The programmer should then annotate the program with such labels, but could otherwise use standard programming practices and paradigms.

Policies should be in place to access this labeled data. The compiler should issue warnings when these policies are not adhered to. In order to increase the trust in the software, these policies should be enforced in hardware, so that in case of failure or a malicious attack, the domain separation is still kept intact.

Technolution has designed a RISC-V softcore [11] for use in its high assurance applications, such as the PrimeLink. The RISC-V architecture [12] is designed to be easily extensible. The color labeling has to be implemented on this platform in order to improve the safety of software solutions in high assurance products.

In order to further improve the safety of the software, the safe language Rust [13] is used as software

platform. The Rust programming language has a number of strong memory safety guarantees while still being able to create very fast executing native code. The RISC-V architecture is an officially supported platform of the language. The de-facto compiler is open source and the language can be extended in a number of different ways in order to support new features.

The research question of this work is thus:

*How can data labeling be used to implement a hardware-enforced security domain separation on a RISC-V processor with support in the Rust Programming Language?*

The main goal of this work is:

- preventing cross-security domain information leaks.

With this work:

- The programmer should be able to annotate variables with a color, indicating the security domain;
- The programmer should be able to annotate functions handling data from multiple security domains;
- The toolchain should be able to process these annotations and generate compile time errors on incorrect usage;
- The toolchain should be able to generate label information and special functionality for handling multiple security domains;
- The hardware should be able to enforce the policies in order to preserve the domain separation even under software failures;

## 1.4 Methodology

In order to answer the research question, the work is split into several smaller parts:

- Investigate attack patterns and existing solutions in a literature study;
- Investigate existing solutions for labeling data in a literature study;
- Determine requirements for the labeling architecture via the results of the literature studies and a representative test program.
- Design the labeling architecture based on these requirements;
- Implement support for the labeling architecture in the software toolchain:
  - Design a programmer's interface;
  - Propagate the annotations into the executable;
  - Extract label data and security information from the executable via an external tool;
- Implement support for the labeling architecture in the RISC-V ISA simulator;

- Implement support for the labeling architecture in the RISC-V softcore.

Due to timing constraints the concept is not implemented in hardware, although some considerations for a possible hardware implementation are given.

## 1.5 Thesis Outline

Background information on common attack patterns and existing defences, tagged architecture and a simple but representitive test program can be found in Chapters 2, 3, and 4, respectively.

The design of the color labeling concept is then explained in detail in Chapter 5. The implementation of this concept is spread over the next four chapters: Chapter 6 provides a broad overview of the complete implementation and the interdependencies. The next chapters zoom in on one aspect of the implemention:

- Chapter 7 describes in more detail the implementation of the concept in the Rust toolchain;
- Chapter 8 describes the label-tool, an external program used to extract color information from the executable and generate a security report;
- Chapter 9 describes the modifications to the Fremu simulator that has been used to test the concept and the software implementation;
- Chapter 10 describes the proposed changes to the hardware implementation of the concept on the Frenox RISC-V core.

Chapter 11 describes the different test setups for testing the concept and implementation functionally and deriving the performance overhead. Concluding remarks and suggestions for future research can be found in Chapter 12.

# Chapter 2

# Attack Patterns and Existing Defence Mechanisms

*"You can't defend. You can't prevent. The only thing you can do is detect and respond."*

*– Bruce Schneier*

## 2.1 Introduction

Attacks on software applications (running on top of hardware) can occur in many different ways. Extensive research has already demonstrated that these attacks can be categorized using so-called attack patterns. In this chapter, we will discuss these attack patterns and common defenses against them before highlighting our own solution called "color labeling".

First, the assumptions on the hardware and software are introduced, followed by the adversary model. Although new attack patterns may emerge over time, a list of contemporary attack patterns is compiled in order to be able to be aware of common pitfalls and to check if color labeling can indeed avert these types of attacks. Existing defences against these attacks are discussed and evaluated in light of the adversary model.

## 2.2 Adversary Model and Assumptions on Hardware and Software

For a high-assurance application, preventing information leaks is of the utmost importance. Color labeling is meant to protect devices such as the PrimeLink. Such a system is typically protected against unauthorized physical access.

For the system, the following assumptions hold:

- the system is bare-metal: no operating system (OS) is present[1];

[1] This is not a hard requirement, with some modifications, the coloring concept could also be applied to processes running under

- the system under attack is brought up properly: code is untampered with and the system and all peripherals have been initialized properly[2];
- the system is protected against unauthorized physical access, only remote, software initiated attacks are possible.

The adversary is assumed to somehow have obtained a device and reverse engineered the system. Armed with this knownledge, he will then perform a remote attack on another, similar device. This means the attacker has:

- full knowledge of the code;
- full knowledge of the memory layout;
- no physical access to the device-under-attack.

These assumptions give lead to an aversary models that models memory out-of-bound access vulnerabilities that can read-out or corrupt all data values, including words in code memory. In some existing literature the adversary is assumed to have full data write access [14]. In that same model, however, the attacker is assumed not to be able to execute injected code, which means the write actions (at least initially) have to be performed with existing code. This system will be equiped with tagged memory (see Chapter 3), the value written to the tags will be subjected to a set of hardware enforced policies and cannot be directly set by the user. This statement is thus rephrased as "the attacker can write any data value to the memory", which only poses restrictions on the tag values, not the actual data values.

## 2.3 Weaknesses

Given the adversary model described in Section 2.2 and the restrictions imposed on the device under attack, possible attack methods are limited to remote, software-initiated attacks.

The type of attacks this work targets are low-level attacks, so attacks that target higher level concepts (algorithmic level, message passing, data structures) are excluded. The Common Weakness Enumeration website (CWE) [15] lists common weaknesses found in software programs and can present different views of the list. One such view, the *research view*, has two topics of particular interest:

- Incorrect Calculation;
- Incorrect Access of Indexable Resource ('Range Error').

Since the system used here also features hardware peripherals, access to which is potentially unrestricted in the absence of an OS, an additional entry is added:

- Incorrect Configuration of Hardware Peripheral.

Figure 2.1 shows a simplified overview of attack methods, based on the overview in [16], Figure 2.1.

Each attack is initiated by some user controlled input (depicted in white) such as keyboard input or a data packet received over an ethernet controller. This input abuses bugs (displayed in green) that are present

an OS. Not having an OS merely eliminates the possibility to use process separation, some protection mechanisms, virtual memory, and other features normally provided by such an OS.

[2]This requires the implementation of a safe-boot mechanism. The actual implementation is beyond the scope of this work

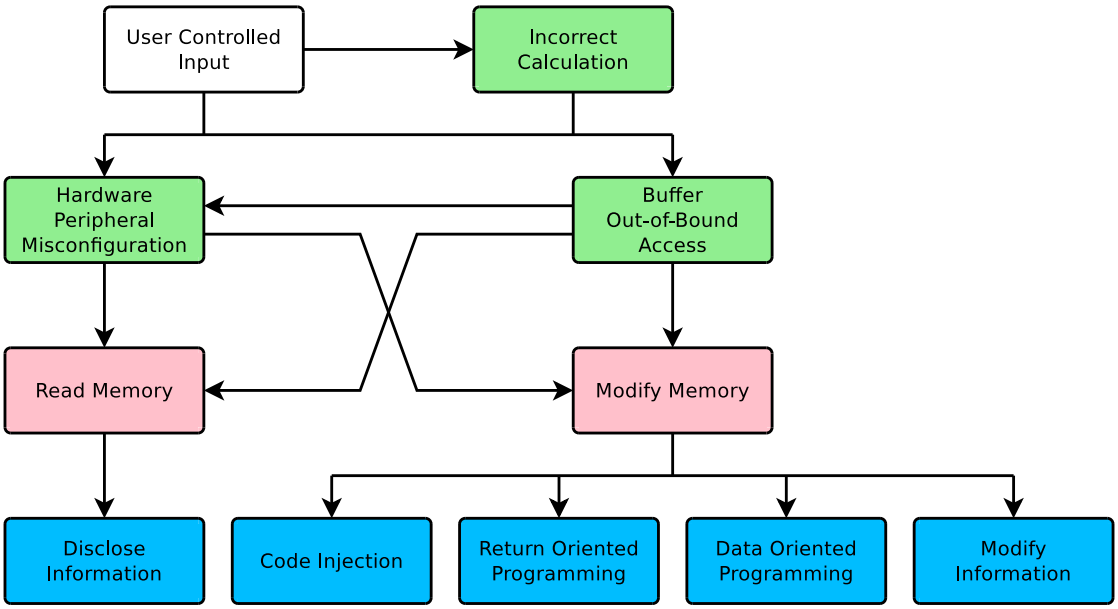



**Figure 2.1:** Simplified overview of attack methods

in the program to cause an integer overflow, a buffer-out-of-bound access, or to maliciously configure a hardware peripheral. These bugs give access to certain permissions (shown in purple) that lead to the exploits (in blue) at the bottom. Of course, in reality, many more causal relations may be present, e.g., executing injected code may be used to reconfigure a hardware module that in turn is used to read out a protected memory region.

**Incorrect Calculation** Incorrect calculation can mean incorrect calculation of buffer size, overflow of a numeric type, division by zero. Some of these calculations may cause an error (division by zero) or the results may cause logic errors in the program flow. An integer overflow (calculation causes a numeric type to overflow) is often the first step in a buffer out-of-bound access bug [17], but might also be used to misconfigure e.g., a DMA controller.

**Incorrect Access of Indexable Resource** Incorrect access of indexable resource or buffer out-of-bound access is the situation in which the bounds of a buffer are not respected and data outside of the buffer is read or overwritten. Conceptually, reads or writes can occur for indices that are either too small or too high. Buffers may reside on the stack (stack-overflow or stack-underflow) or the heap, or could be global data.

Buffer out-of-bound access is caused by incorrect or missing boundary checking and can originate from a simple loop with incorrect end-conditions, or e.g., format string errors. Depending on the hardware platform, out-of-bound access on the stack or heap can have different consequences. The x86 platform uses the stack to store return addresses, so out-of-bound writes on the stack may disrupt the program flow on such a platform. On most platforms, the stack is also used to store local variables, so out-of-bound writes can also cause local variables of functions called earlier to be changed or read out.

The heap, on the other hand, is commonly used to store dynamically allocated data structures, out-of-bound reads might read data conceptually residing in another scope or security domain. In the

case that C++ style objects are stored, function pointers may be present in the data structures as well and out-of-bound writes may be able to modify such pointers.

In case of memory-mapped peripheral configuration registers, buffer out-of-bound access might also modify the configuration of hardware peripherals.

**Incorrect Configuration of Hardware Peripheral** Some hardware peripherals can be configured to read from or write to specific sections of memory using DMA (Direct Memory Access) functionality. These addresses are often user configurable. An attacker could abuse this functionality by misconfiguring the hardware peripheral to read out or modify memory contents. It has been shown that, under certain conditions DMA controllers can be used to implement a Turing machine [18].

## 2.4 Attacks

Using the weaknesses described above an attacker can perform (a combination of) several different attacks, some common examples are described here. Figure 2.2a pictures the initial layout of the stack, the rest of Figure 2.2 shows how the stack is modified under different attack methods.

**Information Disclosure** Information disclosure, leaking confidential data, is commonly one of the end-goals of an attacker. A buffer out-of-bound read can be used directly to read out memory, as is shown in Figure 2.2b. Here, the contents of the `key` variable is read out by accessing array `a` with an out-of-bound index. Alternatively, memory contents can be leaked via injected code or return-oriented programming, or via a hardware peripheral via a DMA readout.

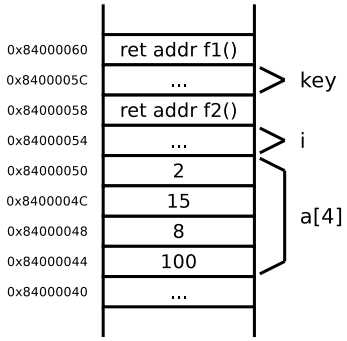
**Code Injection** Code injection is an attack in which instructions for the processor are "injected" as data into the memory by the attacker. This is usually the result of a buffer out-of-bound write e.g., stack overflow [19]. The attacker will then divert the program flow to the location of the injected code and thus execute the injected code. Figure 2.2c shows injected code on the stack and a modified return address in order to execute the injected code. The `ret` instruction will now divert the program flow to the malicious, injected code.

**Return Oriented Programming** With the emerging use of the w$\oplus$x (see Section 2.5) code injection is no longer easily possible. Return oriented programming (ROP) by-passes this defence by identifying so-called gadgets: pieces of code that contain a code section that performs the desired computations and a linking section [20] that transfers control to the next gadget. By chaining these gadgets, an attacker can create a program. This is shown in Figure 2.2d, return addresses have been placed on the stack, replacing an exisiting return address with the address of a gadget.

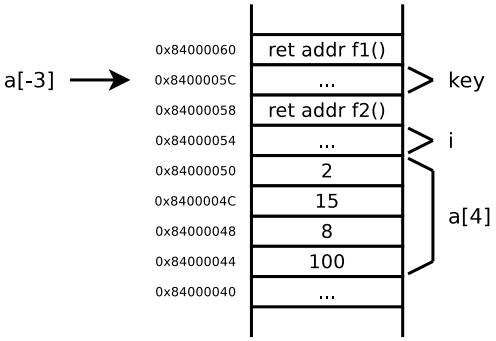
Originally, ROP used the return instruction in the linking section to chain together programs, hence the name. Later research showed that ROP without the use of the return instruction is possible as well [21]. Finding gadgets in a program can be automated [21].

Return-to-libc attacks can be considered to be a special case of return oriented programming.

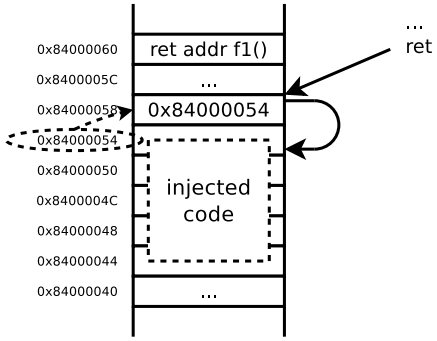
**Data Oriented Programming** With the growth in number of protection mechanisms in place to mitigate control flow attacks, attacks on non-control data elements are becoming more attractive for a potential attacker. The Heartbleed bug [2] is caused by an attack on non-control data. Data oriented programming further steps this up: by modifying key non-control data an attacker can execute data-oriented gadgets (similar to the gadgets in ROP) [22]. An example is shown in Figure 2.2e, in which a loop is controlled by malicious overwrites of the loop variable. In a DOP attack such a loop is used as a gadget dispatcher. Since these attacks do not modify control flow, they can



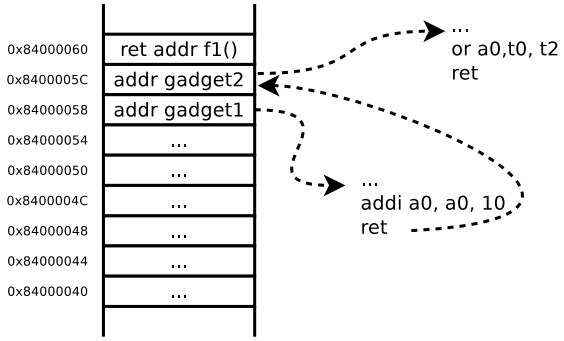

**(a)** Initial stack layout



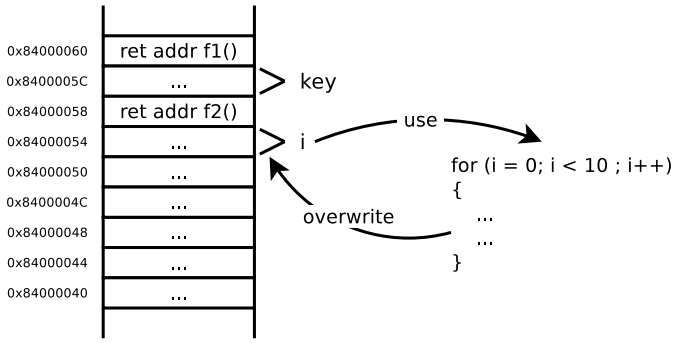

**(b)** Information disclosure via buffer under-read




**(c)** Code injection




**(d)** Return oriented programming (ROP)




**(e)** Data oriented programming (DOP)

**Figure 2.2:** Common attack methods

run successfully in the presence of control flow protection mechanisms such as ASLR and W⊕X. Detecting data-oriented gadgets and chaining dispatchers can be automated, enabling the creation of Turing-complete programs [22].

**Modification of Information** Modifying information can be another end-goal of an attacker: i.e. modifying the account number of a transaction may transfer funds to an attacker's account, adjusting the control system of some industrial process can render a competitor's plant useless.

## 2.5 Defence Mechanisms

The security community has come up with several defence mechanism to avert attacks, such as those described in 2.4. Here some widely deployed mechanisms are discussed and evaluated.

**Safe Languages** One option for reducing the risk of wrting code with buffer out-of-bound access errors, is by using a safe programming language. Such a language can check at compile time if buffer access is within the predefined boundaries. Optionally, run-time code can be used to try to detect out-of-bound access during execution. This is usually implemented with stack or heap canaries (see below).

Use of a safe language might require rewriting software, although the added safety might make it worthwhile for certain types of applications. Using a safe language can theoretically protect against all attacks described in Section 2.4. However, in case of bugs in the compiler or libraries, an attacker may still be able to maliciously read or write beyond buffer boundaries. Using a safe language will also fail to protect against autonomously reading or writing of the memory from a peripheral device.

**Canaries** Canaries are special values written immediately after buffers and before control data. In the most simple case, canaries are a special value, usually determined on program startup and protected against tampering and unauthorized read-out. If a buffer overflow occurs, this canary value will be overwritten. By verifying the canary value before accessing the control data, these buffer overflows can be detected. More complex strategies exist as well.

To a certain extend, canaries can be used to make ROP attacks more difficult, but unfortunately, in many situations this protection mechanism can be by-passed.

If an attacker is aware of the precise location of the canary (which would be case in the adversary model used here) or reads out the stack, the value can be obtained. The protection mechanism can then easlily be by-passed. Furthermore, canaries only protect against contiguous overwrites: it is very well possible to perform an out-of-bound write without modifying the canary. Canaries also don't protect other data buffers against illigitemate overwrites.

**Data Execution Protection (W⊕X)** Attacks such as stack smashing [19] allow an attacker to inject code and divert the program flow to execute this new code. The injected code resides conceptually in data memory. Data Execution Protection or W⊕X (write xor execute) is used to make a memory page either writable or executable, but never both. This will generally prevent code injection attacks via stack and heap based buffer overflows. This protection mechanism is commonly implemented on the level of memory pages. Major hardware architectures, including AMD64, ARM, and SPARC, contain hardware support for this mechanism. Although code injection attacks can be prevented with this technique, code re-use attacks such as ROP will not be affected.

**Address Space Layout Randomization** Address Space Layout Randomization (ASLR) is a technique in which the key addresses of a process are randomly re-arranged in order to prevent an attacker to be able to use persistent memory addresses to jump to exploitable code segments. This technique is based on reducing the statistical probability of an attacker finding specific memory elements, e.g., the top of the stack, the address of a certain function. In order to be as effective as possible, the addresses have to be randomized as much as possible and thus depend on the systems' architecture, the OS, and the frequency at which the randomization takes place (compile time, boot time, process initialization, dynamic).

ASLR is used to make attacks such as code injection, ROP, and DOP much more difficult. Also reading out or modifying key memory locations is more difficult.

ASLR is sensitive to attacks that reduce the amount of entropy in the addresses. Furthermore, ASLR can be bypassed when non-ASLR modules are mapped in the process address space or the value of a pointer is leaked via e.g., a memory leak. Even hardware based side-channel attacks can be used to defeat ASLR [23].

## 2.6 Speculative Execution Side-channel Attacks

On Januari 3, 2018, the Spectre [24] and Meltdown [25] vulnerabilities were disclosed. These vulnerabilities were the first in an ever-growing series of side-channel based attacks centered around speculative execution (the latest addition at the time of writing being SWAPGS, or "Grand Schemozzle" [26], disclosed at August 5, 2019). These attacks are based on observing an internal state of the processor caused by speculative execution, by means of a side-channel, such as cache timing. This class of attacks is extremely difficult to detect and mitigate, and might not even be completely avoidable at all [27]. These side-channels are a side-effect of hardware implementation. Proper mitigation of these problems requires modifications of the hardware in order to prevent creating an internal state that can potentially leak sensitive information. This will have to be taken into account while designing the hardware implementation of the color labeling concept.

## 2.7 Conclusions

There are many sophisticated attack methods available to an attacker. The security community has in many cases failed to come up with proper defence against these attacks. Many of the existing defence mechanisms have an ad-hoc character with a limited scope and often can easily be by-passed. This holds true for some of the hardware implemented defence mechanisms as well. Most of the defence mechanisms do not protect against reads and writes originating from a peripheral.

In order to overcome the limitations of the current defence mechanisms, this work utilizes tagged-memory. The tags enable addtional information to be associated with memory locations.

# Chapter 3

# Memory Protection Mechanisms and Tagged Architectures

*"When intuition and logic agree, you are always right."*
*– Blaise Pascal*

## 3.1 Introduction

As an alternative approach to the more ad-hoc defence mechanisms described in Section 2.5 several alternative memory protection mechanisms have emerged over the years. Recently, with the decreasing costs of memory and the increase in costs of disruption of critical computer systems, there is an increasing interest in using so-called tagged architectures. In such an architecture, each memory element has an associated tag of one of more bits containing meta-data of the memory element. This chapter discusses some of the more interesting memory protection mechanisms from literature and additionally looks at some of the recent developments in tagged architectures.

## 3.2 Memory Protection Mechanisms

As seen in Chapter 2 many, if not most, low-level security problems are related, directly or indirectly, to buffer out-of-bound access. This has driven research into finding ways to protect memory via compiler modifications, binary patching, hardware modifications, and any combination of those.

In many cases, researchers have tried to find ways to protect memory that are minimally invasive, at different levels. Some solutions do not require recompilation, others only minimal modifications to the source code. Some solutions that propose hardware modifications, can also work without those, albeit with a performance penalty.

Many traditional protection schemes work at the page level. This only offers a coarse level of protection. One of the first exceptions is the Mondrian system [28] which enables memory protection at the individual

word level. This is accomplished with the aid of protection tables that themselves reside in the memory and offers the following permissions: read-only, write-only, execute-read, or no permissions at all. By adding a form of caching, the performance overhead of this system is kept reasonably small at less than 8% additional memory references. Tagged architectures can be utilized to implement similar functionality in hardware.

## 3.3 Tagged Architectures

The idea to associate additional information with the value in a memory location or a register using additional tag bits is far from new: the first commercial use of a tagged arthitecture is the Burroughs Large Systems B5000 of 1961, in which a single tag bit was used to denote either a control word or a numerical word. In later iterations of the machine, more tag bits were added that were used to store more type information about the associated word.

In many older x86 computers parity bits were utilized in the memory, in modern computer systems additional bits are used in ECC memory to detect double and revert single bit faults.

Adding tags to a computer system requires a modification of the hardware architecture. Many research projects try to improve the memory protection of existing architectures (specifically x86 or ARM), which precludes the use of tagged memory. There are, however, several modern architectures in which tagged memory is incorporated.

### 3.3.1 Oracle SPARC M7

The Oracle SPARC M7 and M8 processors feature Silicon Secured Memory (SSM) [29] that tags pointers and memory lines with a "version". The pointer and the accessed memory location must have the same version tag, otherwise an exception is generated. The mechanism is used to detect different memory errors, such as buffer overreads, memory writes from different threads, and use of dangling pointers (use-after-free). The policies are fixed: only access from an identically versioned pointer is allowed. Only a small number of different "versions" are available (4 bits per 64-bit memory element).

The mechanism requires toolchain and run-time library support. The programmer does not need to modify the source code to utilize the mechanism.

### 3.3.2 Intel MPX

The Intel MPX extension [30] for x86-64 was introduced in the Skylake architecture from 2015. This extension can be used to limit pointer use by storing base and bound information in special registers. It is not a fully tagged architecture, but instead only provides tags to pointer variables. OS, Compiler and library support is required to fully support MPX.

MPX turned out to contain problems and limitations in the implementation. Combined with the lack of support from Intel programmers, support for MPX is to be dropped from both GCC and the Linux kernel.

### 3.3.3 RISC-V

With the advent of the open source RISC-V instruction set architecture, a renewed interest in tagged memory architectures can be observed, mainly motivated by the potential increase in security. All these

RISC-V implementations use tag bits to annotate registers or memory words, but have quite different goals and means to achieve those.

#### LowRISC

LowRISC is an open-source implementation of the 64-bit RISC-V architecture aimed at secure computing. The LowRISC is equipped with tagged memory [31]: by default two tag bits are allocated per 64-bit word. These bits are used to generate an exception on read, write, both read and write access, or they can be inactive. The intended use is to make e.g., stack canaries, vtables, and function pointers read-only. Toolchain and libraries should be updated to support the tags. Tags can be modified with newly added instructions.

#### Dover

Dover [32] is a RISC-V implementation that uses a separate processing unit in order to process the metadata in the tags programmatically (software-defined metadata processing). The system is extremely flexible since tag handling is done via separate micro-policies that run on the Programmable Unit for Metadata Processing (PUMP). Metadata can only be accessed from the PUMP. The tags are 32-bit wide and each memory word and register has an associated tag.

The Dover system is complemented with a special module that is used to manage peripherals and DMA controllers that do not have any knowledge of tags.

#### Shaki-T

Shaki is an implementation of the RISC-V architecture, Shaki-T [33] adds support for hardware tag bits. These tags are mainly used to restrict the power of pointers. Using a combination of hardware supported tag bits and in-memory data structures, pointer meta-data such as base address and bounds are stored in memory and used to prevent illegal pointer arithmetic and invalid pointer use. Both spatial as well as temporal (dangling pointers) misuse of pointers is prevented. Compiler support is required to add additional instructions in the appropriate places.

## 3.4 Conclusions

The described tagged architectures all target different problems (one could say Dover tries to target all). Using the combination of software support and hardware enforcement a much greater trust in the protection mechanisms can be provided. With the exception of Dover, no attention is paid to the incorrect or malicious use of hardware peripherals. All architectures covered require some form of software support: OS, toolchain, compiler, or any combination of these. This makes wide adaptation difficult, but for specific purposes, it can increase the trust in software.

None of the architectures specifically utilizes tags for security domain purposes, although the Dover RISC-V implementation might possibly be used for that purpose.

# Chapter 4

# Test Program and Requirements

*"Programming is the art of telling another human being what one wants the computer to do."*

*– Donald Knuth*

## 4.1 Introduction

As briefly shown in Chapter 1, associating security domain information to memory locations can potentially prevent cross-security domain information leaks. Chapter 2 introduced a number of common attack patterns this work should be able to withstand. It also showed a number of shortcomings in existing defences. Chapter 3 introduced tagged architectures that have the potential to address the security issues at the root by providing a way to associate security domain information to memory locations.

In order to find out what functionality would be required in a system that labels data with security domains, a simple test program was written. The program would identify both conceptual requirements as well as requirements for the programming interface that a programmer would need to utilize the protection mechanisms.

Using the test program as a top-down approach and the attack patterns as a bottom up approach, the requirements and priorities for both the protection mechanism as well as the programming interface were derived.

## 4.2 Use Cases for the Concept

As discussed before, color labeling targets high assurance applications on an embedded platform. The PrimeLink is a typical example: the box features two network interfaces, residing in two different security environments. Via well-defined policies, traffic from one interface can be routed over the other interface. A simplified version of this situation is shown in Figure 4.1. This shows the transfer from black domain to red domain:

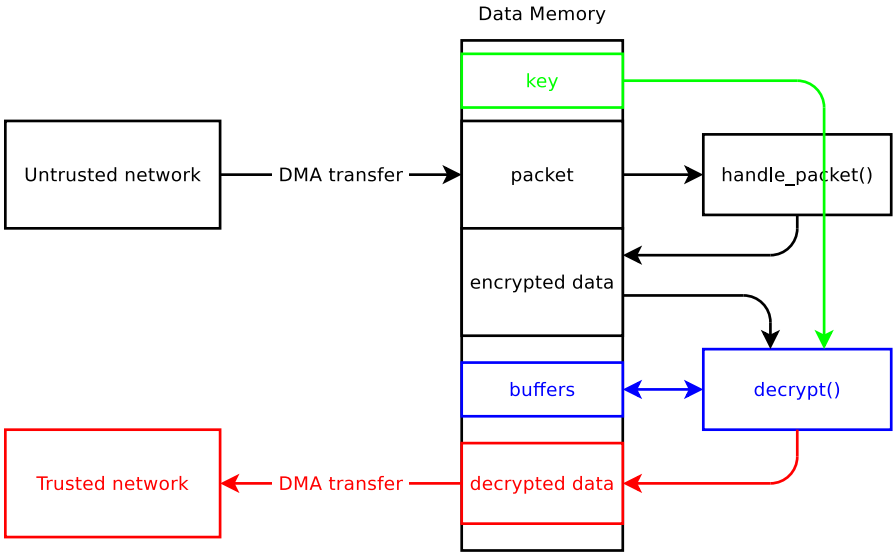



**Figure 4.1:** Simplified overview of a system like the PrimeLink

- Data arrives over the black network and is transferred via a DMA controller into the data memory;
- Software residing in the black domain now processes the data: the packet is processed and the payload is extracted;
- The payload is then decrypted (using a key that resides in yet another security domain) with a decrypt function that runs in a new color in order to prevent leaking partial information to any of the other domains;
- The decrypted data is encapsulated in a new packet and transferred to the red network interface via a DMA transfer and transmitted over the red network.

The black and red data conceptually belong in separate security domains, color labeling should enforce this domain separation throughout the entire system. As can be seen, the initial colors are set by the ethernet controllers.

## 4.3 Analysis of Key Exchange Program

An example program that works with data of different security domains is a key exchange. A key exchange is used to establish a secure connection between two machines. Using a public and private key pair, a key can safely be distributed over an unsecure network and subsequently be used to encrypt further communications. This is similar to the situation depicted in Figure 4.1.

The key exchange program is based on the Diffie-Hellman key exhange [34]. In such a key exchange data from different security domains is used to finally derive the key used for the communications. In the test program the secret key is used to encrypt and decrypt some data as well. This test program can be found in Appendix C, note that for this small example all code is included in a single file.

The code is based on the example from [35]. In the program, the following steps are taken:

- First, public key `pub_key_A` is derived from public data `g` and `p` and secret key `a` and send over the public network;
- Based on the received public key `pub_key_B` and `g` and `p` the symmetric key `symm_key` is calculated;
- Using the symmetric key, some secret data is encrypted and send over the public network;
- Some encrypted data is received over the public network and, using the symmetric key, decrypted into secret data.

Variables and functions in the code have been annotated with comments, denoting the color and security domain. Analyzing the annotated program gives following insights:

- Some data resides in specific security domains. At least the following domains can be identified:
  - Cipher text (encrypted data);
  - Plain text (unencrypted or decrypted data);
  - Keys;
  - At least one special domain for functions handling data from multiple security domains. Using a single domain per such function may be expedient.

  The programmer should be able to specify the color of variables to residing in non-default security domains.
- Even in this example program that was deliberately created to require the use of multiple security domains, a number of functions can be implemented as color-agnositic functions. These functions are not aware of the color concept and process input and output in a single security domain. For these functions no special functionality is required.
- Functions that require handling data from multiple security domains should be able to break the domain separation. These functions should run in a special color. The programmer needs to be able to specify the colors of the parameters, the return value, and the run-color. The hardware should support the domain breakage under controlled circumstances.

Using this information, the requirements of the concept can be defined.

## 4.4 Derived Requirements

The following requirements for color labeling have been derived from the attack patterns, the code of the test program, and Figure 4.1.

**Functional Requirements** The following function requirements are derived:

- Different colors should be available for at least: black and red data, keys (green), and separate colors for each function that handles multiple colors (like `decrypt`);
- Color labeling is not only limited to the CPU and memory, the peripherals should be aware of coloring as well;

- There should be a way to break the domain separation and access multiple colors from a specific security domain (functions like `encrypt` and `decrypt`);
- Color labeling should be able to properly defend against the weaknesses and attack patterns described in Section 2.3 and Section 2.4, for the adversary model and the assumptions described in Section 2.2.

**Programmer's Interface Requirements** From the test program the following requirements regarding the programmer's interface have been identified:

- The programmer should be able to annotate certain variables with a color;
- The programmer should be able to annotate special functions: run-color, colors of the parameters and color of the return value should be specified;

**Implementation Requirements** The hardware and software platform have been defined by the client:

- In order to reduce the trust in the compiler, the policies of the concept should be hardware enforced;
- For the hardware platform the RISC-V architecture should be used;
- For the software platform the Rust Programming Language should be used.

## 4.5 Conclusions

A simple test program is used for a top-down approach of color labeling. The weaknesses and attack patterns identified before are used as a bottom-up approach. Using both approaches, the requirements of the concept have been defined.

# Chapter 5

# Color Labeling Concept

*"We cannot solve our problems with the same thinking we used when we created them"*

*– Albert Einstein*

## 5.1 Introduction

This chapter describes the color labeling concept and motivates the design choices. First, a short overview is given. Subsequently, the data protection mechanism is explained, followed by the support for handling data from multiple security domains. The control flow integrity mechanisms are explained next. Implementation specifics for the RISC-V and a general CISC architecture are discussed. Finally, the concept is theoretically evaluated using the attack patterns described in Chapter 2.

## 5.2 Color Labels

As stated in Section 1.3 the main goal of color labeling is to prevent cross-security domain information leaks. Programmatic errors like buffer over-reads within the same security domain are not considered to be cross-security domain violations. Buffer over-reads and overwrites will never directly cause cross-security domain leaks, as will be explained in Section 5.3.

The security domain of a data element is denoted with a color that is stored in a label. Color labels are 32-bit tags associated with 32-bit memory words. Labels contain the color value and a number of special bits related to control flow integrity (discussed later, see Section 5.5). The labels are conceptually stored in a separate memory, although different implementations are possible on an actual hardware implementation (see Chapter 10). The labels are not accessible from a running program, except via special and restricted instructions.

In typical high assurance applications, we can identify at least three security domains:

- Encrypted data;

- Unencrypted data;
- Key data.

Each of these security domains should be assigned a unique color. Furthermore, each routine handling data from multiple security domains, should run in a separate color in order to prevent intermediate results from leak to other domains (see Section 5.4). The optimal number of colors will depend on the way software developers will utilize the coloring mechanisms. With an increasing codebase that uses labeling, these kind of usage statistics are expected to become apparent.

The CPU runs in a certain color that can only be changed with special restricted instructions. The color of the CPU determines the current security domain and is used to determine if certain data is accessible. Color labels are added throughout the computer system and propagate over the system bus, the memory, the caches, peripherals, etc.

The label data is written by a special loader program on program startup and can only be changed using special instructions (see Section 2.2).

The concept is designed for a generic RISC architecture, but implementation on other architectures is possible as well. In some cases however, this might incur a performance penalty.

## 5.3 Data Protection

The data protection mechanism is implemented by associating color labels with individual memory words and restricting operations on the data, based on these label values.

### 5.3.1 Memory Access

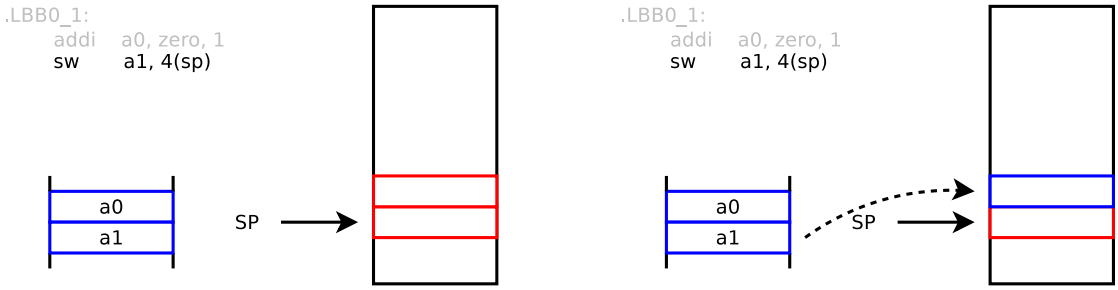
During memory access, labels are propagated with the data without performing checks:

- When reading data from memory, the label data associated with the memory word is copied to the label of the target register;
- When writing data to memory, the label data associated with the register is copied to the label of the target memory location;

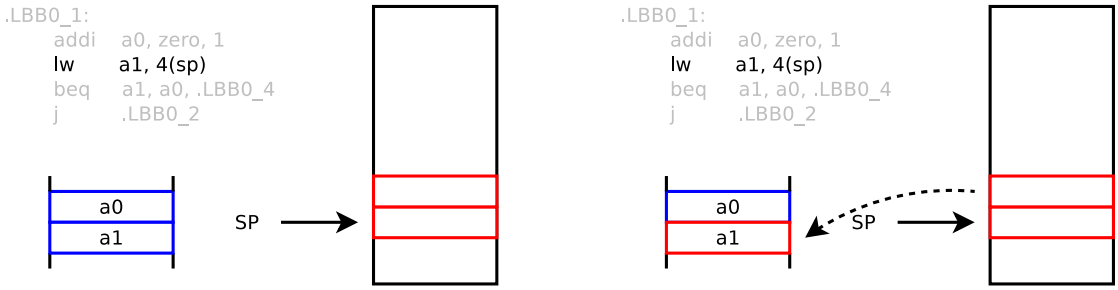
This means that memory access (both read and write access) will never cause an exception due to a security domain violation. This is schematically shown in Figure 5.1.

### 5.3.2 Register Access

Each register has has an associated color label. During execution of an instruction, the labels are checked against the current color of the CPU. If the labels match, the instruction can perform its operation and write back the result. The label of the target register is overwritten and will (for normal non-color related instructions) ultimately be identical to the current color of the CPU. If the color of the source registers is not identical to the color of the CPU, a security violation has occured and an exception will be generated. Examples are shown in Figure 5.2.

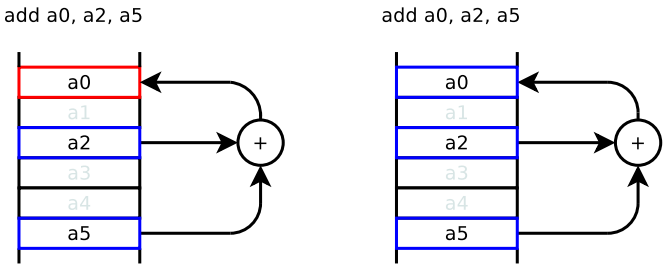



**(a)** Storing the contents of a register into memory will also copy the label data of the register

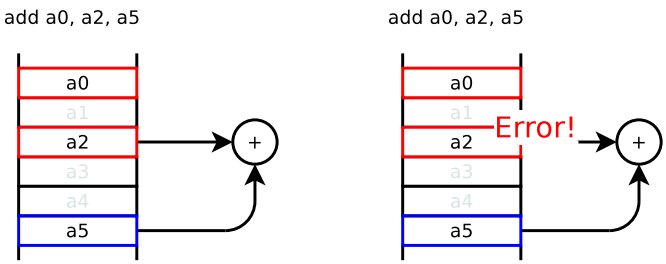



**(b)** Loading a memory word will also copy the associated label data to the destination register

**Figure 5.1:** Accessing memory




**(a)** Correct register access, destination register label overwritten




**(b)** Invalid register access: cannot read red register

**Figure 5.2:** Accessing register file with processor running in color blue

### 5.3.3 I/O Access

Reading from the I/O device will work like reading from memory: a color label is added to the data read from the I/O device. During writes to an I/O device, the label of the data is checked against the label of the colored device. If these labels are different, a bus error is generated.

## 5.4 Handling Multiple Colors

In well-defined locations in the program the security domain separation has to be broken and data from multiple domains has to be combined. A representative example is an encrypt routine, in which unencrypted red data has to be be combined with a green key in order to produce black encrypted data.

Since in these situations the security assurances offered by the color label concept are deliberately bypassed, support for these functions has to be implemented with care:

- Partial results that contain information from multiple domains should not be able to leak to any other domain. Functions handling multiple domains should run in a separate domain;
- Input data from the different domains should be readable by or converted into the color of the function;
- The results of the function should be converted into the target color.

Several different options for reading input data and writing results are possible:

- Processing the data in a super-color that can read and write all colors;
- Converting input data and results at caller;
- Converting input data and results at callee.

The first option makes for a very powerful function that will be a likely target for ROP attacks. Any exploit that might enable the use of this super-color will completely void the protection offered by the coloring concept. For this reason, this option is discarded.

The other two options make it possible to split the functionality of converting to and from a specific color. Code in a high assurance application is subjected to a security audit. This audit is simpler when fewer lines of code have to be verified. When (part of) the conversion takes place at the *caller*, security sensitive code is spread over more lines of code. Each call to the special function will add additional security sensitive code. When the conversion is done at the *callee* the special function forms a self-contained block and can be verified independently of other code. For this reason the third option, doing the conversion at the callee, is chosen. This is shown schematically in Figure 5.3. The differently colored inputs are converted into the run-color of the function (blue in the example). The body of the function is running in blue and thus all results are in blue as well. The final step is then to convert the final result into the target color (black in this case).

Code that is converting the colors of the input and output data represents an interesting target for a potential attacker, for example as ROP gadgets. Care should be taken that:

1. Inputs are only converted from a compile time specified color;
   By specifying the from-color, it is not possible to convert data from other colors. By having this from-color as a compile time constant, a potential attacker cannot modify the from-color via a data-oriented attack.

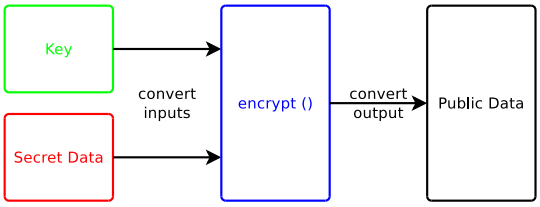



**Figure 5.3:** Encrypt operation

2. Outputs are only converted into a compile time specified color;
   By specifying the to-color as a compile time constant, a potential attacker cannot modify the to-color via a data-oriented attack.

3. The routine is always executed as a whole: it has to be entered at a known entry point and should be executed in its entirety.
   If the program flow cannot be diverted from executing the routine as a whole, no unforseen color changes are possible.

### 5.4.1 Converting Inputs and Outputs

The color labels are stored in memory that is not accessible from a normal user program. In order to convert input and output labels and at the same time adhere to the first two requirements, special instructions are added to the RISC-V core.

Input parameters should only be converted from a compile time known color. One option would be to specify both the "from" and "to" color, however, this would generate a very interesting instruction for an ROP gadget. Instead, only the "from" color is specified and hardcoded as an immediate field in the instruction, the label color is always set to the current color of the CPU. The "from" color is checked against the register color and an exception is generated on a mismatch.

The following instruction is added to convert input registers into the current run-color:

```
reg_from_color rd, color
```

Here, `rd` is the register whose color will be changed, `color` is the numerical representation of the expected color of the register `rd`.

Output results are generated in the run-color of the function and should only be converted into a compile time known result color. The following instruction is added to convert result registers:

```
reg_to_color rd, color
```

Here, `rd` is the register whose color will be changed, `color` is the numerical representation of the requested result color. The register has to be of the current run-color.

### 5.4.2 Tints for Control Flow Integrity

In order to fullfill the third requirement, functions that handle multiple colors are split into three parts, which are denoted as tints:

- In light tint, colors of input parameters will be converted;
- The body of the function is executed in normal tint;

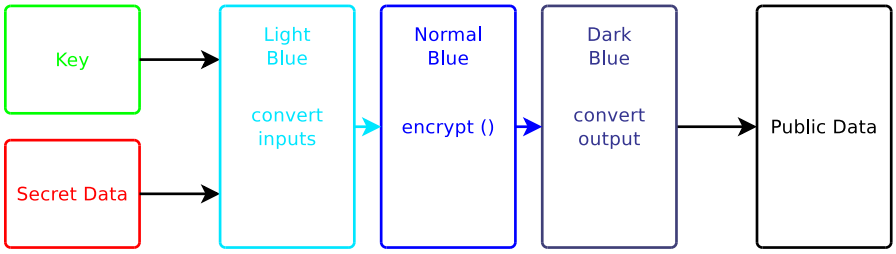



**Figure 5.4:** Encrypt operation with tints

- The result is converted into the target color in dark tint.

This is shown in Figure 5.4. The tint mechanism is used to enable or disable certain instructions in order to ensure control flow integrity. The current execution tint is, just like the color, stored in the CPU. In order to change the execution tint, three additional instructions are added:

```
chg_color_light color
chg_color_normal color
chg_color_dark color
```

Here, `color` is the numerical representation of the color.

**Light Tint** is the start of a special routine that handles multiple colors. Switching to light tint should be the first step in a special function. In light tint:

- only the `j` form of `jal(r)` allowed;
- `chg_color_normal` with `color` identical to the current color is allowed;
- `reg_from_color` is allowed.

The `jal(r)` instruction (jump and link (register)) is used for (indirect) subroutine calls or jumps. These are disallowed in light tint, to ensure that the special possibilities of the light tint, specifically the ability to use the `reg_from_color` instruction, cannot be abused by program flow divertions. Note that on RISC machines the `j` (unconditional jump), `jr` (jump register), and `ret` (return from subroutine) instructions are specialized forms of the `jal` and `jalr` instructions. In light tint only the encoding for the unconditional jump `j` is accepted. In other architectures, similar instructions should be restricted.

Changing to normal tint can only happen in the same run-color and has to be specified explicitly. The color parameter of the `chg_color_<tint>` instructions is encoded as an immediate in the instruction itself. Since code integrity on startup is ensured (see Section 2.2), explicitly specifying this color ensures that the code in light tint cannot be used as a gadget. If the light tint is executed in a certain color, it has to proceed into normal tint in exactly the same color, and thus security domain, thereby ensuring that the special routine is executed as a whole.

When the input parameters of the functions are required to have a compile time known size, code in light tint can be automatically generated by the compiler.

**Normal tint** is used in standard operation. In normal tint:

- `jal(r)` is allowed;

- `chg_color_light` with `color` representing any color is allowed;
- `chg_color_dark` with `color` identical to the current run-color is allowed.

In normal tint, function calls are not restricted. It is possible to enter a special routine and thus enter light tint in a different color. It is allowed to change to dark tint as well, since the current routine can be part of a special routine. In that case, however, the color parameter should be identical to the current run-color, similar to the use of `chg_color_normal` from light tint. Again, this ensures that the special routine is executed as a whole.

**Dark tint** is used to convert results from the run-color of the special routine into a specified color. In dark tint:

- only the `j` and `ret` forms of `jal(r)` are allowed;
- `reg_to_color` is allowed;
- `ret` has the special function of changing the run-color.

Since the dark tint is used to convert the results, the `jal(r)` instruction is disallowed, similar to the light tint. The `reg_to_color` instruction can be used to change the color of registers in the run-color of the special routine into a specified color. The code in the dark tint forms the last part a special routine. A `ret` instruction executed in dark tint will:

- return to the caller routine;
- set the CPU run-color to the original color of the caller routine;
- set the CPU run-tint to normal.

The color of the caller is derived from the color of the return address, that has been generated by a `jal(r)` instruction. Code in dark tint can, similar to code in light tint and under similar conditions, be generated automatically by the compiler.

A schematic overview of the steps taken in a special function is shown in Figure 5.5. An overview of which instructions are allowed in each tint is shown in Table 5.1. Other than the described exceptions, all tints have the same behaviour with regards to the data protection as described in Section 5.3.

## 5.5 Control Flow Integrity

The data protection mechanism will protect against security domain violations due to errors such as buffer over-reads as long as the control flow of the program is ensured. Control flow integrity is required in order to thwart attacks such as code injection and ROP (Return Oriented Programming [21]). Control flow integrity is implemented through two special label bits:

- Code Bit
- Callable Bit

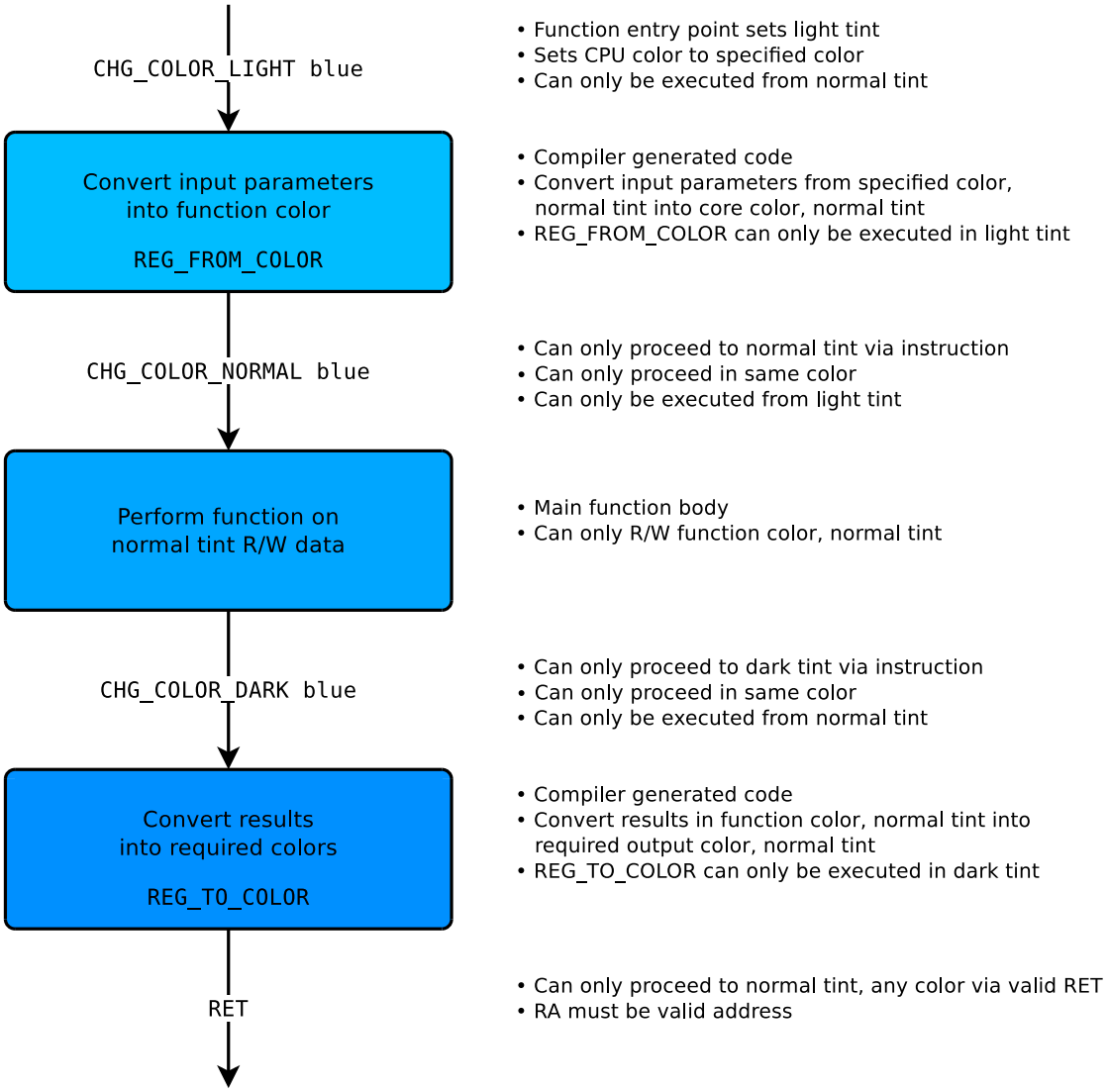



**Figure 5.5:** Overview of a special function

**Table 5.1:** Overview of allowed instructions in different tints

| Instruction | Tint | | | Remarks |
|---|---|---|---|---|
| | Light | Normal | Dark | |
| `reg_from_color` | ✓ | | | |
| `reg_to_color` | | | ✓ | |
| `chg_color_light` | | ✓ | | To any color |
| `chg_color_normal` | ✓ | | | Only in same color |
| `chg_color_dark` | | ✓ | | Only in same color |
| `jal(r)` | | ✓ | | |
| `j` | ✓ | ✓ | ✓ | |
| `ret` | | ✓ | ✓ | Special functionality in dark tint. |

### 5.5.1 Code Bit

The *code bit* denotes whether the corresponding data word contains code (value 1) or data (value 0). Only words with the code bit set can be executed, only words with the code bit cleared (and thus representing data) can be read and used as data in normal (arithmetic, logic, etc.) instructions. The code bit cannot be set programmatically and should thus be set by the program loader.

Since the code bit cannot be set programmatically, code injection attacks are impossible.

### 5.5.2 Callable Bit

The RISC-V instruction set contains a number of jump and branch instructions, as described in [12]:

- Conditional branches: `beq/bne/blt(u) <rs>, <rt>, <imm>`
- Unconditional jump and link: `jal <imm>`
- Unconditional indirect jump and link: `jalr <rd>, <imm>(<rs>)`

The displacement of the conditional branches and the unconditional jump and link is hardcoded in the instruction itself (in the immediate field). However, in case of the unconditional *indirect* jump and link, the displacement is determined by the sum of the value in the register with the number encoded in `rd` and the immediate field. Since the displacement is not determined at compile time but at run time, this might be used by an attacker to divert the control flow.

The indirect jump instruction `jalr` is used in several standardized ways:

1. As a `ret` instruction (`jalr zero, ra, 0`) for returning from a subroutine;
2. As a far call (`auipc t1, offset[31:12]; jalr ra, t1, offset[11:0]`) to call a far-away subroutine;
3. As a call to a subroutine with the address loaded from a jump table.

In the first case, the `ret` instruction is used to return from an earlier subroutine call. The return address is thus written by the `jal` instruction. In the second case, the actual address of the routine that control is transferred to, is known at compile time, but simply out of reach of the 20-bit immediate offset or should be a PC (program counter) relative jump, such as in PIC (position independent code). The third use-case is a jump table that the compiler can generate for certain language constructs. In all these cases, however, the address is either known at compile time, or known to be the result of a call from a valid address (code is started from a secure boot mechanism, see Section 2.2).

In order to ensure that indirect jump and link instructions will only accept values in the source register that represent valid addresses, the *callable bit* is introduced. If the callable bit is set, the associated register or memory location contains a valid address. If the source register of a `jalr` instruction does not have the callable bit set, a security violation is triggered.

For compile time constant jump addresses, the callable bit will be set by the program loader, similar to the color labels and code bits. The `jal` and `jalr` instructions as well as the `auipc` instruction will also set the callable bit in their destination register. Any subsequent operation that modifies the *value* in the register will clear the callable bit.

Note that the immediate field in the indirect jump and link instruction is determined at compile time and, given the secure boot scenario and the code bit, could not have been modified after the program has been loaded into the memory.

### 5.5.3 Limitations

The control flow protection mechanisms in the form of the code bit and the callable bit have consequences for some language paradigms and constructs. The following techniques cannot be combined with these control flow protection mechanisms:

- JIT (Just-in-time) compiled code: code bit cannot be set programmatically;
- Run-time modifications of function-pointers: callable bit will be cleared;
- `setjmp/longjmp`: callable bit not set;
- Self-modifying code (virusses, caching instruction emulators): code bit cannot be set programmatically;

These limitations can mostly be circumvented without much work, although with a performance penalty in some situations. JIT compilation can be avoided completely by just interpreting the code, instruction simulators can be implemented as non-caching. Support for `setjmp/longjmp` is somewhat more involved but can be implemented under some restrictions.

## 5.6 Implementation on the RISC-V

The concept is generic and not specifically tied to a hardware architecture. For this work the RISC-V architecture is targeted, implementing the concept on this architecture has some implications that are described here.

**Data Protection** The RISC-V has the `x0`/`zero` register that always reads as 0 and silently ignores writes. This register does not have a label check: any reads of the register will aways succeed.

The RISC-V is a load/store architecture, the processor performs all operations on register operands. Since the label is propagated but not checked on memory access, the delay in the memory access path will not increase.

**Tints** The RISC-V has the `jal` and `jalr` instructions, `j`, `jr`, and `ret` instructions are special forms of one of these instructions [12]. This means that in light and dark tint only specific encodings of these instructions are allowed.

**Control Flow Integrity** A far-call or far-jump is implemented via a combination of instructions: `auipc x6, offset[31:12]/jalr (x0/x1), offset[11:0]`. The `auipc` (add upper immediate to pc) instruction has to set the `callable` bit in the target register, otherwise no valid address is generated. The address is a compile time constant, encoded as immediate in the instruction and can thus not be set by a potential attacker.

## 5.7 Implementation on a CISC Architecture

The concept can also be implemented on a CISC architecture. Most CISC architectures do allow memory operands in arithmetic and logic instructions. This means that not only the register labels have to be checked against the processor run-color, but the labels of memory operands as well. This might introduce additional delay in the memory path and can thus reduce the performance of such an implementation.

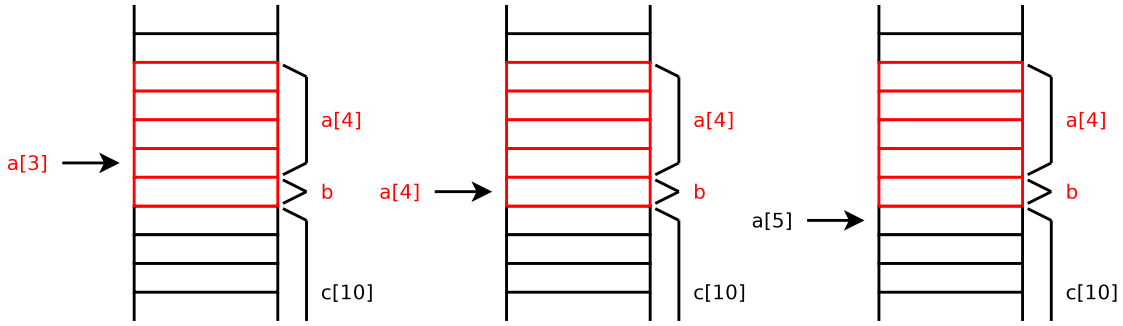
## 5.8 Theoretical Evaluation

Even without an actual implementation of the concept (simulation or actual hardware) it is possible to evaluate the concept by seeing how it will hold up against the attacks, weaknesses, and exploits described in Chapter 2.

**Buffer Out-of-Bounds Access** In case of a buffer out-of-bounds access, the coloring mechanism prevents cross-security domain information leaks: it is still possible that a buffer experiences overflow or over-read, however, this will not lead to a cross-domain information leak.

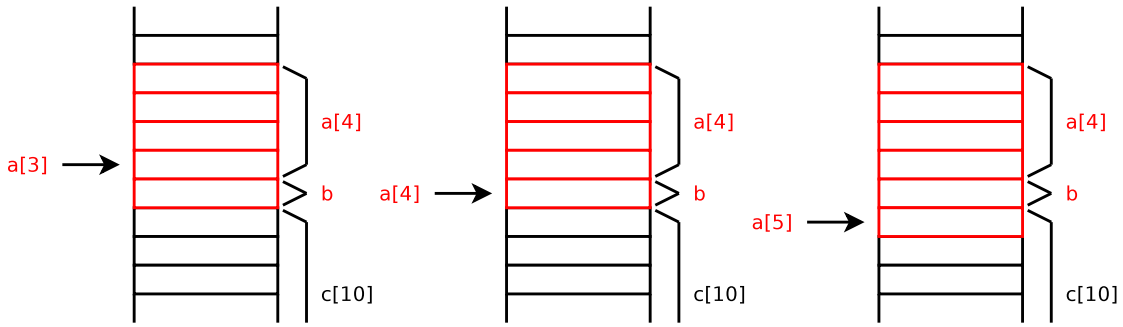
In case of a buffer over-read, the situation is shown in Figure 5.6. When reading memory the label always propagates with the data. The over-read itself will thus never generate an exception. When the over-read reads in data from a buffer that belongs to another security domain (Figure 5.6c), the read-in register will have the color of that security domain. Subsequent processing of the register will than cause an exception. In case that buffer belongs to the same security domain (Figure 5.6b), however, no exception is generated.

In case of a buffer out-of-bounds write, the situation is shown in Figure 5.7. Since the label will always propagate with the data, a buffer overflow will overwrite the label of the overflowed elements with the color of the current security domain (Figure 5.7c). The overwrite itself will thus never generate an exception. When later on the overwritten data is being read and used, an exception will be generated if the data in the read-in register belongs to another security domain. Again, when the overwritten buffer elements belonged to the same security domain as the original buffer, the color is not changed (Figure 5.7b).




**(a)** In-bounds read **(b)** OoB read, same domain **(c)** OoB read, other domain

**Figure 5.6:** Out-of-Bounds (OoB) reads of red array `a[]`




**(a)** In-bounds write **(b)** OoB write, same domain **(c)** OoB write, other domain

**Figure 5.7:** Out-of-Bounds (OoB) writes of red array `a[]`

**Incorrect Configuration of Hardware Peripheral** When a hardware peripheral is, unintentionally or by malicious intent, misconfigured it may perform unintended read and write operations.

Since hardware peripherals also have an assigned color, only data from the specified security domain can be read out. Reading in data from a different security domain will generate an exception.

In case of a misconfigured write action, the label of the data written is colored with the color of the peripheral, similar to a buffer overwrite. No exception will be generated.

**Information Disclosure** Information disclosure should be prevented by the labeling concept. Attacks focussed on information disclosure mostly utilize some form of out-of-bound memory access. With properly chosen data colors and correctly implemented special routines, the data protection mechanism ensures no information can be disclosed from one security domain to another.

**Code Injection** Code injection attacks are usually initiated with a memory out-of-bound write that will overwrite data on the stack with the attacker's payload and a dispatcher that diverts program flow to the attacker's code [19]. The control flow integrity mechanisms will prevent the injected code from being executed, since the code bit cannot be set programmatically.

In addition, the dispatcher is usually a modified return address that must have the callable bit set. The callable bit cannot be set programmatically.

**Return Oriented Programming** In an ROP attack, excerpts of existing code (so-called gadgets) are chained together into a malicious program. ROP attacks are, again, usually initiated from a memory out-of-bounds write action. The labeling concept renders cross-security domain overwrites useless. In addition, ROP attacks require the ability to jump to an attacker-controlled location. This can only be implemented with an indirect jump instruction (`jalr rd, rs`). The `rs` register should have the callable bit set. Since this bit can only be set by a `jal(r)` instruction or by the program loader, and is cleared on modification of the register, indirect jumps will only succeed to compile time defined addresses. On other addresses, the jump will trigger an exception and the ROP attack will thus be thwarted.

**Data Oriented Programming** In DOP key data (e.g., the size of a buffer) is overwritten and can be used to execute and chain data oriented gadgets in order to perform malicious computation. The Heartbleed bug described in Chapter 1 is an example of a DOP attack.

Under the restrictions of the labeling concept, cross-security domain data overwrites cannot be used to execute a DOP attack. By properly implementing different security domains for input data and internal data, the available set of data oriented gadgets and dispatchers can be greatly limited, if not eliminated completely.

**Modification of Information** Buffer overflows are not directly prevented by the labeling concept, however, buffer overflows cannot be used to modify information in another security domain. The label is always propagated with the data, When a buffer overflow of a buffer in one domain is used to overwrite another buffer in another domain, the label of that second domain is changed as well. Subsequent reads of that buffer will thus not succeed. Buffer overflows within the same security domain will still lead to information overwrites. By properly assigning security domains these buffer overwrites will escalate into catchable security domain violations.

## 5.9 Conclusions

The color labeling concept described in this chapter adds the notion of a security domain to data elements via a label in a tagged architecture. The security domain separation is maintained via a simple set of data

protection policies.

Additional special bits (code bit, callable bit) ensure control flow integrity. Using the tint system special functions that handle data from multiple security domains can be implemented, while maintaining control flow integrity.

The control flow integrity mechanisms impose some limitations on programs, but these limitations can either simply be overcome or avoided at all.

Theoretical evaluation of the concept for the weaknesses and attacks identified before show that the concept holds up the promise of keeping security domain separation.

# Chapter 6

# Implementation: Overview

*"If you think it's simple, then you have misunderstood the problem"*

*– Bjarne Stroustrup*

## 6.1 Introduction

With the labeling concept defined (see Chapter 5), it has to be implemented in software and hardware. Implementation involves adding color labeling support to the toolchain, creating a program to extract label data from the executable and generate a security report, and adding support in the Fremu instruction set simulator to test the setup. In addition, some notes and considerations for a possible hardware implementation in the Frenox core are formulated.

This chapter gives an overview of the implementation work, from the compiler to the simulator. The work on the toolchain is described in more detail in Chapter 7, the program for extracting label information is described in Chapter 8. The changes to the Fremu simulator are discussed in Chapter 9, while considerations for hardware design and estimates on performance and area impact are described in Chapter 10.

## 6.2 Requirements

Implementation of color labeling involves both hardware and software. A programmer should be able to annotate the source code, the toolchain should process these annotations and the policies should be hardware enforced. An overview of these roles and tasks is shown in Table 6.1.

The labeling concept is defined at the hardware level and support can thus be implemented into any program language. As stated in Section 4.4 this work targets the Rust programming language.

At the level of the programmer, the concept should allow the programmer to annotate certain variables with a color. Special functions should be annotated with a function color and the colors of both the input

**Table 6.1:** Overview of the roles and tasks in the implementation of color labeling

| | Source Level | Hardware Level |
|---|---|---|
| Programmer | Specification<br>Annotation | |
| Toolchain | Verification<br>Inference | Label generation |
| Hardware | | Enforcement |

parameters and the results should be specified. A mismatch in color use should already be reported during compilation, the hardware enforcement should only be needed in unforeseen situations, such as attacks on the system.

In order to keep the required trust in the compiler to a minimum, label information is extracted from the binary using a simple program dubbed *label-tool*. This tool will not only extract the label information, but generate a security report as well, reporting on colored variables and special functions. High assurance applications are always subjected to a security audit, the report can be a helpful document for identifying the most interesting parts of the program from a security perspective.

In order to test the concept, the Fremu simulator has to be modified in order to support color labeling. This includes implementation of the concept in existing instructions, adding the new color instructions, and adding the ability to load coloring information.

## 6.3 Component Implementation

Implementation of the concept involves work on many different components with a strong interdepence, this is shown in Figure 6.1.

The upper part of the figure, above the orange Linker block, shows the steps involving the compiler. The code of the actual Rust program (in Rust terminology this is called a binary crate) and accompanying or derived files are shown in green. The light purple blocks represent the *RISCV-RT* library crate and the grey blocks represent the *colorlabels* library crate.

After the linking phase, the resulting executable is processed with the *label-tool*, represented by the bright green boxes. The executable and the label data extracted by the label-tool are finally loaded in the *Fremu* simulator for simulation.

The color annotated code in the Rust program and the label-tool are the only components that have to be trusted. When the label-tool is trusted, its outputs are trusted. The programmer annotations have to be trusted as well, and can be compared to the findings in the security report generated by the label-tool. Since the label-tool processes the final result of the Rust toolchain, this toolchain is effectively removed from the trusted code base.

### 6.3.1 Rust Toolchain: Compiling and Linking

A Rust program is called a binary crate. In order to build such a binary crate for the RISC-V, a run-time system is required. This is provided via the `riscv-rt` library crate. This crate sets up the interrupt

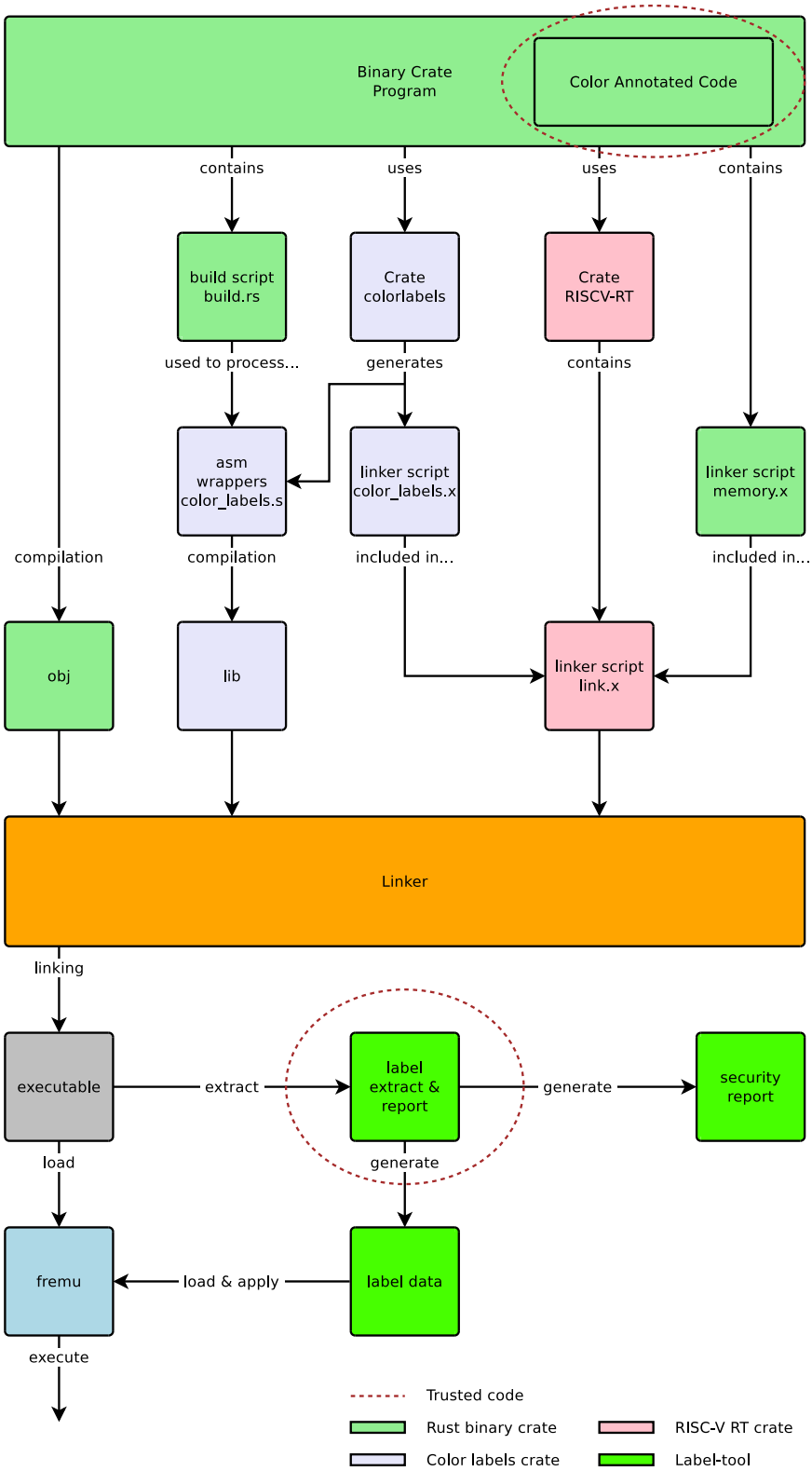



**Figure 6.1:** Overview of the implementation of the labeling system. The encircled parts are the trusted components.

handler, initializes the global variables, and transfers control to the main function of the binary crate. These low-level functions are closely relatated to the layout of the executable in memory. This layout is specified in the linker script `link.x`.

On many embedded systems memory is a scarce resource, the exact size of which depends on the hardware utilized. For this reason the available memory (both flash and RAM) is specified in the binary crate in the form of a linker script excerpt in the file `memory.x` that is included in the `link.x` linker script described earlier. A build script `build.rs` is used to tell the linker where to find the linker scripts.

The `colorlabels` crate adds support for colorlabels to the Rust binary crate. During compilation this crate can generate an assembly file `color_labels.s` and linker script excerpts in the file `color_labels.x`. The build script is used to compile the assembly file into a library. The `color_labels.x` file is included in the linker script of the `riscv-rt` crate.

The code of the binary crate is compiled into an object file. This object file is linked against the generated library file using the linker script of the `riscv-rt` crate. The resulting binary contains the executable code as well as the label data.

The source code of the binary crate contains trusted, color annotated parts and normal, unannotated code. The annotated code is part of the trusted components of the implementation and is used in conjunction with the label-tool (see below) and the generated security report in a security audit. Checking the annotations in the source against the annotation data found in the binary removes the Rust toolchain from the trusted code base.

### 6.3.2 Label-tool: Label Extraction and Reporting

The executable is processed by the label-tool to extract the label data and generate a data file containing this information. The tool also generates a security report that contains findings on special functions and the coloring of variables. The label-tool is part of the trusted components of the implementation: the tool is simple and separate program that processes the final result of the compiler. By splitting off this functionality from the compiler and processing the final resulting binary from the compiler, the tool

### 6.3.3 Fremu Simulator: Execution

Execution of the Rust binary crate is done in the Fremu simulator. The simulator implements the role of the hardware from Table 6.1. To this end, the simulator has been extended to support the labeling concept:

- Labels are added throughout the SoC: the memory, the bus, and peripherals;
- The policies from the concept are implemented;
- The new color instructions are added;
- A loader for the extracted color data had been implemented.

The simulator will load the program and the color data, initialize all peripherals, and load the label memory. Control is transferred to the executable and the loaded program will be executed.

# Chapter 7

# Implementation: Rust Toolchain

*"Real programmers can write assembly code in any language"*

*– Larry Wall*

## 7.1 Introduction

In order to easily use the protection mechanisms offered by the color labeling concept as a software programmer, the development toolchain needs some level of support of the concept. This includes the ability to define colors, annotate variables and functions, and have the compiler report errors and warnings on incorrect use of labeled data.

Implementing this functionality in the toolchain only enables the use of color labeling and enhances the user experience, no additional proctection is added by the compiler. Therefore, the compiler is not part of the trusted code base.

This chapter first describes the work on supporting the RISC-V architecture in Rust and subsequently explains the implementation of the color labeling support.

## 7.2 Supporting the RISC-V Architecture in Rust: `riscv-rt` crate

At the start of the implementation phase of this work, The RISC-V architecture was only supported in the unstable branch of the Rust toolchain. This support was limited to code generation only. The run-time system crate for the RISC-V, `riscv-rt`, was not in a usable state at that moment and did not even compile. In order to support the RISC-V a new `riscv-rt` crate has been implemented based on the existing `riscv-rt` crate, the run-time support crate for the ARM Cortex-M series (`cortex-m-rt`), and previous work done at Technolution [36].

The new crate sets up the trap handler, initializes the `.bss` (containing uninitialized global data) and

`.data` (containing initilized global data) sections and transfers control to the Rust main function. It is closely tied to the linker script `link.x` (see Figure 6.1). This linker script is used to communicate the locations of the `.bss` and `.data` segments to the run-time system startup code. The linker script also includes the `memory.x` script that has to be provided by the user binary crate. This script is used to specify the physical memory layout of the target RISC-V system.

In order to have a RISC-V program compile properly, the Rust compiler has to know where to find this file. The user binary crate includes a build script called `build.rs` that copies the linker script into a location that can be found by the compiler.

A number of compiler flags have to be provided, these are specified via the config file `.cargo/config`. Here, the default target is specified, the use of the compressed instruction set is disabled, and the main linker script is specified.

## 7.3 Supporting the RISC-V Architecture in Custom C and Assembly Code: `cc` crate

The `cc` crate is a standard crate that is used to compile external C or assembly files into a library that can later be linked to the Rust executable. The crate supports automatic cross-compilation based on settings of the Rust project. At the start of the implementation phase of the work the `cc` crate did not have support for the RISC-V architecture. This support has been added in order to use a standard crate for compiling external assembly files.

With the release of the 2018 version of Rust in December of 2018, the RISC-V architecture became an officially supported target and the missing support for the RISC-V in the `cc` crate has been reported as an issue with a proposed patch in April 2018. In August 2018 official support for the RISC-V architecture has been added to the `cc` crate.

## 7.4 Programmers' Interface and Requirements

The programmer should have a simple interface to define colors and annotate variables and functions. In Rust, annotations are conventionally implemented via attributes that take the form `#[this_is_an_annotation]`. Color definitions can be implemented using a macro or macro-like construct.

The code in Appendix E and repeated here in Listing 7.1 shows a simple example program that uses color labels. The program shows:

- how to define colors;
- how to annotate a global variable;
- how to annotate a special function.

Each of these steps will be explained in more detail below.

### 7.4.1 Defining Colors

Color definitions assign a color name to a color number. The number should be user-provided, since, e.g., hardware peripherals can have hardware defined color numbers.

**Listing 7.1:** Example Rust program with color labels

```
#![no_main]
#![no_std]

extern crate riscv_rt as rt;
extern crate panic_halt;

use rt::entry;
use core::ptr;
use colors_proc_macro::{color_label, new_color};

// Color definitions
new_color!(Black, 1);
new_color!(Red, 2);
new_color!(Green, 3);
new_color!(Blue, 4);

#[color_label(Red)]
static mut RED_DATA: u32 = 0;

#[color_label(fn = Blue, return = Red, args(encrypted=Black, key=Green))]
fn decrypt(encrypted: u32, key: u32) -> u32 {
    encrypted ^ key
}

#[entry]
fn main() -> ! {
    let encrypted: Black<u32> = Black(20);
    let key = unsafe{ ptr::read_volatile(0x60040000 as *const Green<u32>) };
    unsafe { RED_DATA = decrypt(encrypted, key); };

    loop { }
}
```

The programmer should have a very simple interface for defining colors. The macro `new_color!` is used for defining colors, see lines 12–15 of Listing 7.1.

### 7.4.2 Annotating Globals

Only global variables (`static`s in Rust) need to be annotated: locals are created in the run-color and can be overwritten with data in another color read from memory. The `color_label(color)` attribute is used to annotate a global variable with a color as shown in lines 17–18 of Listing 7.1.

### 7.4.3 Annotating Special Functions

Special functions require multiple annotations, some of which are optional:

- The function run-color;
- Parameter colors (optional);
- Result color (optional).

In order to unambiguously assign colors to parameters, run-color, and return value, are annotated as in lines 20–23 of Listing 7.1. The run-color is specified via the `fn` keyword, the color of the (optional) return value via the `return` keyword. Argument colors are grouped under the `args` keyword and are specified as `<parameter name> = <color>`, the order of the arguments is not important.

### 7.4.4 Maintainability Requirements

Since the labeling concept is targeting a niche market, it is expected that the modifications should be maintained in-house. The Rust toolchain has a release schedule of 6 weeks [37], with new features regularly being introduced and some unstable features being removed without prior warning. This makes maintaining complex modifications labour-intensive. This leads to the following additional requirements:

- The modifications to the toolchain should be as minimal as possible;
- The modifications should preferably use stable interfaces;

## 7.5 Extending the Rust Programming Language

In order to add color labeling support, the Rust language can be extended in multiple ways, each with their own possibilities, advantages, and disadvantages:

- Declarative macros;
- Procedural macros;
- Compiler Plugin;
- LLVM pass;

### 7.5.1 Declarative Macros

Macros in Rust are very different from the `#define`-style preprocessor macros of C and C++. These macros are essentially just string replacements that are processed by the preprocessor, before the code is processed by the compiler. Rust macros instead operate on the Abstract Syntax Tree (AST), a tree representation of the syntactic structure of the code [38]. The macros are expanded during the parsing phase using a special purpose macro parser. As such, unexpanded Rust macros conceptually occupy an AST node and have to expand into an AST node of identical type.

Rust implements two different types of macros: *declarative* (or `macro_rules!`) and *procedural* macros. The declarative macro is mainly used for general meta-programming and allows the programmer to replace certain pieces of code via a `match`-like expression. Declarative macros are used as part of the Rust language e.g., the `println!` function is in fact a declarative macro, as is `Vec!`.

Declarative macros would be useful to generate color specifications. However, they cannot be used to annotate variables or functions. On top of that, the current implementation of declarative macros will be deprecated in the future [39].

### 7.5.2 Procedural Macros

Procedural macros are the second kind of macros supported by Rust. These macros are stabilized since the release of the 2018 edition. A procedural macro takes as input parameter a `TokenStream` and has to output a (valid, but possibly modified) `TokenStream`. A `TokenStream` is the internal representation of (part of) the AST. In order to simplify using procedural macros, two standard crates are very useful:

**`syn`** is a crate that simplifies parsing the AST;

**`quote`** is a crate that can be used to easily generate a new `TokenStream`;

The combination of these two crates makes it much simpler to parse and modify the AST.

Procedural macros come in three distinct forms:

**Custom `#derives`** are used to automatically implement an item on a data structure. As such, custom derives can only be applied to type definitions;

**Attribute-like** macros provide generic annotations that can be applied to most parts of the language;

**Function-like** macros are the procedural macro equivalent of the declarative macros.

For adding color labeling support, both functions and variables need to be annotated with color information. Only attribute-like macros are suitable for this task. Defining colors can be done with a declarative macro and thus with a function-like procedural macro as well. Since it is not possible to define both declarative and procedural macros in a single crate, a function-like macro should be used.

### 7.5.3 Compiler Plugin

If macros cannot be used, a Rust compiler plugin is the next step to consider. The Rust compiler has a structure very different from most compilers and can be extended by creating plugins that are automatically invoked when needed. A compiler plugin has full access to all the internal data structures in the compiler and can thus be used to implement almost any functionality required.

The major drawback to this power, is that a compiler plugin is more complex than macros and that a compiler plugin uses internal, unstable interfaces.

### 7.5.4 LLVM Pass

The Rust compiler is build on top of the LLVM (Low Level Virtual Machine) compiler infrastructure [40]. The output of the Rust compiler is LLVM intermediate representation (IR) that is converted into native code by LLVM. LLVM works with different passes that can be enabled or disabled and process the IR. For very low-level changes, an LLVM pass can be created. LLVM passes can potentially perform any modification required, but have fewer ties to the original source code, potentially making it more difficult to generate appropriate error messages. Rust tries to use the latest stable version of LLVM, meaning that an LLVM pass would require regular updates to support new LLVM versions.

### 7.5.5 Selection

Only the two styles of macros use a stable interface, of these options, only the procedural macros can provide the kind of modifications required. With the Rust macros operating on the AST, it is likely that these macros are powerful enough to add support for color labels to Rust. The required changes can then be captured in a single, external crate, that only uses stable compiler interfaces, thereby enabling in-house maintenance of the code.

## 7.6 Conceptual Implementation

In order to fully support color labels, support at the semantic level (compile time verification, warnings and error messages on incorrect use) and the binary level (propagating the color information through the compiler and linker) is required. Since macros essentially just replace code with other code, first a manual implementation has been created, the details are later hidden behind macros.

### 7.6.1 Semantic Level

Rust has a very strong typing system, utilizing this typing system to support color labeling will automatically enable proper compile time verification and generation of warnings and error messages on incorrect use. Since the labeling mechanism is hardware supported, it is preferable that no additional run-time code or memory utilization is required for this implementation. Two Rust concepts can potentially be used:

- Define a colored `trait` that specifies a property of a type;
- Define a new type via a `tuple struct`;

Conceptually, a color or security domain specification is a property of a variable, similar, but orthogonal to the type of a variable. This rules out the use of a trait for the color of a variable, since traits specify a property of a type and not an individual variable. A trait `Red` defined on a type `u32` will have all variables of type `u32` implement the trait `Red`, making it impossible to distinguish two differently colored variables of the `u32` type.

**Listing 7.2:** Rust support for labeling variables

```
1  // Trait for color number
2  pub trait Color {
3      const COLOR: u16;
4  }
5
6  // Wrapper struct over generic type T
7  pub struct Red<T>(T);
8
9  // Associate color number with wrapper struct
10 impl<T> Color for Red<T> {
11     const COLOR: u16 = 2;
12 }
13
14 // Red global
15 static RED_DATA: Red<u32> = Red(0);
```

**Listing 7.3:** Rust support for labeling functions

```
1 fn decrypt(encrypted: Black<u32>, key: Green<u32>) -> Red<u32> {
2     let encrypted = encrypted.0;    // u32
3     let key = key.0;                // u32
4
5     Red(encrypted ^ key)
6 }
```

Rust also supports a newtype design pattern via a so-called tuple-struct. A tuple struct is a struct with anonymous members. Members can be accessed using their sequencial number in the struct. By implementing a tuple struct over a generic type, with that type as the only member, this newly defined type will act as a wrapper struct of the generic type. Tuple structs used as newtypes implement what in Rust is described as a zero-cost abstraction: there is no run-time overhead in terms of execution speed or required memory. It is, however, a distinct type during compilation. This means that assigning or using differently colored variables will generate a warning or error during compilation.

Listing 7.2 shows how this is implemented. The struct `Red` (in line 7) is the wrapper struct, generic over the type `T`. The trait `Color` (lines 2–4) is implemented for this new type `Red` (lines 10–12) and is used to associate the color number with the type. The global `RED_DATA` is defined to be of the type `Red<u32>`, a `u32` wrapped in the `Red` struct (line 15). The right-hand side of the initialization is here the value 0, wrapped in a `Red` struct. The Rust compiler can derive that the 0 should be of type `u32`.

Special functions that handle data from different security domains and run in a different color, can be implemented by unwrapping the function parameters, performing the required operations on the unwrapped data and then wrapping the results. This is shown in Listing 7.3.

### 7.6.2 Binary Level

Color labels should be extracted from the binary by an external program. This requires information to pass through the compiler into the resulting binary. This is accomplished by specifying information in special sections in the binary. Every specified color generates a small excerpt of linker script that defines

**Listing 7.4:** Linker script excerpt generated on color specifications of colors with the numbers 1–4

```
.colors.1 : ALIGN(4) { KEEP(*(.color.1)); } > FLASH
.colors.2 : ALIGN(4) { KEEP(*(.color.2)); } > FLASH
.colors.3 : ALIGN(4) { KEEP(*(.color.3)); } > FLASH
.colors.4 : ALIGN(4) { KEEP(*(.color.4)); } > FLASH
```

**Listing 7.5:** Static variable containing the address of a user-specified static variable, to be placed in the `.color.2` section

```
static RED_DATA: Red<u32> = Red(0);

#[link_section = ".color.2"]
static RED_DATA_ENTRY: &'static Red<u32> = &RED_DATA;
```

a section `.colors.`*`number`* that combines and keeps all `.color.`*`number`* sections in the object files. The *`number`* is the color number associated with the color. This shown in Listing 7.4.

Normal variables cannot be placed in such sections, since after the linking step they will have to be placed into either the `.data` (for variables) or `.rodata` (for read-only data) sections of the executable. However, these sections can be used to store the address of a colored static variable. Using Rust attributes, it is possible to specify these sections. A new static variable can be created that contains the address of the original static variable. This new variable is annotated with the `#[link_section = ".color.`*`number`*`"]` attribute, placing it in the appropriate section, see Listing 7.5.

The linker script excerpt (Listing 7.4) ensures that the variable `RED_DATA_ENTRY` that contains this address is stored in the resulting binary. An external tool can use these sections to extract the start addresses of colored variables. Via the symbol table the name and size of the referenced variable can be extracted. With this information the color data for the loader can be generated. This is explained in more detail in Chapter 8. After extracting the data, these sections can be stripped from the executable and will thus not enlarge the final binary.

## 7.7 Implementation with Macros: `colors_proc_macro` crate

As stated before, macros can be used to hide the implementation details of the concept from the programmer. Procedural macros replace part of the `Tokentree` with a new `Tokentree`, representing new code or variables. Listing 7.6 shows the macro-expanded version of the program in Listing 7.1 (see also Appendix E). Each of the expanded steps is explained in more detail below.

### 7.7.1 Defining Colors

Defining colors is implemented with a function-like procedural macro. At the semantic level, checks are in place to ensure:

- colors are not redefined;
- the color numbers are valid;

**Listing 7.6:** Macro-expanded version of the program in Listing 7.1

```
#![feature(prelude_import)]
#![no_std]
#![no_main]
#![no_std]
#[prelude_import]
use ::core::prelude::v1::*;
#[macro_use]
extern crate core as core;
#[macro_use]
extern crate compiler_builtins as compiler_builtins;
extern crate panic_halt;
extern crate riscv_rt as rt;
use colors_proc_macro::{color_label, new_color};
use core::ptr;
use rt::entry;
pub trait Color {
    const COLOR: u16;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Black<T>(T);
impl<T> Color for Black<T> {
    const COLOR: u16 = 1;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Red<T>(T);
impl<T> Color for Red<T> {
    const COLOR: u16 = 2;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Green<T>(T);
impl<T> Color for Green<T> {
    const COLOR: u16 = 3;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Blue<T>(T);
impl<T> Color for Blue<T> {
    const COLOR: u16 = 4;
}
static mut RED_DATA: Red<u32> = Red(0);
#[link_section = ".color.2"]
#[allow(dead_code)]
#[used]
static RED_DATA_ENTRY: &'static Red<u32> = unsafe { &RED_DATA };
#[inline(never)]
#[no_mangle]
extern "C" fn wrapped_decrypt(encrypted: u32, key: u32) -> u32 {
    encrypted ^ key
}
extern "C" {
    fn decrypt(encrypted: Black<u32>, key: Green<u32>) -> Red<u32>;
}
#[export_name = "main"]
pub fn s3702y3g24fw4s6f() -> ! {
    let encrypted: Black<u32> = Black(20);
    let key = unsafe { ptr::read_volatile(0x60040000 as *const Green<u32>) };
    unsafe {
        RED_DATA = decrypt(encrypted, key);
    };
    loop {}
}
```

The `(color, number)` pairs are stored in a `HashMap` to be able to lookup the color number and to make sure colors are only defined once. If the new definition is valid, the macro will generate a color tuple struct, similar to Listing 7.2. If this is the first color defined, the definition of the `Color` trait is generated as well. The four color definitions in lines 12–15 of Listing 7.1 are expanded into the code in lines 16–42 of Listing 7.6.

At the binary level, the macro emits a linker script excerpt similar to Listing 7.4.

### 7.7.2 Annotating Globals

Global variables (statics) are labeled via an annotation. The definition of a global is an `item` in the `TokenTree`. A `#[proc_macro_attribute]` annotated macro function is used to add the `colorlabel` attribute that can be used to annotate a global, see lines 17–18 of Listing 7.1.

At the semantic level, the attribute annotation is checked to make sure the color has been defined. The expanded macro will wrap the variable type in a color tuple struct. Any initial value will be wrapped as well.

At the binary level, a new static variable with the same name, appended with `_ENTRY` is generated and initialized with the address of the original variable. This new variable is annotated with the `#[link_section]` attribute to place it in the section with the appropriate color number.

Both the modified original static declaration as well as the new `_ENTRY` static declaration are both items in the `TokenTree`. The original item (static declaration) is replaced with the two items. The annotated global declaration is expanded into the code in lines 43–47 of Listing 7.6.

### 7.7.3 Annotating Special Functions

At the semantic level, the attribute annotation is checked to make sure:

- All colors are defined;
- Run-color is always specified;
- If the function has parameters:
  - All parameters are annotated;
  - Only valid parameter names are given;
- If the function has a return value, that it is annotated.

The color changes have to be implemented via the special instructions. The compiler and assembler do not have knowledge of these instructions and thus cannot generate the instructions from Rust code or even inline assembly code. Therefore, implementation of these colored functions is somewhat involved. An overview is given in Figure 7.1.

Color changes of the parameters and result of the function will happen only in the preamble and postamble of the actual function. Although Rust has support for functions without preamble (so-called naked functions), these are currently limited to functions without parameters. As a solution, the original function is wrapped in an assembly function, generated by the macro. The color changing instructions are generated as the binary representation of the instructions and included with a `.word` “instruction”. In order to properly pass parameters to the assembly function and to be able to call the original Rust function

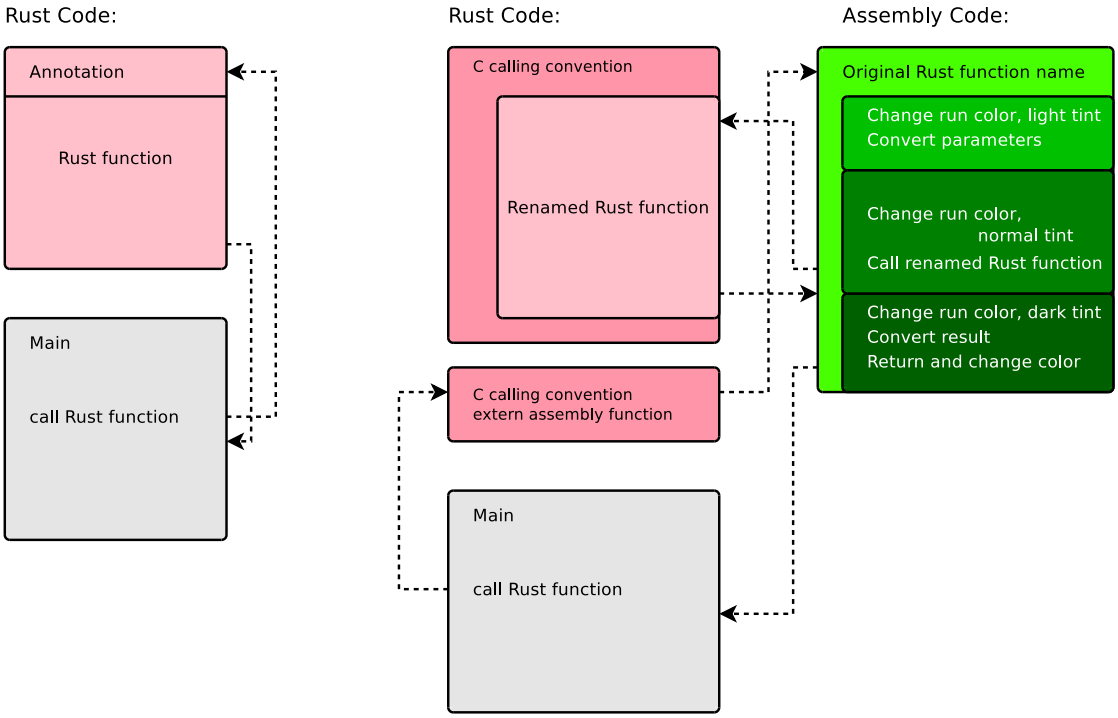



**Figure 7.1:** Overview of macro expansion of colored functions: on the left the Rust code with annotation, on the right the expanded result

from within the assembly function, the Foreign Function Interface (FFI) is used and both the assembly wrapper function and the original Rust function implement the C calling convention. The RISC-V ELF calling convention [41] is used to pass arguments between the Rust functions and the assembly functions.

This calling convention uses a different method of passing the argument, depending on the size of the argument:

- For arguments $\leq$ 32-bit, a single argument register is used;
- For arguments $> t$ 32-bit and $\leq$ 64-bit, two argument registers are used: a1:a0
- Arguments wider than 64-bit are passed by reference;

When no more argument registers available, arguments are passed on the stack, keeping data aligned to the minimum alignment of the stack and the data.

The macro renames the original Rust function by prepending its name with `wrapped_` and specifying C calling convention for this function. The original name is used for the generated assembly routine, which is declared in the Rust program as an external function with C calling convention and having all parameters wrapped in their appropriate color structs. By using the original name for the assembly routine, no further changes have to be made to the calling code.

As described before, wrapping variables in the color tuple structs just adds syntactic sugar: the structs do not add any additional data. This means that the assembly routine can access the wrapped variables by directly accessing the color struct itself: it is exactly the same data. The assembly routine implements the steps shown in Figure 5.5, which is also shown in the green block in Figure 7.1:

- Change into the run-color, light tint;
- Convert the parameters;
- Change to normal tint;
- Execute the body by calling the original, but renamed, Rust function;
- Change to dark tint;
- Convert the result of the original function;
- Return with color change to the Rust main code.

Since calling the original Rust function will overwrite the original return address stored in the return address register `ra`, a new stack frame is created where the `ra` register is stored. After the call to the original function has returned, the original value of `ra` is restored from this stack frame and the stack frame can be removed, see also the generated assembly file for the example program in Appendix E.4.

In order to properly convert the color of the parameters, the size of the parameter has to be known at compile time. For a single word, or a small number of words, the conversion can be unrolled, for larger data types, the conversion can be implemented in a loop. In that case the loop variables and temporaries are overwritten with a label in light or dark tint.

## 7.8 Control Flow Integrity Implementation

The macros can be used to implement the data protection mechanism, the control flow integrity mechanisms (code bit, callable bit) do not require manual annotation.

The code bit is used to denote that a certain memory word contains executable code or data. Code can only be executed, only data can be read. Writing data will always clear the code bit, thereby automatically disabling code injection attacks. The code bit thus implements the functionality of the well-known W⊕X, but is stricter in that code cannot be read and more fine-grained, working at the individual memory word instead of the usual page-level granularity.
During initialization, all the memory containing executable code will have the code bit set, all remaining memory will have the code bit cleared. Since it is not possible to set the code bit programmatically, no additional support in software is required to support the code bit.

The callable bit is used to denote that a data element contains a valid address for a jump or function call. The callable bit has to be set for return addresses and code pointers. Return addresses are created run-time via the `jal` and `jalr` instructions. These instruction will automatically set the callable bit in the destination register that will store the return address. Far jumps and far calls are generated with a combination of the `auipc` instruction and the `jalr` instruction, the `auipc` instruction will therefore also set the callable bit in the destination register.
Jump-tables and other compile time constant function pointers should have the callable bit set during loading. Rust generates jump-tables which are used when a function is called via dynamic dispatch. This is common in object oriented programming languages when implementing polymorphism. In Rust, dynamic dispatch only happens for so-called trait-objects [42]. A trait object is an abstraction over types that all implement a certain trait and implements a form of *polymorphism without classes*. This can be used to group different types together via their common trait, however, all these different types will implement the trait with different code. Trait-objects are implemented as *fat pointers* containing a pointer to the actual data and a virtual function table (vtable)[1]. The vtables are compile time constants and filled with function pointers to the implementation of the functions defined by the trait. This is shown in Figure 7.2.

Unfortunately, the vtables cannot be properly accessed at this moment from the AST. Under some circumstances it is possible to find the vtable pointer, but not the size of the vtable. This means that with the current implementation the callable bit cannot be properly set for vtables, which either disables the use of vtables and thus trait objects, or requires disabling the check on the callable bit for now.

Regular traits are implemented via a static dispatch and as such do not require indirect jumps or calls. This means that no callable bit is required in order to support the use of regular traits.

## 7.9 Evaluation

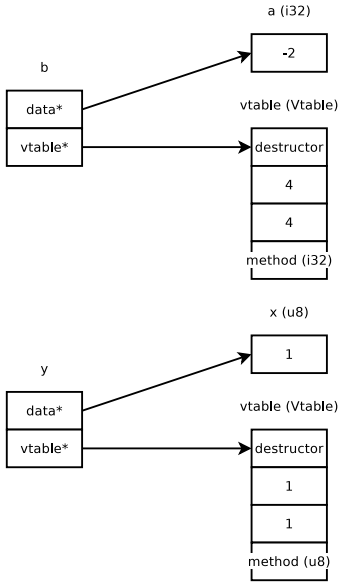
The choice for procedural macros fullfills the requirement for building color labeling support against stable interfaces. Since Rust macros operate at the AST level they are more powerful then the string replacement `#define` style macros from C/C++, however, there are still aspects of the language that are not (yet) exposed to procedural macros, some of which limit the possibilities of the current implementation. Other aspects cannot easily be implemented due to other reasons. Table 7.1 gives an overview of the support for various Rust features in color labeling, more detailed information can be found below.

**Defining color** works properly via the `new_color!()` macro. Currently, special functions have to be

[1] Note the difference with polymorphism in C++, in which the object itself contains a pointer to the vtable

```rust
fn main() {
    // Dynamic dispatch
    let a: i32 = -2;
    let x: u8 = 1;

    let b: &Foo = &a;
    let y: &Foo = &x;

    println!("{}", b.method()); // -1
    println!("{}", y.method()); // 3
}

trait Foo {
    fn method(&self) -> i32;
}

impl Foo for u8 {
    fn method(&self) -> i32 { *self as i32 + 2 }
}

impl Foo for i32 {
    fn method(&self) -> i32 { *self + 1 }
}
```

**(a)** Rust code with trait objects `b` and `y`




**(b)** Internal representation of trait objects

**Figure 7.2:** Rust code with trait objects and their internal representation

implemented in the same file as the color definitions. The color number has to be known by the macros that generate the assembly routines. If the definitions reside in another file, the data types can be forward-referenced and the linker will resolve the symbols. The assembly routines, however, are generated before the linking phase and thus require all information to be present during compilaton. If this restriction is deemed too limiting, one option could be to cache the processed color definitions in a file and, via the build script, make all sources depend on this file.

**Labeling variables** works properly. Global variables of all types can be color annotated. Color information is propagated through the compiler into the binary via special purpose linker sections.

**Labeling special functions** works properly, but with limitations. It is possible to label the function and specify the colors of the input parameters. The expanded macro replacement of the function utilizes the typing system of Rust to perform semantic checks on the input parameters.
On the binary level the implementation with the assembly wrapper function works properly, but there are some limitations:

**Cannot use more than 8 words of parameters** . The RISC-V ELF ABI specifies that parameters up to the first eight words are specified via the `a0..a7` registers. If more parameters are specified, these are passes via the stack. However, since the wrapper function requires building an additional stack frame, this requires modification of the stack frames. This has not been implemented due to timing constraints, but should not be to complex to add. If Rust will support a functionality like naked functions for functions with parameters, wrapping the call is no longer required. Instead the pre-amble and the post-amble of the function can be generated in assembly and the additional stack frame will no longer be neccessary.

**Only primitive datatypes can be used** . Conceptually the sizes of types of the parameters and the result are known at compile time. However, it was not possible to derive the size of the function parameters as constants. For now, only primitive data types are supported, the

**Table 7.1:** Overview of Rust language support with color labeling

| Functionality | Compiles? | Runs? | Remarks |
|---|---|---|---|
| `Box`, `Vec`, ... | ✗ | | Requires heap manager which is currently not available for Technolution's RISC-V ecosystem |
| Coloring globals | ✓ | ✓ | Any data type |
| Primitive type function params | ✓ | ✓ | |
| Non-primitive type function params | ✗ | | Only support for primitive data types |
| Generic type function params | ✗ | | Only support for primitive data types |
| Traits | ✓ | ✓ | Static dispatch |
| Trait objects | ✓ | ✗ | Dynamic dispatch, vtable entries not callable. Currently not used in the Technolution's Rust code base |
| Closures | ✓ | ✓ | Lambda functions |
| $>$ 8 params | ✗ | | Cannot currently use stack parameters. Only pass params via a0-a7 regs |
| Pass-by-ref params | ✗ | | Requires two color specs (pointer and pointed-to) |

sizes of which are stored in the crate. When the size of data types can be found via a `const` function, this limitation can be removed. Another option is to specify each parameter via a struct consisting of a pointer to the data and an integer with the size of the type.

**Trait objects** cannot be used under the current implementation, since the vtable cannot be identified properly with the macro code. This might be the most difficult problem to solve, although not allowing trait objects to be used at all for high assurance applications might be considered a reasonable restriction. It should be noted that currently no Rust products developed by Technolution use trait objects, which further supports the idea that restricting the use of trait objects does not impact the usefulness of the language.

If trait objects are required, then either more support from Rust macros is required or another option, such as a compiler plugin or an LLVM plugin should be considered. It might be possible to combine this with the macro solution. Detection of the vtables in the binary via an external tool such as the label-tool might also be a viable option.

## 7.10 Conclusions

Support for the RISC-V has been (re)-implemented as well as added to the `cc` crate for compiling custom C and assembly code. The Rust toolchain has been updated to support the color labeling concept. Using macros provides a stable interface to add this new functionality, which makes in-house maintenance much simpler. By utilizing the type system of Rust, a robust error checking mechanism for color label use is automatically provided by the compiler. The programmer can define new colors and, using the Rust attributes, both variables and special functions can be annotated.

Color information on global variables is propagated through the toolchain via special linker sections and can be extracted by an external tool.

Special functions are supported via a generated assembly wrapper that handles the color and tint changes of the function color, the parameter colors, and the return value colors. For now, only primitive data types can be used as parameters of a special function, there are other ways to work around this limitation. Future support in the language may also enable the use of other data types.

The control flow integrity mechanism of the code bit does not require any additional work from the programmer. The callable bit that is used to denote valid addresses is currently only partially supported: vtables cannot be automatically annotated, which precludes the use of trait objects. However, as this functionality is not currently used in Technolution's code base, this is not considered to be a prohibitive restriction.

# Chapter 8

# Implementation: Label-tool

*"Secrecy and security aren't the same, even though it may seem that way. Only bad security relies on secrecy; good security works even if all the details of it are public."*

*– Bruce Schneier*

## 8.1 Introduction

This chapter describes the label-tool that is used to extract label data from a compiled binary and generates a report for use in security audits. By implementing the label-tool as a simple, separate program, the compiler can be largely removed from the trusted code base.

## 8.2 Requirements

The label-tool complements the color labeling support in the Rust toolchain. The modifications to the Rust toolchain in order to support color labeling have been kept to a minimum. The label-tool processes the resulting binary, making sure all compiler and linker transformations and optimizations have been applied already. The label-tool should be able to:

- Generate coloring data for use in the simulator or the program loader;
- Generate a security report, highlighting colored variables and special functions.

In order to achieve these goals the label-tool should extract this data from the binary generated by the Rust toolchain. This adds the label-tool to the trusted code base, but largely removes the Rust compiler from the trusted code base. In order to trust the label-tool, the following additional requirement is added:

- The implementation of the program should be as minimal and simple as possible;

## 8.3 Extracting Coloring Information from Binary

The result of the compilation and linking steps is a binary executable. This binary could contain one or more `.colors.<number>` sections. These sections are used to transport the color information from the source through the compiler into the binary. Given the example program of Appendix E, the `objdump` program can show the presence of this section:

```
$ objdump -x example_program

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000248  80000000  80000000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000000  80000248  80000248  00001248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .data         00000000  84000000  80000248  00002000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .bss          00000004  84000000  80000248  00002000  2**2
                  ALLOC
  4 .colors.2     00000004  80000248  7c000490  00002248  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .riscv.attributes 00000024  00000000  00000000  0000224c  2**0
                  CONTENTS, READONLY
  6 .comment      00000012  00000000  00000000  00002270  2**0
                  CONTENTS, READONLY
```

The `.colors.2` section contains the variable `RED_DATA_ENTRY` that in turn contains the address of the `RED_DATA` static (see Appendix E). This can also be verified using `objdump`:

```
$ objdump -s -j .colors.2 example_program

Contents of section .colors.2:
 80000248 00000084                             ....
```

Note that the data is in little endian format, re-ordering the bytes gives the actual address: 0x84000000.

Via the symbol table the original name and size of the static can be derived, this is schematically depicted in Figure 8.1. The demangled symbol table can be printed with `objdump` and `rustfilt`[1]:

```
$ objdump -t example_program | rustfilt

SYMBOL TABLE:
00000000 l    df *ABS*      00000000 colored_function_simple.c5e8k5gn-cgu.0
80000248 l     O .colors.2  00000004 colored_function_simple::RED_DATA_ENTRY
84000000 l     O .bss       00000004 colored_function_simple::RED_DATA
00000000 l    df *ABS*      00000000 riscv_rt.9358rnn3-cgu.0
800000dc g       .text      00000000 decrypt
800000ac g     F .text      00000030 main
80000088 g     F .text      00000024 wrapped_decrypt
....
```

As can be seen, the address 0x84000000 points to the variable `RED_DATA` that has a size of 4 bytes.

The label-tool parses the data in the executable in a similar way and stores the results in a new file that can be used by the loader. For now a very simple data format is used, schematically depicted in Figure 8.2.

[1] `https://github.com/luser/rustfilt/`

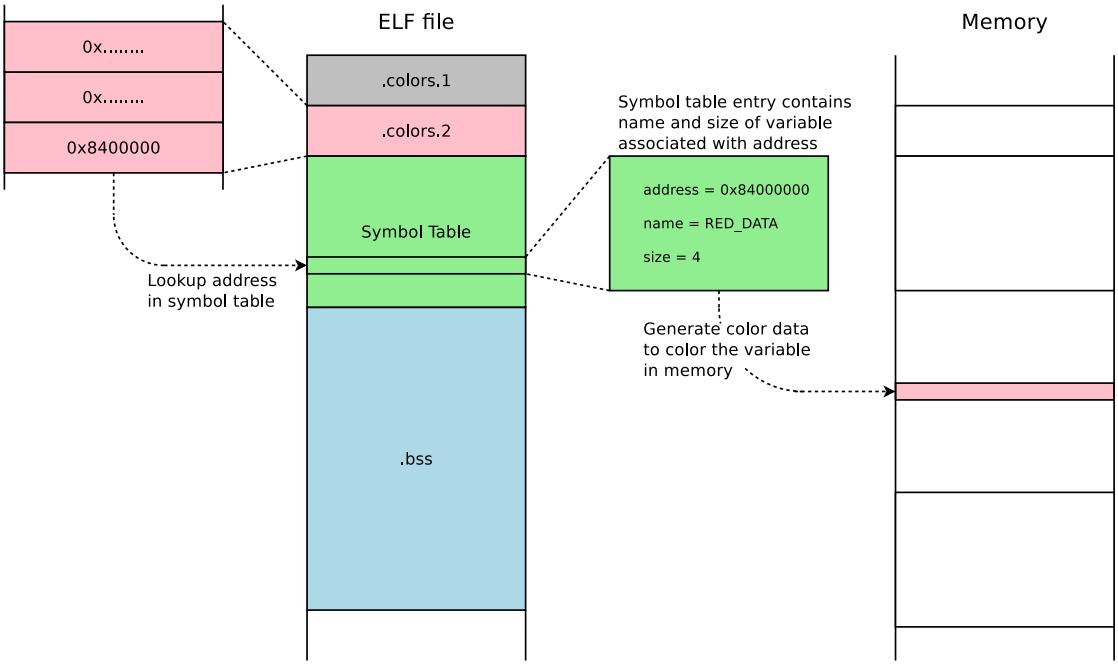



**Figure 8.1:** Reading the `.colors.xx` sections and symbol table to generate the color information.

Data is stored in records, ordered by color number. Each record contains a header consisting of the color number and the number of entries *n*. The header is followed by *n* entries, each consisting of an address and the size of the data type residing at that address. Before being used in a production environment the file format should be more resistant against tampering, options include signing the file and encrypting the data.

During parsing, the program reports the color sections and variables found. Variable names are not demangled, instead this is left to the `rustfilt` program.

## 8.4 Reporting on Special Functions

Besides reporting the colored variables, the label-tool will also check the program for special functions and report on the use of these functions. To this end the program reads in the `.text` section of the program and uses the symbol table to search for functions in the `.text` section. These functions should all have a similar structure:

- Start with the `chg_color_light` instruction;
- Optionally, function parameters are converted;
- Switch to normal mode. The `chg_color_normal` instruction is used to switch to normal tint;
- Then the body of the function is called;
- Following the function call should be a `chg_color_dark` instruction;

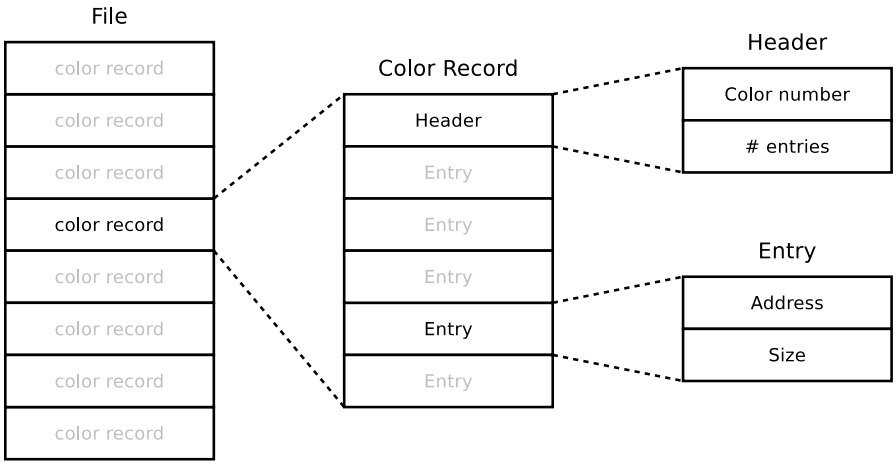



**Figure 8.2:** Data format of the output of the label tool

The label-tool checks for any `chg_color_??` instructions and warns on any dangling `chg_color_normal` and `chg_color_dark` instructions. A `chg_color_light` instruction is assumed to be the start of a special function. After locating such an instruction, the normal structure of a special function is checked and a warning is issued upon deviation of that pattern.

## 8.5 Conclusions

The label-tool is a simple program that will process the executable generated by the Rust toolchain. The tool outputs a simple data file for use in the loader of the simulator that contains the color information of the colored variables. The file format is deliberately very simple for now but will need to be more tamper resistant for production use.

The tool also produces a security report that contains information on the variable coloring and any special functions found. The security report can be used in a security audit.

By having this functionality implemented in an external, simple, and in-house developed tool, the compiler can largely be removed from the trusted code base.

# Chapter 9

# Implementation: Fremu Simulator

*"Simplicity is great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."*

*– Edsger Dijkstra*

## 9.1 Introduction

In order to test the concept the in-house developed simulator of the Frenox SOC, called *Fremu* has been modified to support labeling. This chapter describes the steps taken to modify the simulator.

## 9.2 Basic Structure of the Fremu Simulator

Fremu is the in-house developed simulator of the 32-bit Frenox RISC-V implementation by Technolution. The simulator currently supports the I (integer), M (integer multiplication and division), and A (atomic memory operations) instruction sets, as defined in [12]. The simulator is build around a 32-bit bus (modeling the AXI4-lite bus of the hardware implementation) to which the CPU, the memory, and several peripherals (uart, timer, ethernet controller) are connected. This is schematically shown in Figure 9.1.

During initialization, devices are added to the bus, based on a Device Tree description [43]. Devices are memory mapped. The program and data memories are added as well, after which the ELF program is loaded [44]. The core is then initialized on the start address specified in the ELF file and the program is executed until an endless loop is detected.

The CPU is implemented as an instruction level simulator of the RISC-V. Instructions are decoded and executed on the fly. The simulator can be configured to emit trace messages during program execution, including register dumps, instruction decoding, etc.

Reads from and writes to addresses are dispatched to the bus, which checks to see which device maps

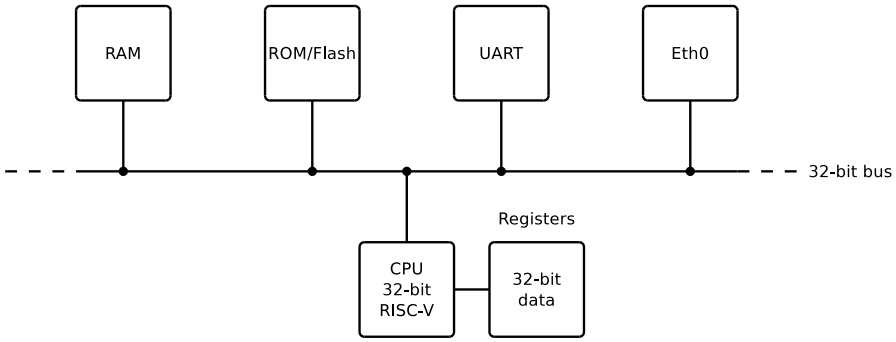



**Figure 9.1:** Schematic overview of the Fremu simulator

the given address. The request is then handled by the appropriate function (if available) provided by the device. This includes actual memory (program memory and data memory) access. This approach allows the simulation models of the peripherals to be designed independently of the simulator.

## 9.3 Basic Labeling Support

Fremu, being a simulator for the 32-bit Frenox core, uses the `uint32_t`/`int32_t` types as its basic datatype. This type is used as data type for the registers, for the bus, and the memory. In order to support labeling, two new basic data types, the `u32_labeled_t` and `i32_labeled_t` types are introduced, see Listing 9.1. These new data types are used in the RISC-V core, as well as in the memory and the bus. The core has an additional internal state that tracks the current color and tint.

### 9.3.1 Debug Output with Labeling Support

Color data has been added to the debug output of the simulator. The current color, tint, code bit and callable bit of the registers and the core is shown.

### 9.3.2 Adding Labeling Support to Existing Instructions

In order to add labeling support to existing instructions, the instructions were first grouped based on the decoding of the opcode field. This allowed for easily adding labeling support to certain classes of instructions. Since the labeling concept checks data labels on register usage, the labels of the decoded `rs1` and `rs2` (if applicable) registers are checked against the color and tint of the core. Implementation of color labeling for existing instructions has thus been isolated to two different checks:

- Can the value in the register be read (`can_read_reg`);
- Can the value in the register be used in an indirect call (`can_call_reg`);

The flowchart for accessing a register is shown in Figure 9.2. If the register(s) can be accessed, the instruction can perform the intended operation and write back the result with the color and tint of the core. In case of a `jalr` or an `auipc` instruction, the callable bit of the result register is set.

**Listing 9.1:** Fremu labeling data types

```
typedef struct i32_labeled i32_labeled_t;
typedef struct u32_labeled u32_labeled_t;

// - 32-bit label per 32-bit word
// - Several bits are reserved for special functions
union colorlabel_t {
    uint32_t raw;
    struct {
        unsigned bit_callable:1;         // Valid address
        unsigned bit_code:1;             // Code/Data
        unsigned tint:2;                 // Light, normal, dark
        unsigned color:COLOR_FIELD_SIZE; // Actual color
    } fields;
};

// Labeled types
struct u32_labeled {
    uint32_t data;
    union colorlabel_t label;
};

struct i32_labeled {
    int32_t data;
    union colorlabel_t label;
};
```

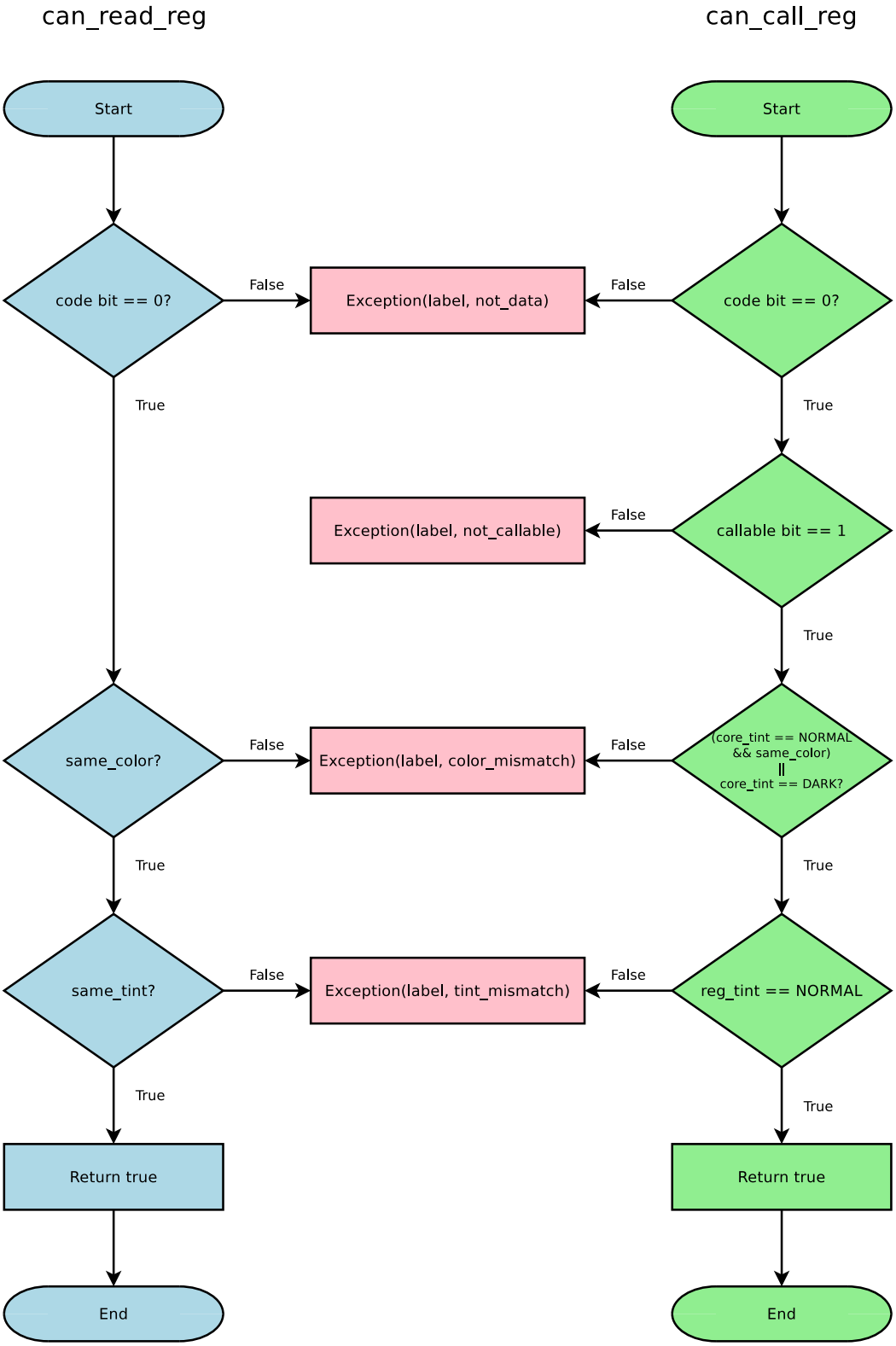



**Figure 9.2:** Flowchart for checking if register contents can be read (left) or used as address (right)

The `jal` and `jalr` instructions have a different behaviour in light or dark tint, see Table 5.1. This is directly implemented in the simulation code of those instructions.

The callable bit is cleared when the value of the data in a register is modified. The only allowed operation that keeps the callable bit, besides a load or store, is a move operation. On RISC-V a move operation is implemented as `addi rd, rs, 0` [12]. If this specific instruction is executed, the value of the callable bit in the `rs` register is copied into the label of the `rd` register.

### 9.3.3 Adding Labeling Support to Peripherals

Peripherals are not modified to support the labeling mechanism, but are instead connected to the bus via a color bridge device. This color bridge device adds a color label to data coming from the connected device and checks the label of data send to the device, closely resembling the intended actual hardware implementation. This schematically shown in Figure 9.3. The use of the color bridge makes it possible to use existing, verified peripheral implementations for both a system with and without labeling support.

In order to add labeled peripherals, the device tree parser is extended to support an optional field specifying the color of the peripheral. During initialization a color bridge in the specified color is placed on the bus and the actual device is connected to that color bridge. The read and write actions from and to the device are routed through the color bridge.

On a read from the device the color bridge will add a label to data. The label color is the color of the bridge, the tint is set to normal.

On a write to the device the color bridge will verify if the color of the data matches the color of the bridge. If those are identical, the write is forwarded to the device, otherwise a bus error is generated.

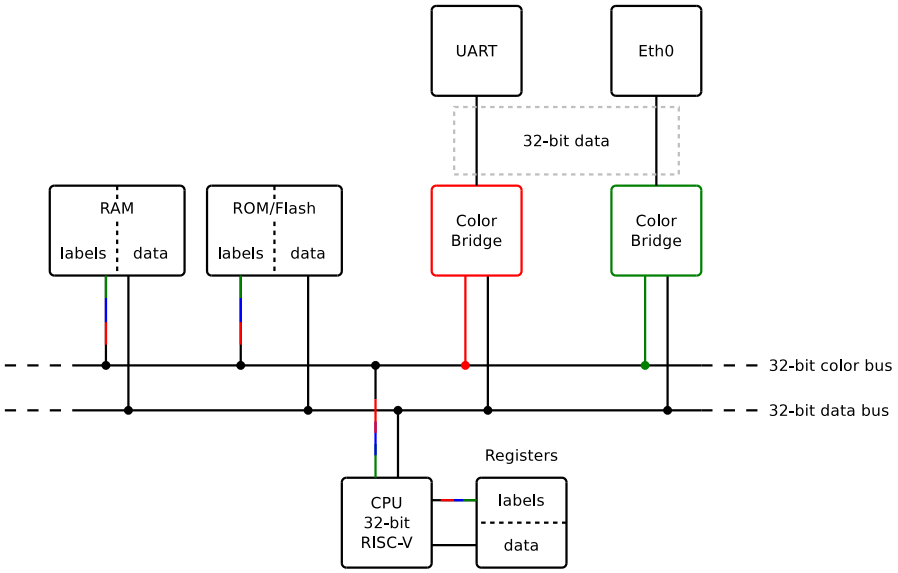



**Figure 9.3:** Schematic overview of the Fremu simulator with support for the coloring concept

### 9.3.4 Adding New Labeling Instructions

The five new instructions (see appendix D) that can be used to change the processor's color and tint and the color of registers can be added to the simulator in a way similar to existing instructions. The processor tint determines whether or not the instructions can be executed. This is checked in the simulation code of the instructions.

The instructions `reg_from_color` and `reg_to_color` that can change the color of a register are able to process a register with a color different from the processor color. The simulator implementation of these instructions does not use the `can_read_reg` and `can_call_reg` functions.

## 9.4 Reading in Label Information

Initially all label memory is cleared (color number 0) and the processor is initialized in run-color 1. This makes sure that buffer out-of-bound reads into uninitialized memory will escalate into security breaches. The data file generated by the label-tool is read in and used to set the color of the specified memory locations.

## 9.5 Using Labeling Conditionally

Most of the labeling code is gated by define blocks. This makes it possible to use a single codebase for use with and without labeling support. In order to minimize the performance overhead caused by the labeling code the `can_read_reg` and `can_call_reg` functions simply return `true` when labeling support is turned off. By compiling the code with compiler optimizations enabled, these functions will be optimized away and not slow down the simulator.

## 9.6 Evaluation

Simple assembly programs have been designed to test each color instruction. Using the debug output with color data proper functioning of both the exising and new instructions under the color labeling concept has been determined.

With the color labeling support functioning properly, the simulator has functioned as the test platform for the labeling concept. The simulator has been used to verify the concept and derive the run-data in Table 7.1. The simulator has also been used to perform functional testing of the software implementation.

## 9.7 Conclusions

The Fremu simulator is extended with support for the labeling concept. Support has been added by replacing the basic data type with a more complex struct that contains the label and the associated data. Checks have been added that implement the policies of color labeling and the new instructions have been added. A loader for the color data has been added that is used to initialize the label memory.

# Chapter 10

# Considerations for Hardware Implementation of Color Labels on Frenox SoC

*"I need something to compare this to. Could I please have a microsecond?"*

*– Grace Hopper*

## 10.1 Introduction

This chapter describes how the color label concept could be implemented in the Frenox RISC-V SoC. First, an overview of the design of Frenox is given, then some points that might require special attention during implementation are highlighted. Finally, an estimation of the area requirements of color labeling is given.

## 10.2 Description of the Frenox SoC

The Frenox RISC-V core is a VHDL implementation of the RISC-V and as such, highly configurable. Frenox supports the "G" instruction set, with the exception of the "C" set of compressed instructions, as defined in [12], but it is possible to remove support for the "A", "M", and "S" subsets of instructions. The Frenox core is an in-order processor with a traditional five-stage RISC pipeline and register forwarding.

The SoC can be implemented with on-FPGA memory or external DRAM. Optionally, an MMU (Memory Management Unit) can be instantiated, which enables running the Linux operating system on the core. The bare-metal platform initially targeted by this work does not require the presence of the MMU. Hardware peripherals are connected via the AXI4-lite bus. Standard peripherals for different communication protocols (e.g., serial, ethernet), offline storage (flash memory), etc, can be instantiated.

## 10.3 Implementing Tag Memory

The tag memory can be implemented in different ways. One option is to modify the memory controller such that only even addresses are used for labels and odd addresses for data. By adjusting the addresses send to the memory controller, this can automatically hide the tag memory from the user program. Another option would be to physically separate the tag memory from the data memory by adding a separate memory module and memory controller and bus for the tag memory alone. This option is likely the fastest with the two memory controllers working in parallel. It is, however, also an expensive choice in terms of hardware resources since the memory controller is instantiated twice.

## 10.4 Checking the Labels

Labels should be checked before the actually using the associated data in execution of the instructions. Checking the labels can be done in any pipeline stage up to and including the execution stage. In light of the Spectre [24] and Meltdown [25] vulnerabilities, care has to be taken that no internal state of the processor can be leaked via a side-channel. Although Frenox does not support speculative execution, the registers between the pipeline stages do possibly contain values that should be inaccessible according to the current execution security domain. Based on these observations, it might be prudent to place the label checks in the instruction decode stage. However, since the failure mode on a security violation is a system halt or reboot, this should not be a problem (provided a proper initialization of said registers is performed after reset). When the system is halted, no more information can be extracted and the only information gained, is that the labels do not match, which already was known information according to the adversary model (see Section 2.2).

By instead delaying the label check to the execution stage the memory access path is not modified and thus no additional delay will be introduced. Checking the labels in the execution stage only accesses data in the register file and should be fast enough to prevent introducing any additional delays. Checking the label in the execution stage has the additional benefit of simplifying the use of register forwarding: by forwarding the label together with the data, the label checks can't be by-passed, as might happen when forwarding the data only.

## 10.5 Loader Program

The loader program should be run with special access rights: it should be able to directly write values in the label memory. This can be implemented as a special mode that can only be entered after a system reset. Once the special functionality is no longer required, the processor should enter a normal execution mode and not be able to switch back to the special mode. Alternatively, loading the executable and initializing the label memory could be done via a special processor that will only be active after a reset and can be disabled during normal operation. A completely different approach could be to generate code that will properly color variables.

## 10.6 Peripherals

Peripherals don't have to be modified in order to support the labeling concept. Instead, a color bridge device is introduced that will be placed between the bus and the actual peripheral. The color bridge

device will be be instantiated in a certain color. On a read of the device, the color bridge will append a color label in its own color. On a write action to the device, the color bridge will check the incoming label against its own color. A mismatch will generate a bus error.

This approach will enable integral re-use of existing and tested peripherals with color labeling. A similar approach has been used to support colored peripherals in the Fremu simulator (see Chapter 9, specifically Figure 9.3).

## 10.7 Estimated Resource Utilization

Color labeling will increase the FPGA resource utilization use of the Frenox SoC, mostly because label memory will be added and thus an additional memory controller, an additional bus, and enlarged caches are required. The resource usage of the Frenox reference implementation on an Intel Cyclone V FPGA 5CGXFC5C6F27C7 is determined via the Quartus software. Table 10.1 shows an overview of the FPGA resource utilization, grouped by function, in terms of Adaptive Logic Modules (ALMs) and 10 Kbit memory blocks (M10Ks).

The expected increase in FPGA resource utilization is as follows:

- An additional memory controller will be instantiated;
- Data and instruction cache memory will double in size, control will increase with 20%;
- The I/O and memory busses will double in size;
- The Frenox core will increase with 20% to account for the label checks and the new instructions;
- Peripheral logic will increase with 20% to account for the color bridge devices;

Besides the resources on the FPGA, the external memory will also double in size.

As can be seen from the table, the total amount of ALMs is estimated to increase with 14%, the amount of memory blocks required is expected to almost double.

## 10.8 Conclusions

Although no actual hardware implementation has been performed, some considerations for a hardware implementation are given:

- The tagged architecture can be implemented in several different ways. The fastest solution, but also most expensive implementation in terms of resources, is adding an extra memory controller just for the memory labels.
- Label checking is best implemented in the execution pipeline stage. Doing so will minimize the performance impact and make register forwarding simpler and more effective. Care has to be taken to make sure no intermediate information can be leaked on a security violation: the system must either halt or reset.
- The loader program should run with elevated access rights that can only be entered following a reset. The loader should be able to write to the label memory. After initialisation, the elevated rights should be dropped.

**Table 10.1:** Overview of approximate FPGA resource utilization of reference implementation of the Frenox SoC and estimated resource utilization with implementation of color labeling

| | Reference Utilization | | Estimated Utilization | | | |
|---|---|---|---|---|---|---|
| Component | ALMs | M10Ks | ALMs | | M10Ks | |
| Memory Controller | 1700 | 18 | 3400 | (+100%) | 36 | (+100%) |
| I-cache | 430 | 60 | 520 | (+20%) | 144 | (+100%) |
| D-cache | 610 | 65 | 730 | (+20%) | 128 | (+100%) |
| I/O Bus | 980 | 0 | 1960 | (+100%) | 0 | (+0%) |
| Memory Bus | 150 | 2 | 300 | (+100%) | 4 | (+100%) |
| Core | 2000 | 0 | 2400 | (+20%) | 0 | (+0%) |
| Peripherals | 1500 | 14 | 1800 | (+20%) | 14 | (+0%) |
| Other | 3130 | 21 | 3130 | (+0%) | 21 | (+0%) |
| Sum | 10500 | 180 | 11980 | (+14%) | 347 | (+93%) |

- Peripherals can be connected by means of a color bridge, no color support in the peripheral is required.

Informed estimates on the resource utilization of implementing color labeling are given: the amount of ALMs and memory blocks are expected to increase by 14% and 74%, respectively. The external memory will also double in size.

# Chapter 11

# Results

## 11.1 Introduction

With the complete toolchain and a test platform in the form of the simulator available, a number of tests were performed in order to determine how the concept and implementation performs functionally. The costs in the form of overhead in performance and area have been determined.

## 11.2 Functional

The main goal of the concept is to prevent cross-domain information leakage. A number of test programs were written that exhibited incorrect behaviour. Since Rust is designed to prevent the compilation of code with buffer overflows, some C-routines were used to force the buffer overflow.

The implementation has been evaluated by testing against the attack patterns described in Chapter 2:

**Information Disclosure** A buffer over-read in another security domain can lead to information disclosure. Using color labeling support and proper labeling of the variables, the Rust compiler automatically detects a buffer over-read and will report this issue as an error during compilation. Using external C routines, a buffer over-read can still be triggered. Without color labeling support, this buffer over-read could read out data from another security domain. With color labeling enabled, this over-read was intercepted by the hardware and information disclosure was thus prevented.

**Code Injection** The adversary model assumes an attacker with full knowledge of the memory layout. An attacker can exploit this knowledge and overwrite existing code memory or inject code on the stack. Since code injection attacks usually originate from buffer overwrites, it not easily possible to implement such an attack in Rust and again C routines were used instead.

Color labeling implements the code bit to denote that a certain memory element contains code, this bit cannot be set at run time. Using the C routing, The injection itself is successful. However, when encoutering the injected code, the simulator will not execute the instruction since the code bit is cleared.

**Return Oriented Programming** ROP is centered around the possibility of an attacker diverting the control flow to an attacker-controlled location. On the RISC-V only the `jalr` instruction can be used

for this purpose, since all other instructions have the displacement encoded as an immediate. Traditional ROP modifies the return address as stored on the stack to divert the program flow. Using C routines to implement this attack the return address could be modified, however, in doing so the `callable` bit is cleared as well. When executing a `jalr` instruction with this modified address in the jump-register, the simulator will halt due to the cleared callable bit.

**Data Oriented Programming** DOP attacks do not modify the program flow. Instead they are based on modifying key data in the application, initially via a buffer overflow. Color labeling can prevent DOP attacks by proper application of colors. The buffer overflow will then overwrite the key data with a different color label. Subsequent access of that data will cause an exception and thus stop the DOP attack. This is similar to information disclosure due to a buffer over-read and therefore has not been tested separately.

**Modification of Information** Modification of information is not impossible under color labeling. However, since the labels are propagated with memory writes, it is not possible to change data in another security domain. The data can be overwritten, but on a read, the hardware will detect a color mismatch. This is similar to information disclosure due to a buffer over-read and therefore has not been tested separately.

In addition to test programs that intentionally contained errors that would trigger an exception under the color labeling implementation, correct test programs were written to test if the system would exhibit unintentional exceptions. For all implemented features no such unintentional exceptions, both during compilation and exectution, were detected.

Since not all features of the Rust programming language have been implemented in the current implementation yet (see Section 7.9 for an overview of the limitations), these test programs can also be used for testing while implementing these features.

## 11.3 Performance

Color labeling impacts the performance of a program:

- Converting the colors of parameters and result of special functions requires additional code;
- Wrapping the original function call requires additional instructions;
- Checking the labels may increase the execution time of all instructions;

Although high performance has not been a major design goal for color labeling, some effort has been put into minimizing the performance impact. Since no actual hardware implementation has been performed, no data on the performance impact of the concept on the performance of the instructions is available. As described in Chapter 10, the performance impact of checking the labels on execution is expected to be negligible. Therefore, only converting the parameters and wrapping the original function call will impact the performance.

### 11.3.1 Setup

In order to determine the performance impact, a typical use-case has been investigated and the performance impact caused by the concept has been calculated. Color labeling targets high assurance applications, one such application is OpenVPN-NL [45], a modified version of the OpenVPN [46] open source

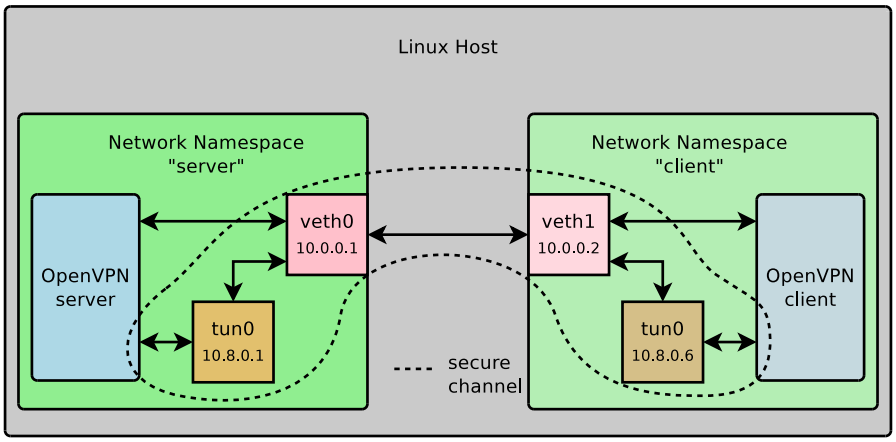



**Figure 11.1:** Test setup for OpenVPN-NL profiling

VPN software endorsed by the Dutch government. OpenVPN-NL can be used to set up a secured connection between different computers or networks. Initially the RSA protocol is used to communicate a symmetric key that is used to encrypt any further communications. The default cipher used by OpenVPN-NL is AES-256-GCM which is implemented in an external TLS library. OpenVPN can use the OpenSSL [47] or mbed TLS [48] libraries, however, in OpenVPN-NL only mbed TLS is allowed.

The OpenVPN-NL configuation files have been modified to add profiling information during compilation in order to find out which functions are called most often and which functions consume most of the execution time.

Using Linux network namespaces, different applications using network related system resources can be isolated from each other. Two such namespaces have been created: `server` and `client`. A virtual ethernet pair has been created and each end has been placed in a separate network namespace. The OpenVPN-NL server is started in the `server` namespace while the OpenVPN-NL client runs in the `client` namespace. The OpenVPN-NL server creates a TUN device that is used for the secure channel. This TUN device is assigned its own IP address and traffic to that IP is routed to the virtual ethernet device. When the client connects to the server, a TUN device is created at the client side as well. Routing information is pushed from the server to the client such that the client can connect directly to the IP of the TUN device on the server. This setup is schematically shown in Figure 11.1.

Using the `netcat` tool to send data from the client to the IP of the TUN device on the server side, 1 GB of data has been transported over the VPN tunnel. By sending this large amount of data, the most-often called functions can be extracted. Now the overhead introduced by the color labeling concept can be calculated.

The setup has been implemented on a PC platform (see Appendix G). This means that the execution time data gathered from the profiling data should be used with caution: there are many differences in architecture, instruction set, and processor frequency between the x86-64 and the RISC-V ISA that make it difficult to draw any meaningful conclusions on the corresponding run time for the RISC-V. The number of function calls, however, will be architecture independent (given the same set of optimizations) and only depend on the input data. Since the overhead caused by the color labeling concept is a per-function overhead, calculating this overhead for the most often called function should give an accurate prediction of the expected total overhead in terms of added instructions.

The results of the profiling can be found in Appendix H. In total, over $4.3 \cdot 10^9$ function calls have

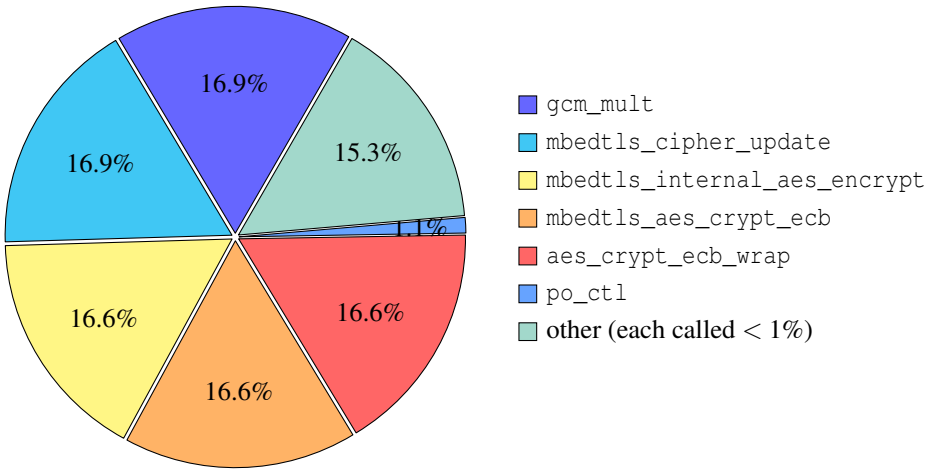



**Figure 11.2:** Distribution of number of function calls in profiling results

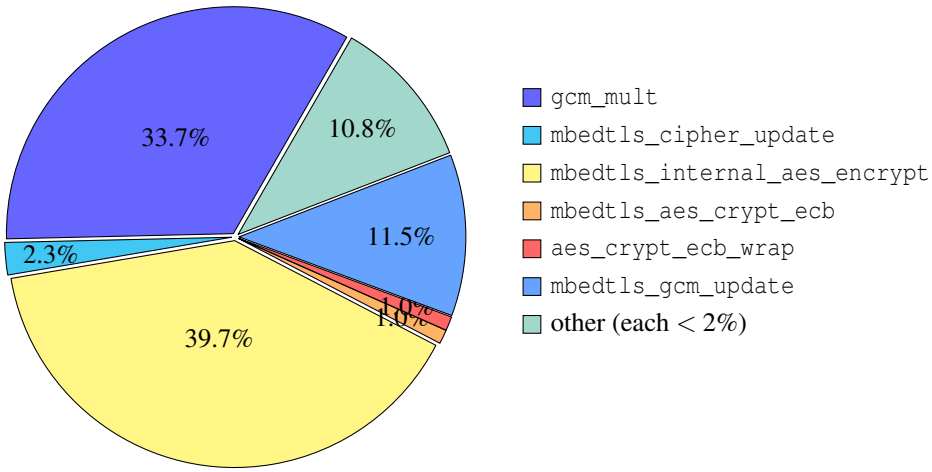



**Figure 11.3:** Distribution of execution time per function in profiling results

been executed. Figure 11.2 shows the distribution of these function calls. Only six functions are called more than 1%, those are shown separately. These functions account for almost 85% of all function calls. All of these functions, except `po_ctl` are part of mbed TLS and thus are related to encryption and authentication. This means that these functions most likely access data from different security domains and thus could benefit from the protection offered by the color labeling concept. The `po_ctl` function does is an OpenVPN function and is used to control polling events. This functions does not handle data from multiple security domains and thus will not be modified under the color labeling concept.

The execution time spend in each of these functions is shown in Figure 11.3. These numbers should only be used as an indication of the most computationally intensive functions, since the x86-64 test system used is very different from the RISC-V ISA. Again, the bulk of the execution time is spend in only a small number of functions: 85% of the total execution time is spend in only three functions. Note that the `mbedtls_aes_crypt_ecb` and `aes_crypt_ecb_wrap` functions account for less than 1% of the total execution time. The `mbedtls_gcm_update` function accounts for over 10% of the execution time, but accounts for less than 1% of the function calls (and thus does not appear in the graph of Figure 11.2).

### 11.3.2 Overhead of Color Labeling in C

The implementation in this work has been done in the Rust programming language, however, the concept is generally applicable. OpenVPN is written in the C programming language, conversion has to follow the C calling convention [41].

In order to be able to calculate the overhead of the concept of each of the most-called functions, this section explains the overhead of the concept for a C program.

#### Converting the Input Parameters

In the Rust implementation, currently some restrictions apply (see Section 7.9). One such restriction is that only primitive data types can be used. Here, however, data is passed by reference. In the C calling convention this means that pointers are passed in the argument registers. Just as for the Rust implementation, the parameters need to have a compile time known size. Color converting now consists of two parts:

1. Converting the color of the actual pointer;
2. Converting the color of the pointed-to data.

The pointer has the color of the callee. The color of the data pointed to is a compile time known constant. This makes it possible to use a simple do-while loop to iterate over the data words of the parameter. A program state diagram (PSD) [49] of this approach is shown in Figure 11.4.

A simple RISC-V assembly implementation of this algorithm is shown in Listing 11.1.

The code that changes the parameter colors is encapsulated between the `chg_color_` instructions and is repeated for each input parameter $\alpha_i$. This gives the formula of Equation 11.1 for the number of instructions for converting the input parameters $\alpha_i$.

$$\sum_{i=0}^{n} (3 + 5 \cdot \text{ wordsize}(\alpha_i)) \tag{11.1}$$

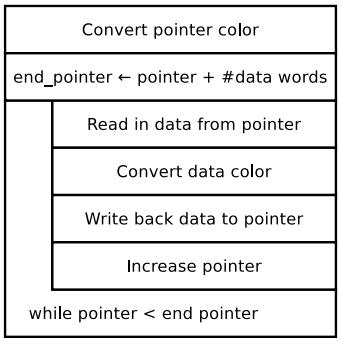



**Figure 11.4:** PSD of the algorithm for converting an input parameter

**Listing 11.1:** RISC-V assembly code for converting an input parameter according to the PSD of Figure 11.4.

```
1         chg_color_light run_color              # set run color, light tint
2
3         reg_from_color  a0, call_color         # convert pointer p
4         mv      t0, a0
5         addi    t1, t0, num                    # roundup(sizeof(*p)/sizeof(uint32_t))
6 lbl1:
7         lw      t2, 0(t1)
8         reg_from_color  t2, param_color1       # convert data
9         sw      t2, 0(t1)
10        addi    t1, t1, 4
11        bne     t0, t1, lbl1
12
13        chg_color_normal run_color             # set run color, normal tint
```

### Converting the Result

Conversion of the result can be done similar to converting the input parameters. Again, the size of the result has be known at compile time. Under that restriction, the same algorithm used for converting the input parameters can be used to convert the result. Note that conversion of the color of the pointer takes place in light tint, at the same time as the conversion of the input parameters, while conversion of the actual data takes place in dark tint. Note that only a single result $\rho$ can be returned, this simplifies the calculation of the additional number of instructions to the following equation:

$$3 + 5 \cdot \text{wordsize}(\rho) \tag{11.2}$$

### Other Per-function Overhead

Each special function requires three run-color and tint changing instructions to be executed:

1. from normal tint to light tint;
2. from light tint to normal tint;
3. from normal tint to dark tint;

4. from dark tint back to normal tint.

The final change from dark tint back to normal tint is automatically executed by the final `ret` instruction (see Section 5.4.2). This `ret` instruction is also extra due to the use of color labeling. Using these instructions adds four additional instructions per function call.

Finally, the original function is being wrapped with the color converting code. In order to call the original function, a new stack frame has to be created, the original function has to be called, and the stack frame has to be destroyed again. This requires a small amount of code, shown in Listing 11.2. As can be seen, five additional instructions are added for wrapping the original function call.

**Listing 11.2:** RISC-V assembly code for calling the original function.

```
1        addi    sp, sp, -16                 # Create stack frame, store ra
2        sw      ra, 8(sp)
3        jal     wrapped_c_function          # Call original function
4        lw      ra, 8(sp)                   # Restore ra, destroy stack frame
5        addi    sp, sp, 16
```

#### Total Overhead

The total overhead of the concept per function call can now be derived:

- 4 instructions for run-color changing;
- 5 instructions for wrapping the call to the original function;
- $\sum_{i=0}^{n}(3+5\cdot \text{ wordsize}(\alpha_i))$ instructions for color converting the input parameters $\alpha_{i\ldots n}$;
- $3+5\cdot \text{ wordsize}(\rho)$ instructions for color converting the result $\rho$.

This gives the following formula for the total overhead of using the color labeling concept:

$$4+5+\sum_{i=0}^{n}(3+5\cdot \text{ wordsize}(\alpha_i))+3+5\cdot\text{wordsize}(\rho) \tag{11.3}$$

### 11.3.3 Overhead per Function

The profiling data gives an overview of the most-often called functions. Each of the top-five of most-called functions is investigated. The overhead in instructions that would have been caused by the color labeling concept is calculated. In order to calculate the relative overhead, these functions have been separately compiled for RISC-V. The `gcm_mult` and `mbedtls_internal_aes_encrypt` are leaf functions that are specifically designed to have a run time independent of the inputs (in order to prevent timing side channel attacks). This enables simple determination of the amount of instructions executed per function. For the non-leaf function`mbedtls_cipher_update` only an estimation based on the execution times on the PC platform could be derived.

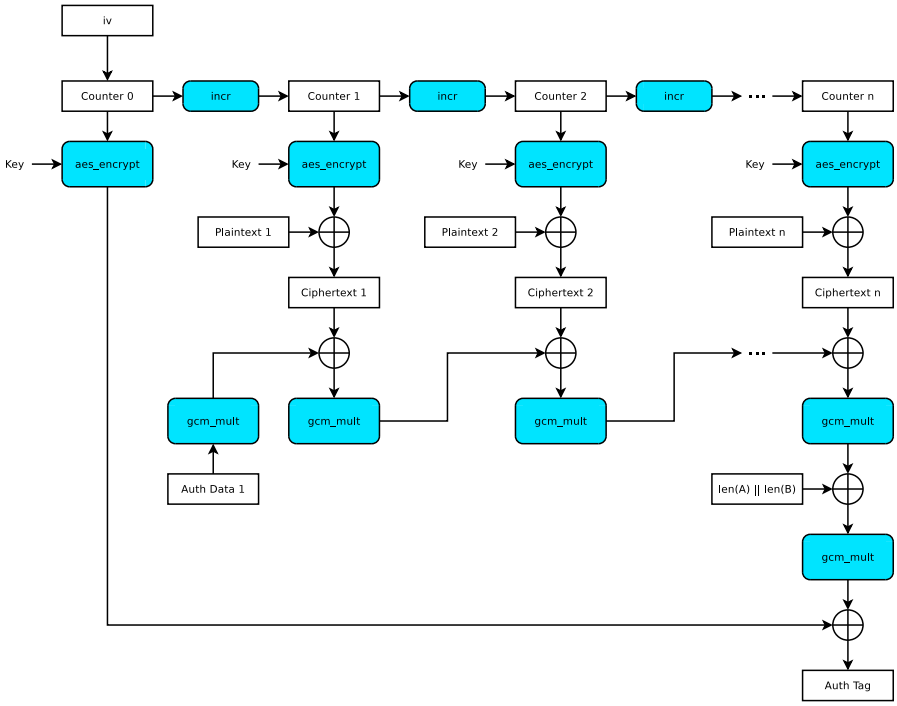



**Figure 11.5:** AES-GCM mode

### `gcm_mult`

The `gcm_mult` function performs a Galois field multiplication and is used in AES-GCM (Galois Counter Mode) to authenticate the encrypted message. Figure 11.5 gives an overview of AES-GCM. As can be seen in Figure 11.5 the ciphertext is xor-ed with the result of the previous `gcm_mult`. Both the input and the output of the function consist of data that has been authenticated, but not necessarily decrypted. This data should therefore reside in a separate security domain, different from both encrypted and unencrypted data. Since the input and output of the function both belong in the same security domain, the function can be implemented as a normal, color-agnostic function. In AES-GCM, the XOR operations process data from multiple security domains: one XOR processes the encrypted counter and the plaintext, resulting in ciphertext and the second XOR processes the ciphertext and authenticated data to produce authenticated data. Both XOR operations process two 4-word data inputs to produce a single 4-word result. Filling in in Equation 11.3 gives:

$$4+5+(3+5\cdot 4)+(3+5\cdot 4)+(3+5\cdot 4)=78$$

Using the color labeling concept thus requires 78 additional instructions for converting the input parameters and results per call of the XOR operation. As can be seen from Figure 11.5 two XOR operations take place for each `gcm_mult`. This means that the equivalent overhead of using the color labeling concept for the `gcm_mult` function is $2\cdot 78=156$ additional instructions.

Running a RISC-V compiled version of this function on the simulator reveals an instruction count of 2871 instructions for an unmodified version. This gives a relative overhead of 5.5% when run under the color labeling concept.

#### mbedtls_cipher_update

The `mbedtls_cipher_update` routine updates the data structure used to keep progress of the current encryption (or decryption) operation and encrypts (decrypts) the next part of the input data. The function has the following parameters:

**`mbedtls_cipher_context_t *ctx`** cipher context;

**`const unsigned char *input`** pointer to input buffer;

**`size_t ilen`** size of the input buffer;

**`unsigned char *output`** pointer to output buffer;

**`size_t *olen`** pointer to the size of the output buffer.

The selected encryption algorithm, the mode (encryption or decryption) and other state data is stored in a so-called context. The data in this context belongs to a security domain that should only be accessible by the functions that perform the encryption or decryption. This means that the struct should be colored with the run-color of these functions, which will exclude this structure from the color conversion procedure.

The input buffer and the variable containing the size should be converted, as should the output buffer and the variable containing the size of the output buffer. Here, the sizes of the input and output buffers are not fixed, however, for AES-GCM the buffers are always parsed in 16 char units (4 words), even though the actual size can be different. Note that the `ilen` variable is not passed via a pointer and converting the color just requires a single instruction.

Filling in these numbers in Equation 11.3 gives:

$$4+5+(3+5\cdot 4)+1+(3+5\cdot 4)+(3+5\cdot 1)=64$$

Using the color labeling concept thus requires 64 additional instructions.

This function is not a leaf function and as such it is difficult to derive an accurate average instruction count for the given setup. A very rough estimation based on the PC run times of the functions (as obtained via profiling) and averaging the RISC-V instruction counts for the `gcm_mult` and `mbedtls_internal_aes_encrypt` functions finds an estimated 170 instructions for the `mbedtls_cipher_update` function. This gives an estimated relative overhead of 37.6% when run under the color labeling concept.

#### mbedtls_internal_aes_encrypt

The `mbedtls_internal_aes_encrypt` function encrypts a 16-byte input buffer with a key of 128, 192 or 256 bits into a 16-byte output buffer. The function has the following parameters:

**`mbedtls_aes_context *ctx`** the AES context;

**`const unsigned char input[16]`** the 16-byte plaintext input buffer;

**`unsigned char output[16]`** the 16-byte encrypted output buffer;

The `mbedtls_aes_context *ctx` struct is the AES context struct that contains the state data of the AES encryprion. Similar to the cipher struct used in the `mbedtls_ciper_update` function this struct has to be colored in the run-color of the processing functions. This will exclude the structure from the color conversion procedure. The pointer to the struct is set by the calling function and has to be color converted.

The input and output buffer of the the `mbedtls_internal_aes_encrypt` routine both reside in a different security domain and should be assigned a unique color. This means that, in accordance with Figure 5.5, the following steps are taken:

- In light tint 4 words of plaintext input data (`input[16]`) should be converted into the run-color;
- In dark tint 4 words of ciphertext data in run-color should be converted into output color (`output[16]`).

Filling in Equation 11.3 for the input parameters and the result gives:

$$4+5+(3+5\cdot 4)+(3+5\cdot 4)=55$$

Using the color labeling concept thus requires 55 additional instructions per call of the `mbedtls_internal_aes_encrypt` function.

Running a RISC-V compiled version of this function on the simulator reveals an instruction count of 2429 instructions for an unmodified version. This gives a relative overhead of 2.3% when run under the color labeling concept.

#### `mbedtls_aes_crypt_ecb`

The function `mbedtls_aes_crypt_ecb` is a simple wrapper around the `mbedtls_internal_aes_encrypt` function and as such does not require any color conversions. The RISC-V compiled version of this function executes 6 instructions.

#### `aes_crypt_ecb_wrap`

The function `aes_crypt_ecb_wrap` is a simple wrapper around the `mbedtls_aes_crypt_ecb` function and as such does not require any color conversions. The RISC-V compiled version of this function executes 5 instructions.

### 11.3.4 Estimated Performance Impact and Improvements

Table 11.1 shows the total overhead of the concept, based on the five most-called functions in this test of OpenVPN-NL. As can be seen, almost $20 \cdot 10^9$ instructions are added by the concept for this test. The relative overhead, based on these five functions only, is found to be less than 5%.

Converting these numbers into actual execution times is difficult and depends on the actual hardware implementation (ISA, pipelining, caching, branch prediction, etc.). Since the Frenox RISC-V is highly configurable (including the use of e.g., hardware multipliers and dividers), the execution times will vary between each and every implementation of the core as well.

With the current implementation of the concept in Rust, the expected overhead is slightly larger than found here, since an additional function call will be added for each wrapped special function. As discussed in Section 7.9 the additional function calls may no longer be neccessary.

Currently, no applications designed specifically for the labeling concept exist. When an application is specifically designed for use with the labeling concept, optimizations are possible. These include solutions such as moving a color transformation inside a loop out of the loop.

**Table 11.1:** Overhead in instructions of using the color labeling concept for the five most-called functions of the OpenVPN-NL test.

| #Calls | Function Name | #Instructions | | #Additional Instructions | |
|---|---|---|---|---|---|
| | | per call | Total | per call | Total |
| 72476612 | `gcm_mult` | 2871 | 208080353052 | 178 | 11161398248 |
| 72476610 | `mbedtls_cipher_update` | 170 | 12321023700 | 64 | 4566026430 |
| 71303902 | `mbedtls_internal_aes_encrypt` | 2429 | 173197177958 | 55 | 3850410708 |
| 71303902 | `mbedtls_aes_crypt_ecb` | 6 | 427823412 | 0 | 0 |
| 71303777 | `aes_crypt_ecb_wrap` | 5 | 356518885 | 0 | 0 |
| Total | | | 394382897007 | | 19577835386 |

## 11.4 Area

Although an actual hardware implementation has not been done due to timing constraints, informed estimates on the required area of the design have been determined (see Chapter 10). The resource utilization of ALMs and memory blocks on an Intel FPGA has been shown to increase by 14% and 74%, respectively.

## 11.5 Conclusions

Via tests in the instruction set simulator, the concept is shown to work functionally correct for the set of attack patterns discussed in Chapter 2. The overhead of the concept is determined by profiling the real-life application Open-VPN-NL. Based on the profiling information, the realtive overhead based on the most called functions is less than 5%. The overhead in area of an actual hardware implementation is estimated to be around +14% ALMs and +74% memory blocks.

# Chapter 12

# Conclusions

## 12.1 Summary

This work introduces the concept and an implementation of *color labeling*.

In a literature study, common attack patterns and defence mechanisms have been investigated. It has been found that many of the defence mechanisms do not sufficiently protect against most of the attack patterns. More eleborate defence mechanisms in the form of special memory protection systems and tagged architectures have been investigated next. By augmenting memory locations with an associated tag, tagged architectures have the potential to protect against a broad range of attacks.

In order to find out what functionality would be required for implementing a tagged architecture for implementing security domain separation, a simple, but authorative test program has been deviced. This lead to a number of requirements on the hardware, software (toolchain), and programmers' interface. Using these requirements, the basic color labeling concept has been designed. Theoretical evaluation has shown that the concept, when properly applied, is able to protect against all investigaged common attack patterns.

Software support for the concept has been implemented in the Rust programming language. After investigation of different implementation options, procedural macros proved to be best option. The concept has been implemented using these macros, a specially adapted platform support crate, and linker scripts. A special separate program is used to process the resulting binary and process and extract the included label information. In order to test this setup, an instruction set simulator has been modified to support color labeling. Testing has shown that this implementation of the concept can succesfully protect against a broad range of attacks. The implementation is subject to some limitations, some of which regard mostly unused features and are thus deemed of no importance. Other limitations are mostly due to limited support in the language at the time of implementation. The Rust macro system is under very active development which makes it reasonable to expect these limits to be alleviated in the near future.

In a typical application (OpenVPN-NL) less then 5% additional instructions were required to implement color labeling.

Due to timing constraints, no actual hardware implementation has been build. However, implications for a possible hardware implementation have been investigated and estimates for overhead in both size and execution time have been established. The performance overhead in hardware has been estimated to be negligible. Extending the RISC-V Frenox core to support the color labeling concept requires an estimated

14% additional ALMs and 74% additional memory blocks.

## 12.2 Main Contributions

The research question of this work, as stated in Chapter 1 and repeated here, was:

*How can data labeling be used to implement a hardware-enforced security domain separation on a RISC-V processor with support in the Rust Programming Language?*

In order to answer this question, first a literature study was conducted on known attack patterns and defences and existing solutions for data labeling. The results were used as a bottom-up approach for the color labeling concept. A simple, but representative test program provided the top-down approach for the concept. These two approaches led to a set of requirements for the color labeling concept.

Based on these requirements, the color labeling concept was derived. The concept can be summarized as follows:

- Data words have associated tags, consisting of a color number and several special tag bits;
- The data and the tag are propagated inseparable throughout the SoC, including but not limited to the register file, memory, cache, peripherals;
- The CPU runs in a certain color and tint;
- Reading from and writing to memory will just propagate the tags;
- During execution of an instruction, the tag of the source registers is checked against the color and tint of the processor. Only if these match, the operation can succeed;
- Upon failure, a security breach has been detected and the processor will halt or reboot;
- Handling data from multiple security domains takes place in special functions that are logically split up into three parts, denoted with tints:

  **light tint** is used to convert the color of input parameters;

  **normal tint** performs the intended operation on the re-colored data;

  **dark tint** is used to convert the result of the operation.

- Control flow integrity is implemented using two special tag bits:

  **code bit** is used to differentiate between code and data at a word level granularity;

  **callable bit** is used to denote a word containing a valid address;

Support for the concept is then implemented in the Rust programming language. First a programmers' interface was designed, based on standard language features. Next, the best way to implement this interface in the toolchain was determined. Based on the requirements of the programmers' interface and maintainability concerns, procedural macros were chosen to implement this functionality.

An external program has been created to process the binary and extract the embedded color information. This tool generates color load information and a security report that can be used in a security audit. By

not including this functionality in the compiler but in a simple external program, the Rust compiler has been removed from the trusted code base. In addition, this enables the use of a main stream compiler, since no color labeling specific support has been added to the compiler. This keeps the entire toolchain maintainable and makes it possible to subject the toolchain to a security audit.

The main contributions of this work are;

- The development of a novel labeling concept for hardware-enforced security domain separation, based on a tagged RISC-V architecture;
- Implementation of a programmers' interface of the concept in the Rust programming language;
- Providing a way to propagate the color labeling information for variables and special functions through the compiler into the binary;
- Implementation of a software tool to extract color information from the binary in order to evaluate the results without significant trust in the compiler;
- Implementation of the concept in a RISC-V instruction set simulator;

The novelty and conceptual nature of the work are further emphasized by the pending patent application.

The goals of the work have been met:

- The programmer can annotate variables with a color to indicate the security domain (see Chapter
- The programmer can annotate functions handling data from multiple security domains (see Chapter
- The toolchain is able to process the annotations, generate label information, and can generate the functionality for special functions (see Chapter
- The hardware will enforce the policies to preserve domain separation even under software failure (see Chapter

The main goal of the work, preventing cross-security domain information leaks, has also been met, as has been shown by both the theoretical evaluation of the concept (Section 5.8) and the tests performed (Section 11.2).

## 12.3 Recommendations for Future Work

Color labeling is a novel concept and opens up a wide range of possible applications and further research:

- One of the main problems with this work is the absence of a code base that can utilize the concept. Some of the design decisions and imposed restrictions are based on educated guesses of the required features for programs targeted by the concepts. Furthermore, no programs have been created specifically for use with the concept. Designing a program specifically for this concept might make for more efficient special functions. Creating a code base of programs that can benefit from the added assurances of the concept should be a a high-priority follow-up task.
- Given the highly experimental nature of the concept, it is likely that adaptation in high assurance products will require more experience with the concept and thus a larger code base. An alternative approach might be to place more decision making into the compiler and automatically color different variables with a different color. This makes it possibible to utilize the color labels for maintaining the scopes and bounds of variables.

- An actual hardware implementation has to be implemented. Currently, the concept has been implemented in the Fremu simulator, it would be valuable to have an actual hardware implementation of the concept in order to verify the assumptions from Chapter 10 and test code on an actual hardware platform;
- The current software implementation has a number of limitations (see Section 7.9). Since the Rust toolchain is still under (very) active development, some of these limitations may no longer exist in the near future. Alternative solutions such as a compiler plugin or an LLVM pass can be examined as well;
- Currently, it is not possible to use a pre-emptive multitasking OS such as Linux. Adding support for Linux to the concept and adding support in Linux for the concept are two highly interdependent tasks that will enable the use of a very large code base;
- The current implementation can be polished. Furthermore, a number of limitations might be removed by future additions in the Rust macro system or by modifying part of the user interface. Alternatively, a different implementation method, such as a compiler plugin or an LLVM pass, might be considered;
- A number of tag bits are unused in the current implementation, use cases for these bits can be explored.
- In the current implementation, code is not colored, the execution color is determined by the internal state of the processor. Further research could focus on repurposing the label bits used for storing the color number. Alternatively, possible benefits of coloring code can be examined.

# Bibliography


[1] OpenSSL, "Openssl security advisory [07 apr 2014]." [Online]. Available: https://www.openssl.org/news/secadv/20140407.txt

[2] Codenomicon, "Heartbleed website." [Online]. Available: http://heartbleed.com/

[3] Netcraft, "Half a million widely trusted websites vulnerable to Heartbleed bug." [Online]. Available: https:://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html

[4] CBC, "Heartbleed bug: RCMP asked Revenue Canada to delay news of SIN thefts." [Online]. Available: https://www.cbc.ca/news/business/heartbleed-bug-rcmp-asked-revenue-canada-to-delay-news-of-sins-thefts-1.2609192

[5] D. Kennedy. (2014) CHS hacked via Heartbleed vulnerability. TrustedSec. [Online]. Available: https://www.trustedsec.com/2014/08/chs-hacked-heartbleed-exclusive-trustedsec/

[6] M. Prince. (2014) The hidden costs of Heartbleed. Cloudflare. [Online]. Available: https://blog.cloudflare.com/the-hard-costs-of-heartbleed/

[7] Wikipedia contributors. Heartbleed — Wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.com/wiki/Heartbleed

[8] Mitre, "Common vulnerabilities and exposures." [Online]. Available: https://cve.mitre.org/

[9] Technolution. (2019) Technolution website. [Online]. Available: https://www.technolution.eu/

[10] Genode Labs. (2019) Genode Operting System Framework. [Online]. Available: https://genode.org/

[11] Technolution. (2019) RISC-V softcore: Veilige hardware voor embedded systemen. [Online]. Available: https://www.technolution.eu/nl/public-safety-security/19-risc-v-softcore-veilige-hardware-voor-embedded-systemen.html

[12] A. Waterman, K. Asanović, *The RISC-V Instruction Set Manual – Volume I: Unprivileged ISA*, June 2019. [Online]. Available: https://riscv.org/specifications/

[13] Rust Team. (2019) Rust Programming Language. [Online]. Available: https://www.rust-lang.org/

[14] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A.-R. Sadeghi, "HardScope: Thwarting DOP with Hardware-assisted Run-time Scope Enforcement," *Cryptography and Security*, vol. abs/1705.10295, 2017. [Online]. Available: http://tubiblio.ulb.tu-darmstadt.de/110847/

[15] The MITRE Corporation. CWE VIEW: Research Concepts. [Online]. Available: https://cwe.mitre.org/data/definitions/1000.html

[16] M. in 't Hout, "Kernel isolation of a capability-based security operating system," Master's thesis, Delft University of Technology, October 2018. [Online]. Available: http://resolver.tudelft.nl/uuid:41767be9-f48f-468e-abf6-949dbd7cce96

[17] The MITRE Corporation. CWE-680: Integer Overflow to Buffer Overflow. [Online]. Available: https://cwe.mitre.org/data/definitions/680.html

[18] M. Rushanan and S. Checkoway, "Run-DMA," in *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. Washington, D.C.: USENIX Association, Aug. 2015. [Online]. Available: https://www.usenix.org/conference/woot15/workshop-program/presentation/rushanan

[19] Aleph One. (1996) Smashing the stack for fun and profit. [Online]. Available: http://phrack.org/issues/49/14.html

[20] T. Kornau, "Return oriented programming for the arm architecture. technical report," Master's thesis, Bochum, 2010.

[21] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Info. & System Security*, vol. 15, no. 1, Mar. 2012.

[22] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 969–986.

[23] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. IEEE Press, 2016, pp. 40:1–40:13.

[24] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *CoRR*, vol. abs/1801.01203, 2018. [Online]. Available: https://spectreattack.com/spectre.pdf

[25] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990. [Online]. Available: https://meltdownattack.com/meltdown.pdf

[26] Bitdefender. (2019) SWAPGS attack. [Online]. Available: https://www.bitdefender.com/business/swapgs-attack.html

[27] R. McIlroy, J. Sevcík, T. Tebbi, B. Titzer, and T. Verwaest. (2019) Spectre is here to stay: An analysis of side-channels and speculative execution. [Online]. Available: http://arxiv.org/abs/1902.05178

[28] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 304–316. [Online]. Available: http://doi.acm.org/10.1145/605397.605429

[29] "Oracle's SPARC T8 and SPARC M8 Server Architecture," White Paper, Oracle, pp. 12–13, Sept 2017.

[30] RB. (2013) Introduction to intel memory protection extensions. Intel. [Online]. Available: https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions

[31] A. Bradbury, G. Ferris, and R. Mullins. (2014) Tagged memory and minion cores in the lowRISC SoC. [Online]. Available: https://www.lowrisc.org/docs/memo-2014-001-tagged-memory-and-minion-cores/

[32] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon, "Architectural support for software-defined metadata processing," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 487–502, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694383

[33] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, "Shakti-t: A risc-v processor with light weight security extensions," in *Proceedings of the Hardware and Architectural Support for Security and Privacy*, ser. HASP '17. New York, NY, USA: ACM, 2017, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/3092627.3092629

[34] W. Diffie and M. E. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. 22, pp. 644–654, nov 1976.

[35] Wikipedia contributors. Diffie-Hellman key exchange. [Online]. Available: https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange

[36] S. Huurman, "Evaluatie van de bruikbaarheid van Rust voor de RISC-V – Analyserapport," B. Eng. Thesis, De Haagse Hogeschool, April 2018.

[37] S. Klabnik, C. Nichols *et al.* (2019) The Rust Programming Language – Appendix G: How Rust is Made and "Nightly Rust". [Online]. Available: https://doc.rust-lang.org/book/appendix-07-nightly-rust.html

[38] Wikipedia contributors. Abstract syntax tree — Wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.com/wiki/Abstract_syntax_tree

[39] S. Klabnik, C. Nichols *et al.* (2019) The Rust Programming Language – Macros. [Online]. Available: https://doc.rust-lang.org/book/appendix-07-nightly-rust.html

[40] LLVM-admin team. (2019) The LLVM Compiler Infrastructure Project. [Online]. Available: https://llvm.org/

[41] P. Dabbelt, S. O'Rear, K. Cheng, A. Waterman, M. Clark, A. Bradbury, D. Horner, M. Nordlund, and K. Merker. (2019) RISC-V ELF psABI Specification. [Online]. Available: https://gitbuh.com/riscv/riscv-elf-psabi-doc/

[42] S. Klabnik, C. Nichols *et al.* (2019) The Rust Programming Language – Using Trait Objects that Allow for Values of Different Types. [Online]. Available: https://doc.rust-lang.org/book/ch17-02-trait-objects.html

[43] Linaro Limited. (2019) Device tree. [Online]. Available: https://www.devicetree.org/

[44] Wikipedia contributors. Executable and Linkable Format. [Online]. Available: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

[45] (2019) OpenVPN-NL. FOX-IT. [Online]. Available: https://openvpn.fox-it.com/

[46] (2019) Vpn software solutions & services for business. OPENVPN INC. [Online]. Available: https://openvpn.net/

[47] (2019) OpenSSL – Cryptography and SSL/TLS Toolkit. OpenSSL Software Foundation. [Online]. Available: https://www.openssl.org/

[48] (2019) SSL Library mbed TLS / PolarSSL. ARM Limited. [Online]. Available: https://tls.mbed.org/

[49] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices*, vol. XII, aug 1973. [Online]. Available: http://www.cs.umd.edu/hcil/members/bshneiderman/nsd/1973.pdf

# Appendix A

# Statistics on CVE Database

## A.1 Introduction

The Common Vulnerabilities and Exposures website [8] contains the CVE database, a list of publicly known security vulnerabilities. The list can be freely downloaded in different file formats, including the plain text CSV format. Although suitable for importing into a spreadsheet program, the CSV format lends itself for use with standard Posix text processing tools such as `grep` and `sed`. Using these tools a number of metrics can easily be extracted from the database

## A.2 Obtaining the Number of CVE Entries

A line of the CSV version of the CVE database containing a CVE has the following format:
`CVE-year-sequence number, "Description", Phase, Votes, Comments`

Of these fields, only the Name, Description and Comments fields are still in active use:

**Name** is the assigned CVE designator. It consists of the text `CVE-` followed by a four-digit year specification and an ID;

**Description** field contains a textual description of the CVE entry;

**Comments** is intended for remarks, but seldomly used anymore;

The description field can also contain the text `REJECT` for rejected CVE entries and `RESERVED` for reserved but as of yet unassigned entries. The total number of CVE entries from 1999–2018 can now easily be determined with an `egrep` commands:

```
egrep allitems.cvs "^CVE-\b(1999|200[0-9]|201[0-8])\b" | grep -v REJECT | grep -v RESERVED | wc -l
```

And for the entries of 2018 only:

```
egrep allitems.cvs "^CVE-2018" | grep -v REJECT | grep -v RESERVED | wc -l
```

For the version of 2019-07-03 this yields 113410 and 14721 respectively.

## A.3 Entries Related to Buffer Overflows

The description field of a CVE entry is a freeform text field. As a consequence, slightly different wording or spelling is used for describing the same issue. For example, the term "buffer overflow" gives 9612 results for 1999–2018, the term "buffer-overflow" yields 475 results.[1] Likewise "buffer overread" (33), "buffer over read" (5) and "buffer over-read" (707) all yield different results for the same problem.[2]

The following regular expressions were searched for:

- `buffer.overflow`
- `buffer.over.?read`
- `buffer.over.?write`
- `read.past.the.end`
- `write.past.the.end`
- `integer.overflow`
- `double.free`
- `use.after.free`
- `out.of.bound.read`
- `out.of.bound.write`

These search terms gave 2467 hits for 2018 alone and 15677 hits for 1999–2018 corresponding to 13.8% and 16.8% respectively.

[1] The correct spelling is "buffer overflow"
[2] The correct spelling is "buffer over-read"

# Appendix B

# Heartbleed Explanation According to XKCD

The heartbleed bug is (slightly simplified) very clearly described by XKCD comic 1354 at `https://xkcd.com/1354/`, also shown here in Figure B.1.

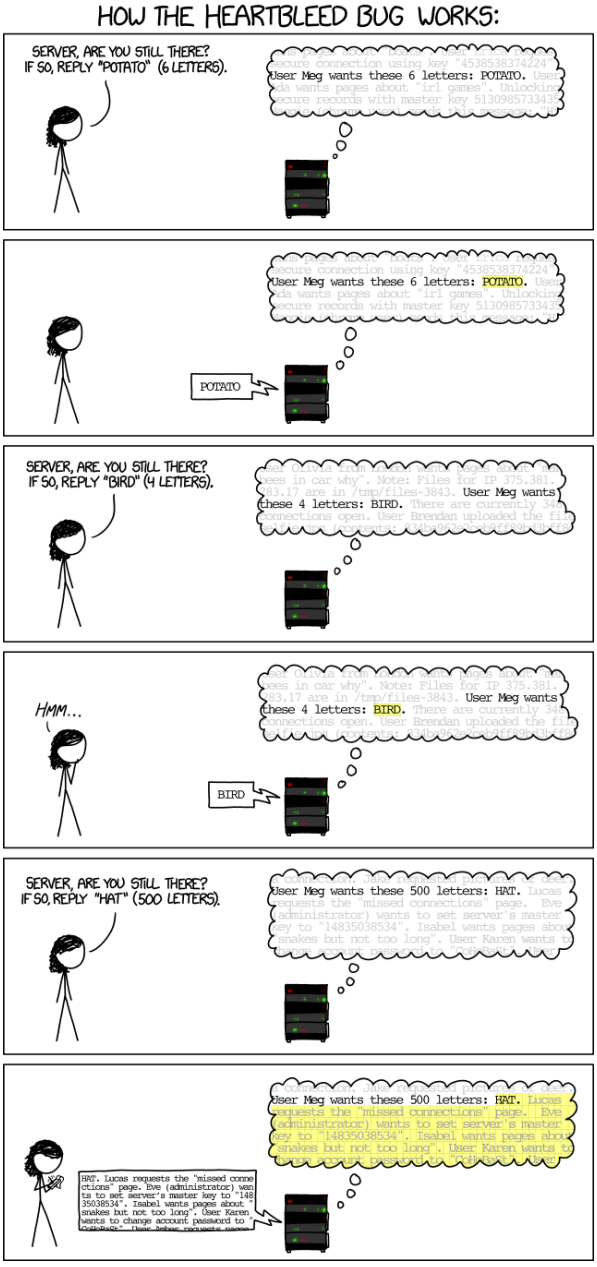



**Figure B.1:** Slightly simplified explanation of the Heartbleed Bug, according to XKCD 1354: `https:xkcd.com/1354/`

# Appendix C

# Test Program: Key Exchange

This is the key exchange test program used to derive some of the requirements of color labeling.

```
#include <stdio.h>
#include <string.h>

#define BUF_LEN (16)

int receive_pub_key_B (void)
{
    fprintf (stderr, "receive_pub_key_B\n");

    // Problematic user input as characters (out-of-bounds check)
    char key[20];
    scanf ("%s", key);

    // Convert problematic user input to number (out-of-bounds check)
    int B;
    sscanf (key, "%d", &B);

    return B;
}

void send_public_key_A (int public_key)
{
    // Should only send public data, eg. public key or encrypted
    fprintf (stderr, "send_public_key_A, key = %d\n", public_key);
}

int generate_symm_key (                         // return secret
    int p,                                      // public
    int g,                                      // public
    int a)                                      // secret (key)
{
    // Generates (secure) key data based on secure and public data
    // -> potential data privilege escalation
    int pub_key_B = receive_pub_key_B ();
    int symm_key = pub_key_B^a % p;

    fprintf (stderr, "symm_key = %d\n", symm_key);

    return symm_key;
}

int generate_public_key (                       // return secret
    int g,                                      // public
    int p,                                      // public
    int a)                                      // secret (key)
{
    // Generate public data based on secure and public data
```

```
48      // -> potential information disclosure
49      int pub_key_A = g^a % p;
50
51      fprintf (stderr, "pub_key_A = %d\n", pub_key_A);
52      return pub_key_A;
53  }
54
55  void encrypt_data ( char *encrypted_buffer,     // public
56                      char *plain_text_buffer,    // secret (data)
57                      unsigned int length,        // public/local
58                      int key)                    // secret (key)
59  {
60      // Intentional conversion from secret data (plain text) to public data
61      // -> Potential information disclosures
62      unsigned int i;
63      for (i = 0; i < length; i++) {  // Potential buffer overflow
64          encrypted_buffer[i] = plain_text_buffer[i] ^ key;
65      }
66  }
67
68  void send_data (    char *buffer,               // public (encrypted data)
69                      unsigned int length)        // public/local
70  {
71      // Only send public data (public key, encrypted data)
72      // -> Potential buffer over-read might lead to information disclosure
73      printf ("send_data:\n");
74
75      unsigned int i;
76      for (i = 0; i < length; i++) {              // Potential buffer overread
77          putchar (buffer[i]);
78      }
79      putchar ('\n');
80  }
81
82  void receive_data ( char *buffer,               // public
83                      unsigned int length)        // public/local
84  {
85      // Read in public (encrypted) data
86      // -> potential buffer overflow
87      printf ("receive_data:\n");
88
89      const char data[] = {                       // public, send over net
90          'j', 'o', 'z', 'o', '.',
91          'j', 'o', 'z', 'o', '.',
92          'j', 'o', 'z', 'o', '.', (char)14
93      };
94
95      unsigned int i;
96      for (i = 0; i < length; i++) {  // Potential buffer overflow
97          buffer[i] = data[i];
98      }
99  }
100
101 void decrypt_data ( char *plain_text_buffer,    // secret (data)
102                     char *encrypted_buffer,     // public (encrypted data)
103                     unsigned int length,        // public/local
104                     int key)                    // secret (key)
105 {
106     // Intentional conversion from public (encrypted data) to secret data
107     // -> Potential information disclosure
108     unsigned int i;
109     for (i = 0; i < length; i++) {              // Potential buffer overflow
110         plain_text_buffer[i] = encrypted_buffer[i] ^ key;
111     }
112 }
113
114 static int a = 4;   // a = global secret key data
115
116 int main (void)
117 {
118     int p = 23; // p, g = local public data
119     int g = 5;
120
```

```
121     int pub_key_A = generate_public_key (g, p, a);  // Public data from (secret key data + public data)
122
123     send_public_key_A (pub_key_A);                  // Only public data (public keys, encrypted data)
124                                                     // should be send out over the network interface
125
126     int symm_key = generate_symm_key (p, g, a);     // symm_key is generated based on public and secret data
127                                                     // and is secret data by itself
128
129     char data_buffer[BUF_LEN] = "data data data ";  // Secret data: plain text
130
131     char encrypted_buffer[BUF_LEN];                 // Public data: encrypted data from secret data and secret key
132     encrypt_data (encrypted_buffer, data_buffer, BUF_LEN, symm_key);
133
134     send_data (encrypted_buffer, BUF_LEN);          // Only send public (encrypted) data
135
136     char received_buffer[BUF_LEN];
137     receive_data (received_buffer, BUF_LEN);        // Only read public (encrypted) data
138
139     char decrypted_buffer[BUF_LEN];                 // Secret data: plain text
140     decrypt_data (decrypted_buffer, received_buffer, BUF_LEN, symm_key);
141
142     printf ("Decrypted data = %s\n", decrypted_buffer);
143
144     return 0;
145 }
```

# Appendix D

# Color Label Instructions Extension for the RISC-V

A number of new instructions is required to implement this mechanism. The RISC-V architecture is designed to be extensible. The new instructions form a non-standerd extension and are currently implemented as a *greenfield extension* [12]. This means a new opcode is used to select the extension, the actual instructions are encoded in other fields. As opcode the *custom-0* code is selected: 010. For the instruction encoding currently the I-type convention is used.

## D.1 `CHG_COLOR_LIGHT` – Change Core to Color in Light Tint

Changes the core color to the color specified as constant and the core tint to "light". This instruction is only allowed when:

- current core tint is "normal"

Operation:
core tint ← light
core color ← K

| Syntax: | Operands | Program Counter |
|---|---|---|
| CHG_COLOR_LIGHT K | $0 \leq K < 2^{12}$ | PC ← PC + 4 |

Encoding:

| color [11..0] | 00000 | 000 | 00000 | 0001011 |
|---|---|---|---|---|
| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |

## D.2 `CHG_COLOR_NORMAL` – Change Core to Color in Normal Tint

Changes the core color to the color specified as constant and the core tint to "normal". This instruction is only allowed when:

- current core tint is “light” and
- specified color is identical to current core color

Operation:
core tint ← normal
core color ← K

| Syntax: | Operands | Program Counter |
|---|---|---|
| CHG_COLOR_NORMAL K | $0 \leq K < 2^{12}$ | PC ← PC + 4 |

Encoding:

| color [11..0] | 00000 | 001 | 00000 | 0001011 |
|---|---|---|---|---|
| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |

## D.3 `CHG_COLOR_DARK` – Change Core to Color in Dark Tint

Changes the core color to the color specified as constant and the core tint to “dark”. This instruction is only allowed when:

- current core tint is “normal” and
- specified color is identical to current core color

Operation:
core tint ← dark
core color ← K

| Syntax: | Operands | Program Counter |
|---|---|---|
| CHG_COLOR_DARK K | $0 \leq K < 2^{12}$ | PC ← PC + 4 |

Encoding:

| color [11..0] | 00000 | 010 | 00000 | 0001011 |
|---|---|---|---|---|
| 31 20 | 19 15 | 14 12 | 11 7 | 6 0 |

## D.4 `REG_FROM_COLOR` – Change Register in Specified Color to Core Color

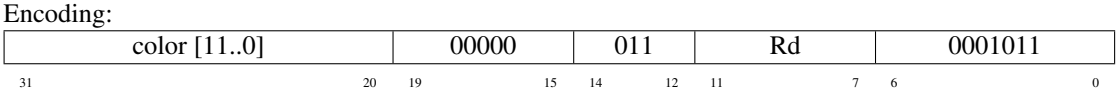
Changes the color of a register in the color specified as constant to the current core color and “normal” tint. This instruction is only allowed when:

- current core tint is “light” and
- specified color is identical to current color of register Rd

Operation:
Rd tint ← normal
Rd color ← core color

| Syntax: | Operands | Program Counter |
|---|---|---|
| REG_FROM_COLOR Rd, K | $0 \leq K < 2^{12}$ | PC ← PC + 4 |

Encoding:




## D.5 REG_TO_COLOR – Change Register Color to Specified Color

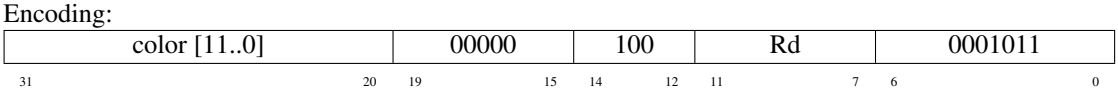
Changes the color of a register into the color specified as constant and "normal" tint. This instruction is only allowed when:

- current core tint is "dark" and
- register color is identical to current core color

Operation:
Rd tint ← normal
Rd color ← K

| Syntax: | Operands | Program Counter |
|---|---|---|
| REG_TO_COLOR Rd, K | $0 \leq K < 2^{12}$ | PC ← PC + 4 |

Encoding:



# Appendix E

# Example RISC-V Rust Program with Color Annotations

## E.1 Introduction

This appendix contains a small example RISC-V Rust program with color definitions, a color annotated variable, and a color annotated function. The macro expanded code and the generated files (assembly and linker script) are shown as well.

## E.2 Rust Program

A simple, color annotated, Rust program. The value of the `key` variable is read from a green peripheral that is defined in the Device Tree.

```
#![no_main]
#![no_std]

extern crate riscv_rt as rt;
extern crate panic_halt;

use rt::entry;
use core::ptr;
use colors_proc_macro::{color_label, new_color};

// Color definitions
new_color!(Black, 1);
new_color!(Red, 2);
new_color!(Green, 3);
new_color!(Blue, 4);

#[color_label(Red)]
static mut RED_DATA: u32 = 0;

#[color_label(fn = Blue, return = Red, args(encrypted=Black, key=Green))]
```

```rust
fn decrypt(encrypted: u32, key: u32) -> u32 {
    encrypted ^ key
}

#[entry]
fn main() -> ! {
    let encrypted: Black<u32> = Black(20);
    let key = unsafe{ ptr::read_volatile(0x60040000 as *const Green<u32>) };
    unsafe { RED_DATA = decrypt(encrypted, key); };

    loop { }
}
```

## E.3 Macro Expanded Program

This is the fully macro expanded version of the example program.

```rust
#![feature(prelude_import)]
#![no_std]
#![no_main]
#![no_std]
#[prelude_import]
use ::core::prelude::v1::*;
#[macro_use]
extern crate core as core;
#[macro_use]
extern crate compiler_builtins as compiler_builtins;
extern crate panic_halt;
extern crate riscv_rt as rt;
use colors_proc_macro::{color_label, new_color};
use core::ptr;
use rt::entry;
pub trait Color {
    const COLOR: u16;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Black<T>(T);
impl<T> Color for Black<T> {
    const COLOR: u16 = 1;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Red<T>(T);
impl<T> Color for Red<T> {
    const COLOR: u16 = 2;
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Green<T>(T);
impl<T> Color for Green<T> {
    const COLOR: u16 = 3;
```

```rust
}
#[repr(transparent)]
#[allow(dead_code)]
pub struct Blue<T>(T);
impl<T> Color for Blue<T> {
    const COLOR: u16 = 4;
}
static mut RED_DATA: Red<u32> = Red(0);
#[link_section = ".color.2"]
#[allow(dead_code)]
#[used]
static RED_DATA_ENTRY: &'static Red<u32> = unsafe { &RED_DATA };
#[inline(never)]
#[no_mangle]
extern "C" fn wrapped_decrypt(encrypted: u32, key: u32) -> u32 {
    encrypted ^ key
}
extern "C" {
    fn decrypt(encrypted: Black<u32>, key: Green<u32>) -> Red<u32>;
}
#[export_name = "main"]
pub fn s3702y3g24fw4s6f() -> ! {
    let encrypted: Black<u32> = Black(20);
    let key = unsafe { ptr::read_volatile(0x60040000 as *const Green<u32>) };
    unsafe {
        RED_DATA = decrypt(encrypted, key);
    };
    loop {}
}
```

## E.4 Assembly Wrapper Code

This is the generated assembly wrapper code that will perform the run-color change and the parameter and result color conversions.

```
.text
.extern wrapped_decrypt
.globl decrypt
decrypt:
        .word       0x0040000b              # chg_color_mode_light <Blue> (4)
        # Converting parameters
        .word       0x0010350b              # reg_from_color <Black>, a0        (encrypted)
        .word       0x0030358b              # reg_from_color <Green>, a1        (key)

        # Store ra and call wrapped function
        .word       0x0040100b              # chg_color_mode_normal <Blue>
        addi        sp, sp, -16
        sw          ra, 8(sp)
        jal         wrapped_decrypt
        lw          ra, 8(sp)
        addi        sp, sp, 16
        .word       0x0040200b              # chg_color_mode_dark <Blue>

        # Convert result (u32)
        .word       0x0020450b              # reg_to_color <Red>, a0
        ret
```

## E.5 Linker Script Lines

These generated linker script lines are included in the linker script of the RISC-V-rt crate. These lines make sure that the variables in the `.color` sections are kept in the executable.

```
.colors.1 : ALIGN(4) { KEEP(*(.color.1)); } > FLASH
.colors.2 : ALIGN(4) { KEEP(*(.color.2)); } > FLASH
.colors.3 : ALIGN(4) { KEEP(*(.color.3)); } > FLASH
.colors.4 : ALIGN(4) { KEEP(*(.color.4)); } > FLASH
```

# Appendix F

# Exploit Code

## F.1 Introduction

This appendix contains some small programs that implement different attacks in order to test the color labeling concept. Since implementing most of these exploits in Rust is very difficult, C routines have been created and called via the Foreign Language Interface (FFI). This models the very plausible situation of a security breach via an external library.

## F.2 Buffer Over-read

This program defines a struct in C representation and creates two variables (`RED_STRUCT` (in color `Red`) and `GREEN_STRUCT` (in color `Green`) of the struct type. The C program then over-reads the `RED_STRUCT` variable in `Red` context. This generates a label exception.

### F.2.1 Rust Program

```
#![no_main]
#![no_std]

extern crate riscv_rt as rt;
extern crate panic_halt;

use rt::entry;
use colors_proc_macro::{color_label, new_color};

// Color definitions
new_color!(Black, 1);
new_color!(Red, 2);
new_color!(Green, 3);
new_color!(Blue, 4);

#[repr(C)]
struct TestStruct {
```

```
    a: u32,
    b: u32,
}

#[color_label(Red)]
static RED_STRUCT: TestStruct = TestStruct { a: 10, b: 20 };

#[color_label(Green)]
static GREEN_STRUCT: TestStruct = TestStruct { a: 10, b: 20 };

#[color_label(fn = Red)]
fn test_func () {
    unsafe { buf_overflow(&RED_STRUCT, &GREEN_STRUCT); };
}

extern "C" {
    fn buf_overflow (p: &Red<TestStruct>, q: &Green<TestStruct>);
}

#[entry]
fn main() -> ! {
    unsafe { test_func(); };

    loop { }
}
```

### F.2.2 C Program

```
#include <stdint.h>

struct TestStruct {
        uint32_t a;
        uint32_t b;
};

void buf_overflow (struct TestStruct *p, struct TestStruct *q) {
        uint32_t *void_p = (uint32_t*)p;
        uint32_t *void_q = (uint32_t*)q;

        // TestStruct is 2 uint32_t, let's process 4
        for (unsigned i = 0; i < 4; i++) {
                *void_q = *void_p++ + 1;
        }
}
```

## F.3 Code Injection

The C routine injects code by overwriting the first instruction of the routine with a `ret` instruction and then recursively call the routine again. Overwriting the instruction will succeed but should clear the `code` bit that in turn will prevent execution of the instruction.

### F.3.1 Rust Program

```
#![no_main]
#![no_std]

extern crate riscv_rt as rt;
extern crate panic_halt;

use rt::entry;
use colors_proc_macro::{color_label, new_color};

// Color definitions
new_color!(Black, 1);

extern "C" {
    fn code_injection ();
}

#[entry]
fn main() -> ! {
    unsafe { code_injection(); };

    loop { }
}
```

### F.3.2 C Program

```
void code_injection (void) {
        void (*p)(void) = 0x800000a0;        // = code_injection;

        *(int *)p = 0x00008067;               // ret

        code_injection();
}
```

## F.4 Return Oriented Programming

The C routine overwrites the its return address with the start address of the function itself. Doing so, however, will push a return address without the `callable` bit set and result in an exeception on execution of the `ret` instruction.

### F.4.1 Rust Program

```
#![no_main]
#![no_std]

extern crate riscv_rt as rt;
extern crate panic_halt;
```

```
use rt::entry;
use colors_proc_macro::{color_label, new_color};

// Color definitions
new_color!(Black, 1);

extern "C" {
    fn rop ();
}

#[entry]
fn main() -> ! {
    unsafe { rop(); };

    loop { }
}
```

### F.4.2 C Program

```
void rop (void) {
        int a = 20;
        int *p = &a;


        *(p+5) = 0x800000a;        // Modify return address on stack
        foo();        // Need to call other function to force store of ra reg
}

void foo(void) {
}
```

# Appendix G

# Benchmarking OpenVPN-NL

## G.1 Introduction

In order to measure the performance impact of the labeling concept, test have been performed on OpenVPN-NL [45] running on a PC. The benchmark was used to derive the number of function calls to each function (see Chapter 11). This appendix describes the steps required to perform this benchmark. This text assumes the benchmark is performed under Linux.

## G.2 OpenVPN-NL

### G.2.1 Build OpenVPN-NL

OpenVPN-NL can be found at: `https://openvpn.fox-it.com/software.html`. Download the sources and extract into an suitable directory. Gathering profiling information requires adding profiling code to the application. To this end, add the option `-ggdb` to `CFLAGS` in the following files:

- `configure-openvpn-nl.sh`
- `configure-mbedtls.sh`
- `configure-pkcs11-helper.sh`

Now build the system by executing the `build-openvpn-nl.sh` script.

### G.2.2 Setting up OpenVPN-NL

Setting up OpenVPN requires a number of signed certificates. Follow the steps in `https://openvpn.net/community-resources/how-to/#examples` to generate all required files. For the configuration of the server and the client the sample configuration files can be used without modification. These can be found in the OpenVPN-NL source tree (`<openVPN-NL-dir>/openvpn/sample/sample-config-files/`).

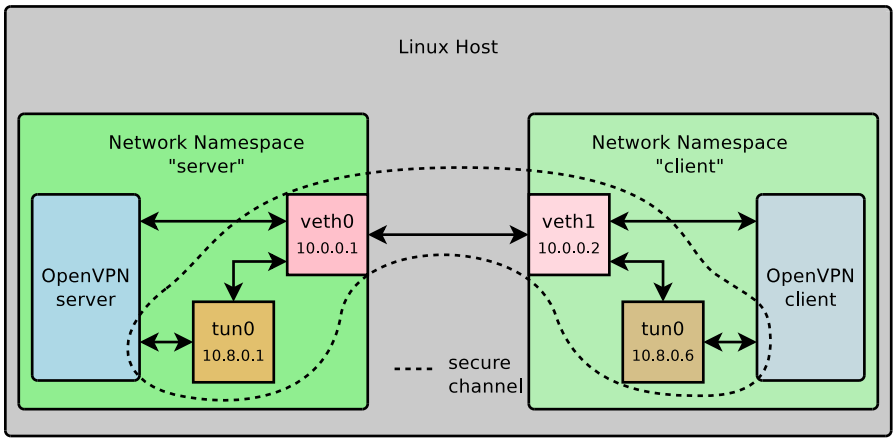



**Figure G.1:** Test setup for OpenVPN-NL profiling

## G.3 Generate Data File

The data file is generated under Linux using the `urandom` pseudo random device. This makes the actual data different on each run, but for encryption this should not give different results. Use the following command:

```
dd if=/dev/urandom of=datafile bs=1M count=1024
```

## G.4 Running the Benchmark

### G.4.1 Setting up the System

Figure G.1 (Figure 11.1, repeated here for convenience) shows the intended test setup. Setting up this system consists of a number of steps, all of which should be executed with the appropriate access rights:

1. Creating the network namespaces;
   ```
   ip netns add server
   ip netns add client
   ```

2. Creating the virtual ethernet interfaces;
   ```
   ip link add veth0 type veth peer name veth1
   ```

3. Assigning the virtual interfaces to the appropriate namespaces;
   ```
   ip link set veth0 netns server
   ip link set veth1 netns client
   ```

4. Setting up the virtual ethernet interfaces in the namespaces;
   ```
   ip netns exec server ifconfig veth0 10.0.0.1/24 up ip netns exec client ifconfig
   veth0 10.0.0.2/24 up
   ```

### G.4.2 Starting the Client and Server

The client and server need to be started in seperate terminals that should be kept open for the duration of the benchmark.

**server** In server configuration directory `ip netns exec server openvpn -config server.conf`

**client** In client configuration directory `ip netns exec client openvpn -config client.conf`

### G.4.3 Copy the File via the Secure Tunnel

The netcat program is used to copy the datafile over the secure tunnel. The receiving and sending ends should be started in separate terminal sessions that should be kept open for the duration of the benchmark.

**server** Open the waiting connection on port 1194 (default secure port of OpenVPN) on the server: `ip netns exec server nc -l -p 1194 > received_data`

**client** Send the file from the client: `ip netns exec client nc 10.0.0.1 1194 < datafile`

After copying the file has completed and all programs are closed, the output file containing the profile data can be analyzed using the `gprof` program. The results of such a run are included in Appendix H.

# Appendix H

# Profiling Data from the OpenVPN-NL Test Setup

This is the output of profiling the OpenVPN-NL test setup with `gprof` as described in Chapter 11.

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 39.69 | $7.13 \cdot 10^7$ | `mbedtls_internal_aes_encrypt` |
| 33.71 | $7.25 \cdot 10^7$ | `gcm_mult` |
| 11.54 | $1.17 \cdot 10^6$ | `mbedtls_gcm_update` |
| 2.27 | $7.25 \cdot 10^7$ | `mbedtls_cipher_update` |
| 1.20 | | `__addvsi3` |
| 1.02 | $7.13 \cdot 10^7$ | `aes_crypt_ecb_wrap` |
| 0.96 | $7.13 \cdot 10^7$ | `mbedtls_aes_crypt_ecb` |
| 0.72 | $2.35 \cdot 10^6$ | `pre_select` |
| 0.48 | $7.79 \cdot 10^5$ | `process_outgoing_link` |
| 0.48 | $7.79 \cdot 10^5$ | `encrypt_sign` |
| 0.48 | $3.94 \cdot 10^5$ | `openvpn_decrypt` |
| 0.48 | | `openvpn_main` |
| 0.42 | $7.79 \cdot 10^5$ | `openvpn_encrypt` |
| 0.36 | $1.17 \cdot 10^6$ | `mbedtls_cipher_update_ad` |
| 0.36 | $2.35 \cdot 10^6$ | `po_wait` |
| 0.33 | $2.35 \cdot 10^6$ | `io_wait_dowork` |
| 0.30 | | `mbedtls_aes_setkey_dec` |
| 0.30 | $1.17 \cdot 10^6$ | `mbedtls_gcm_starts` |
| 0.27 | $3.94 \cdot 10^5$ | `process_incoming_link_part1` |
| 0.27 | $4.69 \cdot 10^6$ | `po_ctl` |
| 0.24 | $2.35 \cdot 10^6$ | `check_tls_dowork` |
| 0.24 | $1.17 \cdot 10^6$ | `mbedtls_gcm_finish` |
| 0.24 | $1.17 \cdot 10^6$ | `mss_fixup_ipv4` |
| 0.24 | | `__addvdi3` |
| 0.21 | $3.94 \cdot 10^5$ | `packet_id_add` |
| 0.18 | $1.17 \cdot 10^6$ | `is_ipv_X` |
| 0.18 | $7.79 \cdot 10^5$ | `tls_post_encrypt` |
| 0.18 | $3.94 \cdot 10^5$ | `tls_pre_decrypt` |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.18 | $3.94 \cdot 10^5$ | process_incoming_link_part2 |
| 0.15 | $2.35 \cdot 10^6$ | update_now |
| 0.15 | $7.79 \cdot 10^5$ | read_incoming_tun |
| 0.12 | $3.13 \cdot 10^6$ | proto_is_dgram |
| 0.12 | $2.35 \cdot 10^6$ | po_reset |
| 0.12 | $2.35 \cdot 10^6$ | process_io |
| 0.12 | $1.17 \cdot 10^6$ | process_ip_header |
| 0.12 | | aes_crypt_cbc_wrap |
| 0.12 | | mbedtls_strerror |
| 0.12 | | tls_pre_decrypt_lite |
| 0.09 | $7.79 \cdot 10^5$ | packet_id_write |
| 0.09 | | ep_del |
| 0.06 | $2.35 \cdot 10^6$ | socket_set |
| 0.06 | $1.17 \cdot 10^6$ | cipher_ctx_reset |
| 0.06 | $1.17 \cdot 10^6$ | mbedtls_cipher_set_iv |
| 0.06 | $7.79 \cdot 10^5$ | cipher_ctx_final |
| 0.06 | $7.79 \cdot 10^5$ | read_tun |
| 0.06 | $7.79 \cdot 10^5$ | tls_pre_encrypt |
| 0.06 | $7.79 \cdot 10^5$ | tls_prepend_opcode_v2 |
| 0.06 | $3.94 \cdot 10^5$ | packet_id_test |
| 0.06 | $3.94 \cdot 10^5$ | cipher_ctx_final_check_tag |
| 0.06 | 2 | mss_fixup_dowork |
| 0.06 | | __subvsi3 |
| 0.06 | | ep_free |
| 0.06 | | link_socket_write_udp_posix_sendmsg |
| 0.06 | | mbedtls_internal_aes_decrypt |
| 0.06 | | show_wait_status |
| 0.06 | | tls_prepend_opcode_v1 |
| 0.03 | $2.35 \cdot 10^6$ | cipher_kt_mode_aead |
| 0.03 | $3.94 \cdot 10^5$ | crypto_check_replay |
| 0.03 | $3.94 \cdot 10^5$ | packet_id_read |
| 0.03 | $3.94 \cdot 10^5$ | process_outgoing_tun |
| 0.03 | 9 | packet_id_free |
| 0.03 | 4 | cipher_kt_mode_ofb_cfb |
| 0.03 | 1 | prng_reset_nonce |
| 0.03 | | options_string_import |
| 0.00 | $2.35 \cdot 10^6$ | cipher_ctx_get_cipher_kt |
| 0.00 | $1.95 \cdot 10^6$ | proto_is_udp |
| 0.00 | $1.17 \cdot 10^6$ | cipher_kt_tag_size |
| 0.00 | $1.17 \cdot 10^6$ | cipher_ctx_block_size |
| 0.00 | $1.17 \cdot 10^6$ | cipher_ctx_iv_length |
| 0.00 | $1.17 \cdot 10^6$ | cipher_ctx_update |
| 0.00 | $1.17 \cdot 10^6$ | cipher_ctx_update_ad |
| 0.00 | $1.17 \cdot 10^6$ | is_ipv4 |
| 0.00 | $1.17 \cdot 10^6$ | mbedtls_cipher_finish |
| 0.00 | $1.17 \cdot 10^6$ | mbedtls_cipher_reset |
| 0.00 | $7.79 \cdot 10^5$ | cipher_ctx_get_tag |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | $7.79 \cdot 10^5$ | mbedtls_cipher_write_tag |
| 0.00 | $7.79 \cdot 10^5$ | process_incoming_tun |
| 0.00 | $3.94 \cdot 10^5$ | link_socket_read_udp_posix |
| 0.00 | $3.94 \cdot 10^5$ | read_incoming_link |
| 0.00 | $3.94 \cdot 10^5$ | mbedtls_cipher_check_tag |
| 0.00 | $3.94 \cdot 10^5$ | write_tun |
| 0.00 | $1.61 \cdot 10^5$ | mpi_mul_hlp |
| 0.00 | 62,466 | mbedtls_mpi_grow |
| 0.00 | 62,192 | mbedtls_mpi_free |
| 0.00 | 58,036 | mbedtls_mpi_cmp_abs |
| 0.00 | 46,091 | mbedtls_mpi_cmp_mpi |
| 0.00 | 38,709 | mbedtls_mpi_shift_r |
| 0.00 | 38,056 | mpi_sub_hlp |
| 0.00 | 35,008 | mbedtls_mpi_sub_abs |
| 0.00 | 27,905 | mbedtls_mpi_copy |
| 0.00 | 20,742 | mbedtls_mpi_sub_mpi |
| 0.00 | 12,658 | mbedtls_mpi_bitlen |
| 0.00 | 12,235 | mbedtls_mpi_lset |
| 0.00 | 10,737 | mbedtls_mpi_mul_mpi |
| 0.00 | 8,609 | ecp_mod_p384 |
| 0.00 | 8,609 | ecp_modp |
| 0.00 | 8,193 | mbedtls_timing_hardclock |
| 0.00 | 8,057 | mbedtls_mpi_add_abs |
| 0.00 | 7,183 | mbedtls_mpi_add_mpi |
| 0.00 | 6,803 | mbedtls_mpi_safe_cond_assign |
| 0.00 | 4,590 | mbedtls_mpi_init |
| 0.00 | 3,896 | mbedtls_mpi_shift_l |
| 0.00 | 3,259 | mbedtls_mpi_cmp_int |
| 0.00 | 3,182 | char_class |
| 0.00 | 968 | env_string_equal |
| 0.00 | 956 | mbedtls_mpi_mul_int |
| 0.00 | 827 | my_debug |
| 0.00 | 798 | mbedtls_mpi_get_bit |
| 0.00 | 769 | ecp_double_jac |
| 0.00 | 636 | mbedtls_debug_print_msg |
| 0.00 | 496 | gc_malloc |
| 0.00 | 476 | buf_printf |
| 0.00 | 331 | mbedtls_internal_md5_process |
| 0.00 | 286 | mbedtls_internal_sha1_process |
| 0.00 | 222 | mbedtls_internal_sha512_process |
| 0.00 | 198 | mbedtls_internal_sha256_process |
| 0.00 | 193 | string_alloc |
| 0.00 | 188 | mbedtls_debug_print_ret |
| 0.00 | 187 | ecp_add_mixed |
| 0.00 | 179 | mbedtls_asn1_get_len |
| 0.00 | 179 | mbedtls_md5_update_ret |
| 0.00 | 177 | md5_update_wrap |
| 0.00 | 154 | alloc_buf_gc |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 152 | mbedtls_sha1_update_ret |
| 0.00 | 151 | x_gc_free |
| 0.00 | 150 | sha1_update_wrap |
| 0.00 | 149 | mbedtls_md_hmac_update |
| 0.00 | 145 | ecp_safe_invert_jac |
| 0.00 | 143 | ecp_select_comb |
| 0.00 | 134 | parse_line |
| 0.00 | 127 | hmac_ctx_update |
| 0.00 | 117 | string_mod |
| 0.00 | 112 | mbedtls_mpi_div_mpi |
| 0.00 | 111 | mbedtls_asn1_get_tag |
| 0.00 | 110 | mbedtls_md_hmac_finish |
| 0.00 | 110 | mbedtls_mpi_shrink |
| 0.00 | 110 | mbedtls_sha512_update_ret |
| 0.00 | 106 | mbedtls_md_hmac_reset |
| 0.00 | 106 | string_mod_const |
| 0.00 | 102 | mbedtls_md_get_size |
| 0.00 | 90 | sha384_update_wrap |
| 0.00 | 88 | hmac_ctx_final |
| 0.00 | 88 | hmac_ctx_reset |
| 0.00 | 87 | mbedtls_mpi_mod_int |
| 0.00 | 82 | mbedtls_md5_starts_ret |
| 0.00 | 82 | mbedtls_sha256_update_ret |
| 0.00 | 80 | md5_starts_wrap |
| 0.00 | 80 | sha224_update_wrap |
| 0.00 | 78 | mbedtls_debug_print_buf |
| 0.00 | 76 | mbedtls_md5_finish_ret |
| 0.00 | 76 | md5_finish_wrap |
| 0.00 | 70 | ecp_normalize_jac |
| 0.00 | 70 | mbedtls_sha1_starts_ret |
| 0.00 | 68 | sha1_starts_wrap |
| 0.00 | 67 | ssl_bio_read |
| 0.00 | 66 | mbedtls_sha1_finish_ret |
| 0.00 | 64 | array_mult_safe |
| 0.00 | 64 | sha1_finish_wrap |
| 0.00 | 63 | remove_env_item |
| 0.00 | 61 | env_set_add_nolock |
| 0.00 | 59 | mbedtls_ssl_fetch_input |
| 0.00 | 56 | free_buf |
| 0.00 | 54 | mbedtls_sha512_finish_ret |
| 0.00 | 54 | setenv_str_ex |
| 0.00 | 52 | env_set_add |
| 0.00 | 51 | mbedtls_sha256_finish_ret |
| 0.00 | 49 | mbedtls_sha512_starts_ret |
| 0.00 | 47 | buf_rmtail |
| 0.00 | 47 | time_string |
| 0.00 | 46 | dont_mute |
| 0.00 | 46 | msg_fp |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 46 | openvpn_snprintf |
| 0.00 | 46 | x_msg |
| 0.00 | 46 | x_msg_va |
| 0.00 | 45 | mbedtls_ssl_flush_output |
| 0.00 | 45 | sha384_finish_wrap |
| 0.00 | 44 | event_timeout_trigger |
| 0.00 | 44 | mbedtls_ssl_read_record |
| 0.00 | 44 | mbedtls_ssl_read_record_layer |
| 0.00 | 44 | sha384_starts_wrap |
| 0.00 | 43 | alloc_buf |
| 0.00 | 41 | mbedtls_sha256_starts_ret |
| 0.00 | 39 | sha256_starts_wrap |
| 0.00 | 37 | mbedtls_ssl_handshake_client_step |
| 0.00 | 37 | mbedtls_ssl_handshake_step |
| 0.00 | 37 | sha224_finish_wrap |
| 0.00 | 36 | argv_grow.constprop.3 |
| 0.00 | 36 | hmac_ctx_size |
| 0.00 | 36 | md_ctx_size |
| 0.00 | 33 | reliable_can_send |
| 0.00 | 33 | reliable_get_buf |
| 0.00 | 32 | setenv_str |
| 0.00 | 31 | tls_clear_error |
| 0.00 | 29 | reliable_get_buf_sequenced |
| 0.00 | 28 | key_state_read_plaintext |
| 0.00 | 28 | mbedtls_mpi_read_binary |
| 0.00 | 28 | mbedtls_ssl_read |
| 0.00 | 27 | reliable_get_buf_output_sequenced |
| 0.00 | 26 | is_allowed_tls_cipher |
| 0.00 | 26 | key_state_read_ciphertext |
| 0.00 | 26 | md_kt_size |
| 0.00 | 25 | mbedtls_mpi_mod_mpi |
| 0.00 | 25 | reliable_send_timeout |
| 0.00 | 25 | tls_authentication_status |
| 0.00 | 25 | tls_multi_process |
| 0.00 | 24 | external_pkcs1_sign |
| 0.00 | 23 | add_option |
| 0.00 | 23 | parse_hash_fingerprint |
| 0.00 | 22 | mbedtls_aes_setkey_enc |
| 0.00 | 22 | mbedtls_ssl_handshake |
| 0.00 | 21 | mbedtls_md_free |
| 0.00 | 21 | mbedtls_mpi_size |
| 0.00 | 20 | mbedtls_md_init |
| 0.00 | 19 | check_file_access |
| 0.00 | 19 | mbedtls_md_info_from_type |
| 0.00 | 19 | mbedtls_sha1_self_test |
| 0.00 | 19 | print_in_addr_t |
| 0.00 | 18 | clear_buf |
| 0.00 | 18 | swap_hmac |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 16 | buf_parse |
| 0.00 | 16 | buf_puts |
| 0.00 | 16 | free_key_ctx |
| 0.00 | 16 | mbedtls_asn1_get_bool |
| 0.00 | 16 | mbedtls_md |
| 0.00 | 16 | mbedtls_md_setup |
| 0.00 | 16 | mbedtls_sha256_self_test |
| 0.00 | 15 | bypass_doubledash |
| 0.00 | 15 | check_inline_file |
| 0.00 | 15 | get_cipher_name_pair |
| 0.00 | 15 | mbedtls_oid_get_x509_ext_type |
| 0.00 | 14 | mbedtls_cipher_free |
| 0.00 | 14 | mbedtls_md_hmac_starts |
| 0.00 | 14 | mbedtls_mpi_sub_int |
| 0.00 | 14 | mbedtls_sha256_ret |
| 0.00 | 14 | myrand |
| 0.00 | 14 | setenv_int |
| 0.00 | 14 | sha256_wrap |
| 0.00 | 13 | ctr_drbg_update_internal |
| 0.00 | 13 | mbedtls_md_info_from_string |
| 0.00 | 13 | mbedtls_sha512_init |
| 0.00 | 13 | md_kt_get |
| 0.00 | 12 | cipher_kt_block_size |
| 0.00 | 12 | mbedtls_sha512_free |
| 0.00 | 12 | ssl_update_checksum_start |
| 0.00 | 11 | key_state_write_plaintext_const |
| 0.00 | 11 | mbedtls_ctr_drbg_random |
| 0.00 | 11 | mbedtls_ctr_drbg_random_with_add |
| 0.00 | 11 | mbedtls_ecp_copy |
| 0.00 | 11 | mbedtls_rsa_import_raw |
| 0.00 | 11 | mbedtls_ssl_write |
| 0.00 | 10 | buf_sub |
| 0.00 | 10 | dev_type_enum |
| 0.00 | 10 | hmac_ctx_cleanup |
| 0.00 | 10 | hmac_ctx_free |
| 0.00 | 10 | hmac_ctx_init |
| 0.00 | 10 | hmac_ctx_new |
| 0.00 | 10 | is_dev_type |
| 0.00 | 10 | key_state_write_plaintext |
| 0.00 | 10 | mbedtls_asn1_get_alg |
| 0.00 | 10 | mbedtls_oid_get_attr_short_name |
| 0.00 | 10 | packet_id_reap |
| 0.00 | 10 | reliable_ack_write |
| 0.00 | 10 | write_control_auth |
| 0.00 | 9 | __pthread_atfork |
| 0.00 | 9 | argv_reset |
| 0.00 | 9 | mbedtls_ecp_point_free |
| 0.00 | 9 | mbedtls_mpi_gcd |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 9 | mbedtls_rsa_deduce_crt |
| 0.00 | 9 | mbedtls_ssl_get_ciphersuite_name |
| 0.00 | 9 | mbedtls_ssl_handle_message_type |
| 0.00 | 9 | mbedtls_ssl_parse_change_cipher_spec |
| 0.00 | 9 | md_full |
| 0.00 | 9 | pf_check_reload |
| 0.00 | 9 | plugin_defined |
| 0.00 | 9 | ssl_update_checksum_sha384 |
| 0.00 | 8 | _pkcs11h_log |
| 0.00 | 8 | free_key_ctx_bi |
| 0.00 | 8 | is_hard_reset |
| 0.00 | 8 | mbedtls_cipher_setkey |
| 0.00 | 8 | mbedtls_cipher_setup |
| 0.00 | 8 | mbedtls_md_update |
| 0.00 | 8 | mbedtls_mpi_inv_mod |
| 0.00 | 8 | mbedtls_rsa_get_len |
| 0.00 | 8 | mbedtls_ssl_write_record |
| 0.00 | 8 | mbedtls_x509_dn_gets |
| 0.00 | 8 | prng_bytes |
| 0.00 | 8 | read_control_auth |
| 0.00 | 8 | reliable_ack_read |
| 0.00 | 8 | reliable_send_purge |
| 0.00 | 8 | ssl_bio_write |
| 0.00 | 8 | x509_check_time |
| 0.00 | 8 | x509_get_current_time |
| 0.00 | 7 | _pkcs11_openvpn_log |
| 0.00 | 7 | aes_gen_tables |
| 0.00 | 7 | buffer_list_free |
| 0.00 | 7 | cipher_kt_get |
| 0.00 | 7 | cipher_kt_iv_size |
| 0.00 | 7 | compat_flag |
| 0.00 | 7 | ipchange_fmt |
| 0.00 | 7 | is_ipv6 |
| 0.00 | 7 | mbedtls_aes_init |
| 0.00 | 7 | mbedtls_cipher_info_from_string |
| 0.00 | 7 | mbedtls_debug_print_mpi |
| 0.00 | 7 | mbedtls_mpi_exp_mod |
| 0.00 | 7 | mbedtls_mpi_write_binary |
| 0.00 | 7 | mss_fixup_ipv6 |
| 0.00 | 7 | reliable_empty |
| 0.00 | 7 | string_mod_remap_name |
| 0.00 | 7 | x509_crt_find_parent_in |
| 0.00 | 6 | argv_printf_arglist |
| 0.00 | 6 | cipher_kt_key_size |
| 0.00 | 6 | cipher_kt_mode_cbc |
| 0.00 | 6 | do_setenv_x509 |
| 0.00 | 6 | get_random |
| 0.00 | 6 | init_key_type |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 6 | is_allowed_data_channel_cipher |
| 0.00 | 6 | key_state_free |
| 0.00 | 6 | key_state_rm_auth_control_file |
| 0.00 | 6 | key_state_ssl_free |
| 0.00 | 6 | mbedtls_aes_free |
| 0.00 | 6 | mbedtls_asn1_get_bitstring_null |
| 0.00 | 6 | mbedtls_base64_decode |
| 0.00 | 6 | mbedtls_gcm_crypt_and_tag |
| 0.00 | 6 | mbedtls_md5_free |
| 0.00 | 6 | mbedtls_md5_init |
| 0.00 | 6 | mbedtls_mpi_fill_random |
| 0.00 | 6 | mbedtls_sha1_free |
| 0.00 | 6 | mbedtls_sha1_init |
| 0.00 | 6 | mbedtls_ssl_config_free |
| 0.00 | 6 | mbedtls_ssl_prepare_handshake_record |
| 0.00 | 6 | mbedtls_x509_get_alg |
| 0.00 | 6 | mbedtls_x509_get_name |
| 0.00 | 6 | mbedtls_x509_get_time |
| 0.00 | 6 | md_ctx_update |
| 0.00 | 6 | reliable_ack_acknowledge_packet_id |
| 0.00 | 6 | reliable_ack_read_packet_id |
| 0.00 | 6 | reliable_can_get |
| 0.00 | 6 | reliable_mark_active_incoming |
| 0.00 | 6 | reliable_mark_active_outgoing |
| 0.00 | 6 | reliable_mark_deleted |
| 0.00 | 6 | reliable_not_replay |
| 0.00 | 6 | reliable_send |
| 0.00 | 6 | reliable_wont_break_sequentiality |
| 0.00 | 6 | sha512_starts_wrap |
| 0.00 | 6 | translate_cipher_name_to_openvpn |
| 0.00 | 6 | x509_get_uid |
| 0.00 | 5 | argv_msg |
| 0.00 | 5 | argv_printf |
| 0.00 | 5 | buf_catrunc |
| 0.00 | 5 | check_file_access_chroot |
| 0.00 | 5 | format_hex_ex |
| 0.00 | 5 | is_allowed_data_channel_digest |
| 0.00 | 5 | key_state_write_ciphertext |
| 0.00 | 5 | make_env_array |
| 0.00 | 5 | mbedtls_asn1_get_int |
| 0.00 | 5 | mbedtls_debug_print_crt |
| 0.00 | 5 | mbedtls_ecp_curve_info_from_grp_id |
| 0.00 | 5 | mbedtls_pem_read_buffer |
| 0.00 | 5 | mbedtls_pk_get_bitlen |
| 0.00 | 5 | mbedtls_rsa_check_pubkey |
| 0.00 | 5 | mbedtls_sha256_free |
| 0.00 | 5 | mbedtls_sha256_init |
| 0.00 | 5 | mbedtls_ssl_ciphersuite_from_id |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 5 | openvpn_execve |
| 0.00 | 5 | openvpn_execve_check |
| 0.00 | 5 | openvpn_getaddrinfo |
| 0.00 | 5 | platform_access |
| 0.00 | 5 | platform_system_ok |
| 0.00 | 5 | print_argv |
| 0.00 | 5 | rsa_get_bitlen |
| 0.00 | 5 | setenv_format_indexed_name.constprop.6 |
| 0.00 | 5 | setenv_route_addr |
| 0.00 | 5 | setenv_str_i |
| 0.00 | 5 | sha384_ctx_alloc |
| 0.00 | 5 | sha384_ctx_free |
| 0.00 | 4 | _pkcs11h_mem_free |
| 0.00 | 4 | _pkcs11h_mem_malloc |
| 0.00 | 4 | aes_ctx_alloc |
| 0.00 | 4 | aes_ctx_free |
| 0.00 | 4 | aes_setkey_enc_wrap |
| 0.00 | 4 | argv_new |
| 0.00 | 4 | buf_clear |
| 0.00 | 4 | buf_string_match_head_str |
| 0.00 | 4 | cert_hash_free |
| 0.00 | 4 | check_key |
| 0.00 | 4 | cipher_kt_name |
| 0.00 | 4 | crypto_adjust_frame_parameters |
| 0.00 | 4 | crypto_max_overhead |
| 0.00 | 4 | entropy_update |
| 0.00 | 4 | fixup_key |
| 0.00 | 4 | frame_finalize |
| 0.00 | 4 | gcm_aes_setkey_wrap |
| 0.00 | 4 | gcm_ctx_alloc |
| 0.00 | 4 | gcm_ctx_free |
| 0.00 | 4 | init_key_ctx |
| 0.00 | 4 | key_des_num_cblocks |
| 0.00 | 4 | mbedtls_cipher_info_from_values |
| 0.00 | 4 | mbedtls_cipher_init |
| 0.00 | 4 | mbedtls_gcm_free |
| 0.00 | 4 | mbedtls_gcm_init |
| 0.00 | 4 | mbedtls_gcm_setkey |
| 0.00 | 4 | mbedtls_md_get_name |
| 0.00 | 4 | mbedtls_mpi_write_string |
| 0.00 | 4 | mbedtls_oid_get_pk_alg |
| 0.00 | 4 | mbedtls_pk_free |
| 0.00 | 4 | mbedtls_pk_info_from_type |
| 0.00 | 4 | mbedtls_pk_setup |
| 0.00 | 4 | mbedtls_rsa_complete |
| 0.00 | 4 | mbedtls_rsa_free |
| 0.00 | 4 | mbedtls_rsa_init |
| 0.00 | 4 | mbedtls_sha512_clone |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 4 | mbedtls_sha512_ret |
| 0.00 | 4 | mbedtls_x509_serial_gets |
| 0.00 | 4 | mbedtls_x509_time_is_future |
| 0.00 | 4 | mbedtls_x509_time_is_past |
| 0.00 | 4 | md5_ctx_alloc |
| 0.00 | 4 | md5_ctx_free |
| 0.00 | 4 | md_kt_name |
| 0.00 | 4 | open_plugins |
| 0.00 | 4 | packet_id_init |
| 0.00 | 4 | pk_get_pk_alg |
| 0.00 | 4 | print_sockaddr_ex |
| 0.00 | 4 | rand_bytes |
| 0.00 | 4 | reliable_free |
| 0.00 | 4 | reliable_init |
| 0.00 | 4 | rsa_alloc_wrap |
| 0.00 | 4 | rsa_can_do |
| 0.00 | 4 | rsa_free_wrap |
| 0.00 | 4 | sha1_ctx_alloc |
| 0.00 | 4 | sha1_ctx_free |
| 0.00 | 4 | tls1_P_hash |
| 0.00 | 4 | tls_prf_generic |
| 0.00 | 4 | tls_prf_sha384 |
| 0.00 | 4 | tls_session_free |
| 0.00 | 4 | x509_get_sha256_fingerprint |
| 0.00 | 3 | _pkcs11h_threading_mutexFree |
| 0.00 | 3 | _pkcs11h_threading_mutexInit |
| 0.00 | 3 | _pkcs11h_threading_mutexLock |
| 0.00 | 3 | dev_type_string |
| 0.00 | 3 | do_init_timers |
| 0.00 | 3 | frame_print |
| 0.00 | 3 | getaddr |
| 0.00 | 3 | init_tun |
| 0.00 | 3 | init_verb_mute |
| 0.00 | 3 | ip_addr_dotted_quad_safe |
| 0.00 | 3 | ip_or_dns_addr_safe |
| 0.00 | 3 | key_direction_state_init |
| 0.00 | 3 | local_route |
| 0.00 | 3 | log_history_free_contents |
| 0.00 | 3 | log_history_init |
| 0.00 | 3 | log_history_obj_init |
| 0.00 | 3 | mbedtls_asn1_get_bitstring |
| 0.00 | 3 | mbedtls_asn1_get_mpi |
| 0.00 | 3 | mbedtls_cipher_auth_decrypt |
| 0.00 | 3 | mbedtls_cipher_auth_encrypt |
| 0.00 | 3 | mbedtls_dhm_free |
| 0.00 | 3 | mbedtls_ecp_check_pubkey |
| 0.00 | 3 | mbedtls_ecp_group_free |
| 0.00 | 3 | mbedtls_gcm_auth_decrypt |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 3 | mbedtls_oid_get_extended_key_usage |
| 0.00 | 3 | mbedtls_oid_get_oid_by_md |
| 0.00 | 3 | mbedtls_oid_get_sig_alg |
| 0.00 | 3 | mbedtls_pem_free |
| 0.00 | 3 | mbedtls_pem_init |
| 0.00 | 3 | mbedtls_pk_can_do |
| 0.00 | 3 | mbedtls_pk_get_type |
| 0.00 | 3 | mbedtls_pk_load_file |
| 0.00 | 3 | mbedtls_pk_parse_subpubkey |
| 0.00 | 3 | mbedtls_rsa_public |
| 0.00 | 3 | mbedtls_ssl_get_ciphersuite_id |
| 0.00 | 3 | mbedtls_ssl_hash_from_md_alg |
| 0.00 | 3 | mbedtls_x509_crt_free |
| 0.00 | 3 | mbedtls_x509_get_ext |
| 0.00 | 3 | mbedtls_x509_get_serial |
| 0.00 | 3 | mbedtls_x509_get_sig |
| 0.00 | 3 | mbedtls_x509_get_sig_alg |
| 0.00 | 3 | openvpn_inet_aton |
| 0.00 | 3 | pk_get_rsapubkey |
| 0.00 | 3 | rand_ctx_get |
| 0.00 | 3 | random_bytes_to_buf |
| 0.00 | 3 | read_string |
| 0.00 | 3 | rsa_rsassa_pkcs1_v15_encode |
| 0.00 | 3 | run_up_down |
| 0.00 | 3 | setenv_sockaddr |
| 0.00 | 3 | sha224_ctx_alloc |
| 0.00 | 3 | sha224_ctx_free |
| 0.00 | 3 | tls_ctx_initialised |
| 0.00 | 3 | tls_get_cipher_name_pair |
| 0.00 | 2 | backend_x509_get_issuer |
| 0.00 | 2 | backend_x509_get_serial |
| 0.00 | 2 | backend_x509_get_serial_hex |
| 0.00 | 2 | backend_x509_get_username |
| 0.00 | 2 | block_cipher_df |
| 0.00 | 2 | buf_string_compare_advance |
| 0.00 | 2 | cert_hash_remember |
| 0.00 | 2 | check_connection_established_dowork |
| 0.00 | 2 | cipher_ctx_cleanup |
| 0.00 | 2 | cipher_ctx_free |
| 0.00 | 2 | cipher_ctx_init |
| 0.00 | 2 | cipher_ctx_new |
| 0.00 | 2 | comp_add_to_extra_buffer |
| 0.00 | 2 | do_uid_gid_chroot |
| 0.00 | 2 | ecp_mul_comb |
| 0.00 | 2 | env_set_create |
| 0.00 | 2 | env_set_del |
| 0.00 | 2 | env_set_destroy |
| 0.00 | 2 | env_set_get |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 2 | frame_init_mssfix |
| 0.00 | 2 | frame_set_mtu_dynamic |
| 0.00 | 2 | init_key_ctx_bi |
| 0.00 | 2 | init_options |
| 0.00 | 2 | key_ctx_update_implicit_iv.constprop.21 |
| 0.00 | 2 | key_source_print |
| 0.00 | 2 | key_state_init |
| 0.00 | 2 | key_state_ssl_init |
| 0.00 | 2 | keydirection2ascii |
| 0.00 | 2 | log_entry_free_contents |
| 0.00 | 2 | log_history_add |
| 0.00 | 2 | mbedtls_asn1_get_sequence_of |
| 0.00 | 2 | mbedtls_debug_print_ecp |
| 0.00 | 2 | mbedtls_debug_set_threshold |
| 0.00 | 2 | mbedtls_dhm_init |
| 0.00 | 2 | mbedtls_ecdh_free |
| 0.00 | 2 | mbedtls_ecdh_init |
| 0.00 | 2 | mbedtls_ecp_check_privkey |
| 0.00 | 2 | mbedtls_ecp_grp_id_list |
| 0.00 | 2 | mbedtls_ecp_mul |
| 0.00 | 2 | mbedtls_md_finish |
| 0.00 | 2 | mbedtls_md_starts |
| 0.00 | 2 | mbedtls_oid_get_sig_alg_desc |
| 0.00 | 2 | mbedtls_pk_debug |
| 0.00 | 2 | mbedtls_pk_get_name |
| 0.00 | 2 | mbedtls_pk_verify |
| 0.00 | 2 | mbedtls_rsa_pkcs1_verify |
| 0.00 | 2 | mbedtls_rsa_rsassa_pkcs1_v15_verify |
| 0.00 | 2 | mbedtls_sha1_ret |
| 0.00 | 2 | mbedtls_ssl_conf_authmode |
| 0.00 | 2 | mbedtls_ssl_conf_ca_chain |
| 0.00 | 2 | mbedtls_ssl_conf_cbc_record_splitting |
| 0.00 | 2 | mbedtls_ssl_conf_cert_profile |
| 0.00 | 2 | mbedtls_ssl_conf_ciphersuites |
| 0.00 | 2 | mbedtls_ssl_conf_dbg |
| 0.00 | 2 | mbedtls_ssl_conf_min_version |
| 0.00 | 2 | mbedtls_ssl_conf_own_cert |
| 0.00 | 2 | mbedtls_ssl_conf_rng |
| 0.00 | 2 | mbedtls_ssl_conf_verify |
| 0.00 | 2 | mbedtls_ssl_config_defaults |
| 0.00 | 2 | mbedtls_ssl_config_init |
| 0.00 | 2 | mbedtls_ssl_free |
| 0.00 | 2 | mbedtls_ssl_get_ciphersuite |
| 0.00 | 2 | mbedtls_ssl_init |
| 0.00 | 2 | mbedtls_ssl_list_ciphersuites |
| 0.00 | 2 | mbedtls_ssl_session_free |
| 0.00 | 2 | mbedtls_ssl_set_bio |
| 0.00 | 2 | mbedtls_ssl_setup |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 2 | mbedtls_x509_crt_info |
| 0.00 | 2 | mbedtls_x509_crt_parse |
| 0.00 | 2 | mbedtls_x509_crt_parse_file |
| 0.00 | 2 | mbedtls_x509_key_size_helper |
| 0.00 | 2 | mbedtls_x509_sig_alg_gets |
| 0.00 | 2 | mpi_write_hlp |
| 0.00 | 2 | openvpn_PRF.constprop.22 |
| 0.00 | 2 | options_string |
| 0.00 | 2 | packet_id_persist_load_obj |
| 0.00 | 2 | pkcs11h_setLogHook |
| 0.00 | 2 | pkcs11h_setPINPromptHook |
| 0.00 | 2 | pkcs11h_setTokenPromptHook |
| 0.00 | 2 | pkcs11h_terminate |
| 0.00 | 2 | platform_getpid |
| 0.00 | 2 | platform_stat |
| 0.00 | 2 | pre_init_signal_catch |
| 0.00 | 2 | prng_init |
| 0.00 | 2 | proto_remote |
| 0.00 | 2 | resolve_remote |
| 0.00 | 2 | rsa_debug |
| 0.00 | 2 | rsa_verify_wrap |
| 0.00 | 2 | sanitize_control_message |
| 0.00 | 2 | session_id_random |
| 0.00 | 2 | set_check_status |
| 0.00 | 2 | set_cloexec |
| 0.00 | 2 | set_debug_level |
| 0.00 | 2 | set_mute_cutoff |
| 0.00 | 2 | set_nonblock |
| 0.00 | 2 | setenv_link_socket_actual |
| 0.00 | 2 | setenv_str_incr |
| 0.00 | 2 | signal_description |
| 0.00 | 2 | socket_adjust_frame_parameters |
| 0.00 | 2 | socket_get_sndbuf |
| 0.00 | 2 | ssl_calc_finished_tls_sha384 |
| 0.00 | 2 | ssl_calc_verify_tls_sha384 |
| 0.00 | 2 | ssl_handshake_init |
| 0.00 | 2 | ssl_write_real |
| 0.00 | 2 | tls_session_init |
| 0.00 | 2 | uninit_proxy_dowork |
| 0.00 | 2 | verify_callback |
| 0.00 | 2 | verify_cert |
| 0.00 | 2 | virtual_output_callback_func |
| 0.00 | 2 | write_string |
| 0.00 | 2 | x509_crt_check_cn |
| 0.00 | 2 | x509_get_sha1_fingerprint |
| 0.00 | 2 | x509_get_subject |
| 0.00 | 2 | x509_name_cmp |
| 0.00 | 2 | x509_profile_check_key |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 2 | x509_setenv |
| 0.00 | 2 | x_check_status |
| 0.00 | 1 | __pkcs11h_crypto_polarssl_initialize |
| 0.00 | 1 | __pkcs11h_crypto_polarssl_uninitialize |
| 0.00 | 1 | _pkcs11h_slotevent_terminate |
| 0.00 | 1 | add_route |
| 0.00 | 1 | add_route_to_option_list |
| 0.00 | 1 | add_routes |
| 0.00 | 1 | alloc_connection_entry |
| 0.00 | 1 | apply_push_options |
| 0.00 | 1 | argv_printf_cat |
| 0.00 | 1 | ascii2af |
| 0.00 | 1 | ascii2keydirection |
| 0.00 | 1 | ascii2proto |
| 0.00 | 1 | auth_retry_get |
| 0.00 | 1 | buf_null_terminate |
| 0.00 | 1 | buffer_list_peek |
| 0.00 | 1 | check_addr_clash |
| 0.00 | 1 | check_incoming_control_channel_dowork |
| 0.00 | 1 | check_replay_consistency |
| 0.00 | 1 | check_subnet_conflict |
| 0.00 | 1 | check_version_3_17_plus |
| 0.00 | 1 | close_instance |
| 0.00 | 1 | close_management |
| 0.00 | 1 | close_tun |
| 0.00 | 1 | comp_generate_peer_info_string |
| 0.00 | 1 | context_clear_2 |
| 0.00 | 1 | context_clear_all_except_first_time |
| 0.00 | 1 | context_gc_free |
| 0.00 | 1 | context_init_1 |
| 0.00 | 1 | create_socket |
| 0.00 | 1 | crypto_init_lib |
| 0.00 | 1 | crypto_read_openvpn_key |
| 0.00 | 1 | crypto_uninit_lib |
| 0.00 | 1 | delayed_auth_pass_purge |
| 0.00 | 1 | delete_routes |
| 0.00 | 1 | do_close_tun |
| 0.00 | 1 | do_close_tun_simple |
| 0.00 | 1 | do_deferred_options |
| 0.00 | 1 | do_genkey |
| 0.00 | 1 | do_ifconfig |
| 0.00 | 1 | do_ifconfig_setenv |
| 0.00 | 1 | do_init_crypto_tls |
| 0.00 | 1 | do_init_traffic_shaper |
| 0.00 | 1 | do_open_tun |
| 0.00 | 1 | do_persist_tuntap |
| 0.00 | 1 | do_route |
| 0.00 | 1 | do_test_crypto |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 1 | do_up |
| 0.00 | 1 | entropy_gather_internal |
| 0.00 | 1 | env_set_inherit |
| 0.00 | 1 | error_reset |
| 0.00 | 1 | event_set_init |
| 0.00 | 1 | event_set_init_simple |
| 0.00 | 1 | frame_finalize_options |
| 0.00 | 1 | frame_subtract_extra |
| 0.00 | 1 | free_ssl_lib |
| 0.00 | 1 | gc_addspecial |
| 0.00 | 1 | gc_freeaddrinfo_callback |
| 0.00 | 1 | generate_key_expansion |
| 0.00 | 1 | get_cached_dns_entry |
| 0.00 | 1 | get_debug_level |
| 0.00 | 1 | get_default_gateway |
| 0.00 | 1 | get_special_addr |
| 0.00 | 1 | get_ssl_library_version |
| 0.00 | 1 | hash_func |
| 0.00 | 1 | havege_fill |
| 0.00 | 1 | helper_client_server |
| 0.00 | 1 | helper_keepalive |
| 0.00 | 1 | helper_tcp_nodelay |
| 0.00 | 1 | in_extra_reset |
| 0.00 | 1 | incoming_push_message |
| 0.00 | 1 | init_context_buffers |
| 0.00 | 1 | init_crypto_pre |
| 0.00 | 1 | init_instance |
| 0.00 | 1 | init_instance_handle_signals |
| 0.00 | 1 | init_management |
| 0.00 | 1 | init_management_callback_p2p |
| 0.00 | 1 | init_options_dev |
| 0.00 | 1 | init_plugins |
| 0.00 | 1 | init_query_passwords |
| 0.00 | 1 | init_route_list |
| 0.00 | 1 | init_ssl |
| 0.00 | 1 | init_ssl_lib |
| 0.00 | 1 | init_static |
| 0.00 | 1 | init_tun_post |
| 0.00 | 1 | initialization_sequence_completed |
| 0.00 | 1 | interval_init |
| 0.00 | 1 | is_allowed_prng_digest |
| 0.00 | 1 | key2_print |
| 0.00 | 1 | key_schedule_free |
| 0.00 | 1 | link_socket_close |
| 0.00 | 1 | link_socket_connection_initiated |
| 0.00 | 1 | link_socket_current_remote |
| 0.00 | 1 | link_socket_init_phase1 |
| 0.00 | 1 | link_socket_init_phase2 |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 1 | link_socket_new |
| 0.00 | 1 | man_connection_close |
| 0.00 | 1 | man_def_auth_set_client_reason |
| 0.00 | 1 | man_output_list_push_finalize |
| 0.00 | 1 | management_close |
| 0.00 | 1 | management_init |
| 0.00 | 1 | mbed_log_func_line |
| 0.00 | 1 | mbedtls_cipher_info_from_type |
| 0.00 | 1 | mbedtls_ctr_drbg_init |
| 0.00 | 1 | mbedtls_ctr_drbg_reseed |
| 0.00 | 1 | mbedtls_ctr_drbg_seed |
| 0.00 | 1 | mbedtls_ctr_drbg_seed_entropy_len |
| 0.00 | 1 | mbedtls_ctr_drbg_update |
| 0.00 | 1 | mbedtls_ecdh_calc_secret |
| 0.00 | 1 | mbedtls_ecdh_compute_shared |
| 0.00 | 1 | mbedtls_ecdh_make_public |
| 0.00 | 1 | mbedtls_ecdh_read_params |
| 0.00 | 1 | mbedtls_ecp_gen_keypair |
| 0.00 | 1 | mbedtls_ecp_gen_keypair_base |
| 0.00 | 1 | mbedtls_ecp_group_load |
| 0.00 | 1 | mbedtls_ecp_is_zero |
| 0.00 | 1 | mbedtls_ecp_point_init |
| 0.00 | 1 | mbedtls_ecp_point_read_binary |
| 0.00 | 1 | mbedtls_ecp_point_write_binary |
| 0.00 | 1 | mbedtls_ecp_tls_read_group |
| 0.00 | 1 | mbedtls_ecp_tls_read_point |
| 0.00 | 1 | mbedtls_ecp_tls_write_point |
| 0.00 | 1 | mbedtls_entropy_func |
| 0.00 | 1 | mbedtls_entropy_init |
| 0.00 | 1 | mbedtls_entropy_update_manual |
| 0.00 | 1 | mbedtls_hardclock_poll |
| 0.00 | 1 | mbedtls_havege_init |
| 0.00 | 1 | mbedtls_havege_poll |
| 0.00 | 1 | mbedtls_havege_random |
| 0.00 | 1 | mbedtls_pk_check_pair |
| 0.00 | 1 | mbedtls_pk_parse_key |
| 0.00 | 1 | mbedtls_pk_parse_keyfile |
| 0.00 | 1 | mbedtls_pk_sign |
| 0.00 | 1 | mbedtls_pk_verify_ext |
| 0.00 | 1 | mbedtls_platform_entropy_poll |
| 0.00 | 1 | mbedtls_rsa_check_privkey |
| 0.00 | 1 | mbedtls_rsa_check_pub_priv |
| 0.00 | 1 | mbedtls_rsa_pkcs1_sign |
| 0.00 | 1 | mbedtls_rsa_private |
| 0.00 | 1 | mbedtls_rsa_rsassa_pkcs1_v15_sign |
| 0.00 | 1 | mbedtls_rsa_validate_crt |
| 0.00 | 1 | mbedtls_rsa_validate_params |
| 0.00 | 1 | mbedtls_ssl_check_cert_usage |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 1 | mbedtls_ssl_check_curve |
| 0.00 | 1 | mbedtls_ssl_check_sig_hash |
| 0.00 | 1 | mbedtls_ssl_derive_keys |
| 0.00 | 1 | mbedtls_ssl_get_ciphersuite_sig_pk_alg |
| 0.00 | 1 | mbedtls_ssl_get_key_exchange_md_tls1_2 |
| 0.00 | 1 | mbedtls_ssl_get_peer_cert |
| 0.00 | 1 | mbedtls_ssl_get_version |
| 0.00 | 1 | mbedtls_ssl_handshake_wrapup |
| 0.00 | 1 | mbedtls_ssl_md_alg_from_hash |
| 0.00 | 1 | mbedtls_ssl_optimize_checksum |
| 0.00 | 1 | mbedtls_ssl_parse_certificate |
| 0.00 | 1 | mbedtls_ssl_parse_finished |
| 0.00 | 1 | mbedtls_ssl_pk_alg_from_sig |
| 0.00 | 1 | mbedtls_ssl_read_version |
| 0.00 | 1 | mbedtls_ssl_sig_from_pk |
| 0.00 | 1 | mbedtls_ssl_write_certificate |
| 0.00 | 1 | mbedtls_ssl_write_change_cipher_spec |
| 0.00 | 1 | mbedtls_ssl_write_finished |
| 0.00 | 1 | mbedtls_ssl_write_version |
| 0.00 | 1 | mbedtls_version_get_number |
| 0.00 | 1 | mbedtls_x509_crl_free |
| 0.00 | 1 | mbedtls_x509_crt_check_extended_key_usage |
| 0.00 | 1 | mbedtls_x509_crt_check_key_usage |
| 0.00 | 1 | mbedtls_x509_crt_init |
| 0.00 | 1 | mbedtls_x509_crt_parse_der |
| 0.00 | 1 | mbedtls_x509_crt_verify_with_profile |
| 0.00 | 1 | md_ctx_cleanup |
| 0.00 | 1 | md_ctx_final |
| 0.00 | 1 | md_ctx_free |
| 0.00 | 1 | md_ctx_init |
| 0.00 | 1 | md_ctx_new |
| 0.00 | 1 | mstats_close |
| 0.00 | 1 | new_route_option_list |
| 0.00 | 1 | next_connection_entry |
| 0.00 | 1 | notnull |
| 0.00 | 1 | open_management |
| 0.00 | 1 | open_tun |
| 0.00 | 1 | openvpn_exit |
| 0.00 | 1 | options_cmp_equal |
| 0.00 | 1 | options_cmp_equal_safe |
| 0.00 | 1 | options_postprocess |
| 0.00 | 1 | options_postprocess_verify_ce |
| 0.00 | 1 | options_string_extract_option |
| 0.00 | 1 | packet_id_persist_close |
| 0.00 | 1 | packet_id_persist_init |
| 0.00 | 1 | packet_id_persist_save |
| 0.00 | 1 | parse_argv |
| 0.00 | 1 | parse_topology |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 1 | pf_destroy_context |
| 0.00 | 1 | pk_parse_key_pkcs1_der |
| 0.00 | 1 | pk_parse_key_pkcs8_unencrypted_der |
| 0.00 | 1 | pkcs11_initialize |
| 0.00 | 1 | pkcs11_terminate |
| 0.00 | 1 | pkcs11h_engine_setCrypto |
| 0.00 | 1 | pkcs11h_engine_setSystem |
| 0.00 | 1 | pkcs11h_initialize |
| 0.00 | 1 | pkcs11h_setForkMode |
| 0.00 | 1 | pkcs11h_setLogLevel |
| 0.00 | 1 | pkcs11h_setPINCachePeriod |
| 0.00 | 1 | pkcs11h_setProtectedAuthentication |
| 0.00 | 1 | platform_fopen |
| 0.00 | 1 | platform_group_get |
| 0.00 | 1 | platform_nice |
| 0.00 | 1 | platform_open |
| 0.00 | 1 | platform_user_get |
| 0.00 | 1 | plugin_abort |
| 0.00 | 1 | po_free |
| 0.00 | 1 | possibly_become_daemon |
| 0.00 | 1 | post_init_signal_catch |
| 0.00 | 1 | pre_pull_restore |
| 0.00 | 1 | pre_setup |
| 0.00 | 1 | print_details |
| 0.00 | 1 | print_link_socket_actual |
| 0.00 | 1 | print_openssl_info |
| 0.00 | 1 | print_signal |
| 0.00 | 1 | prng_uninit |
| 0.00 | 1 | process_incoming_push_msg |
| 0.00 | 1 | process_signal |
| 0.00 | 1 | proto2ascii |
| 0.00 | 1 | proto_is_net |
| 0.00 | 1 | pull_permission_mask |
| 0.00 | 1 | purge_user_pass |
| 0.00 | 1 | rand_update_manual |
| 0.00 | 1 | read_config_file |
| 0.00 | 1 | read_key_file |
| 0.00 | 1 | reliable_ack_adjust_frame_parameters |
| 0.00 | 1 | reliable_schedule_now |
| 0.00 | 1 | remap_signal |
| 0.00 | 1 | reset_check_status |
| 0.00 | 1 | reset_coarse_timers |
| 0.00 | 1 | rsa_check_pair_wrap |
| 0.00 | 1 | rsa_sign_wrap |
| 0.00 | 1 | send_control_channel_string |
| 0.00 | 1 | send_push_request |
| 0.00 | 1 | session_id_print |
| 0.00 | 1 | set_actual_address |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 1 | set_mtu_discover_type |
| 0.00 | 1 | setenv_connection_entry |
| 0.00 | 1 | setenv_routes |
| 0.00 | 1 | setenv_settings |
| 0.00 | 1 | setenv_unsigned |
| 0.00 | 1 | show_connection_entry |
| 0.00 | 1 | show_library_versions |
| 0.00 | 1 | show_settings |
| 0.00 | 1 | signal_restart_status |
| 0.00 | 1 | socket_get_rcvbuf |
| 0.00 | 1 | status_open |
| 0.00 | 1 | tls_adjust_frame_parameters |
| 0.00 | 1 | tls_check_ncp_cipher_list |
| 0.00 | 1 | tls_common_name |
| 0.00 | 1 | tls_ctx_check_cert_time |
| 0.00 | 1 | tls_ctx_client_new |
| 0.00 | 1 | tls_ctx_free |
| 0.00 | 1 | tls_ctx_load_ca |
| 0.00 | 1 | tls_ctx_load_cert_file |
| 0.00 | 1 | tls_ctx_load_priv_file |
| 0.00 | 1 | tls_ctx_personalise_random |
| 0.00 | 1 | tls_ctx_restrict_ciphers |
| 0.00 | 1 | tls_ctx_set_cert_profile |
| 0.00 | 1 | tls_ctx_set_options |
| 0.00 | 1 | tls_free_lib |
| 0.00 | 1 | tls_init_lib |
| 0.00 | 1 | tls_item_in_cipher_list |
| 0.00 | 1 | tls_multi_free |
| 0.00 | 1 | tls_multi_init |
| 0.00 | 1 | tls_multi_init_finalize |
| 0.00 | 1 | tls_multi_init_set_options |
| 0.00 | 1 | tls_peer_info_ncp_ver |
| 0.00 | 1 | tls_rec_payload |
| 0.00 | 1 | tls_send_payload |
| 0.00 | 1 | tls_session_generate_data_channel_keys |
| 0.00 | 1 | tls_session_update_crypto_params |
| 0.00 | 1 | tls_version_to_major_minor |
| 0.00 | 1 | tls_x509_clear_env |
| 0.00 | 1 | translate_cipher_name_from_openvpn |
| 0.00 | 1 | tun_abort |
| 0.00 | 1 | uninit_management_callback |
| 0.00 | 1 | uninit_options |
| 0.00 | 1 | uninit_static |
| 0.00 | 1 | verify_final_auth_checks |
| 0.00 | 1 | verify_fix_key2 |
| 0.00 | 1 | x509_crt_check_signature |
| 0.00 | 1 | x509_verify_cert_eku |
| 0.00 | 1 | x509_verify_cert_ku |

| Relative Execution Time [%] | Num of Calls | Function |
|---|---|---|
| 0.00 | 1 | `x_gc_freespecial` |