1. Szymon WRÓBEL, 2. Krzysztof KĘPA

Wroclaw University of Science and Technology
ORCID: 1. 0009-0009-0112-4092; 2. 0000-0003-0702-9570

doi:10.15199/48.2025.02.46

FIRV: A language-based control flow integrity protection for RISC-V architectures

Abstract. *Side-channel and fault-injection attacks using e.g. EM/laser pulse, power glitching are a major concern in the context of embedded systems, IoT devices, and cloud security. The Software-implemented Hardware-fault Tolerance (SIHFT) countermeasures are the main approach to hardening the systems built using Commercial Off-the-Shelf (COTS) components, in which modification of hardware is not feasible. The research presented in this article is focused on an open-source solution to language-based, compile-time application of SIHFT countermeasures. The proof-of-concept implementation is based on the LLVM compiler framework and demonstrates using Rust language frontend, allowing the use of other compiler features, like optimisation passes and support for multiple target platforms. The results of the research are publicly available in GitHub repository.*

Streszczenie. *Ataki kanałem pobocznym I wstrzykiwanie błędów przy użyciu impulsu elektromagnetycznego/laserowego, lub usterki zasilania, stanowią poważny problem w kontekście systemów wbudowanych, urządzeń IoT i bezpieczeństwa w chmurze. Implementowane programowo środki zaradcze Hardware-Fault Tolerance (SIHFT) są głównym podejściem do utwardzania systemów zbudowanych przy użyciu komercyjnych komponentów, w których modyfikacja sprzętu nie jest możliwa. Badania przedstawione w tym artykule koncentrują się na otwarto-zródłowym językowym rozwiązaniu stosowanym w czasie kompilacji. Implementacja prototypu jest oparta na projekcie modularnego kompilatora LLVM i demonstruje użycie kompilatora dla języka Rust, co pozwala na korzystanie z innych funkcji kompilatora, takich jak przebiegi optymalizacyjne i obsługa wielu platform docelowych. Wyniki badań są publicznie dostępne w repozytorium GitHub. (**FIRV: oparta na języku ochrona integralności przepływu sterowania dla architektur RISC-V**)*

Keywords: compilation, fault injection, language countermeasures, LLVM, RISC-V, Rust.
Słowa kluczowe: kompilacja, wstrzykiwanie błędów, środki zaradcze języka, LLVM, RISC-V, Rust.

Introduction

Modern Operational Technology (OT) systems like industry and avionics controllers, IOT devices, and embedded systems in general, are increasingly complex solutions of software layers forming a software stack, and many interconnected physical components, the hardware. While software-level attacks focus on exploiting vulnerabilities and inconsistencies in the abstract logic model of computing the hardware-level attacks target directly the very physical structure of the device to observe or tamper with the information that is stored in form of energy in registers or passed through the device internal buses or interfaces.

The side-channel attacks like EM/laser pulse or power glitching are a family of hardware attacks, focusing on the exploitation of the hardware by observing and/or altering the physical state of the device, rather than exploiting the algorithmic or hardware vulnerabilities. The side-channel analysis can be performed using a variety of methods, including timing [1], power consumption [2], electromagnetic emanation [3], or temperature characteristics [4].

The fault-injection attacks (also known as glitching), are a type of active side-channel attacks, where through altering the state of the device, an adversary is attempting to alter the outcome of the program, for example: skipping key verification, accepting unsigned code, modifying data. Fault injection can be performed using many methods, from electromagnetic impulse [5], to alteration of the power supply voltage [6] or laser light [7].

Side-channel analysis and fault-injection attacks are a major concern in several branches of software and hardware security, including embedded systems, IoT device development, and cloud platforms.

The Software-implemented Hardware-fault Tolerance (SIHFT) countermeasures are the preferred, and often the only possible method of hardening the device against fault-injection attacks, when using the Commercial Off-the-Shelf (COTS) components, where modification of the underlying hardware is not feasible [8].

The countermeasures currently used in the industry very often require manual application on the source or binary level. The former approach entails a need for disabling the compiler optimisations or insertions of artificial code fragments, effectively preventing compilers from optimising away the countermeasures, while the latter requires a deep understanding of target architecture and results in a high coupling, likely leading to increasingly complex development process and requires highly trained and specialised software engineers [9], [10].

The main goal of this work is to improve the firmware development workflow by providing a proof-of-concept solution for applying the countermeasures by specifying a simple attribute/pragma in the source code, which the compiler can then use to apply required countermeasures.

Side-channel attacks and fault injection

Side-channel attacks are a class of attacks focused on the exploitation of physical properties of the target system, allowing for leaking some information on the conducted computations through unintended channels (electromagnetic field, temperature, power consumption, etc.) rather than through direct vulnerability of the system.

Side-channel analysis stems from research on the security of cryptographic protocols, where leaking even a small amount of information can lead the adversary to gain advantage and can compromise the security of the system. It is important to note that side-channel analysis focuses mostly on the particular implementation of the protocol, usually on a specific hardware.

Fault-injection (FI) attacks are separate, but closely related branch of attacks. Their core component is to focus on exploiting the physical characteristics of the hardware, for example, altering the power supply voltage or changing temperature, to introduce a change in the system. It is usually extremely hard to explain the exact process leading to the change, but the observed behavioural change can be similar to, for example, instruction skips or data alteration. Fault injection models can also be effective when designing solutions for harsh environments, like aerospace, defence, or critical infrastructure (e.g. nuclear power plants), where the increased radiation levels can lead to spontaneous faults.

The main difference between side-channel analysis and fault-injection attacks is that the former is focused on the passive or semi-passive observation of the operation of the system, while the latter consists of performing very invasive actions on the system. Some resources classify fault injection attacks as a branch of side-channel analysis [11], [12].

The history of side-channel analysis comes from the papers by Kocher [1] and Kocher et al. [2], where timing attacks and differential power analysis methods were used to retrieve partial information or a whole secret key based on side-channel knowledge. An extensive survey of fault injection attacks and software and hardware countermeasures was presented in the report by Bar-El et al. [13].

The primary sources of side-channel information come from both unintentional sources (electromagnetic emanation, power consumption, timing) and intentional sources (memory footprint, sensor usage, data consumption) [12].

Originally, side-channel attacks relied on physical access to the analysed device, but the recent growth of multi-tenant cloud architectures opened a new area of research, where side-channel information can be leaked despite the logical separation of the computations, for example, power-consumption analysis in the FPGA accelerators [14] or remote exploitation of power management mechanisms [15].

RISC-V architecture and LLVM compiler framework

The research presented in this work is focused on, but not limited to, RISC-V architecture. The choice of RISC-V as the architecture of focus is motivated by the increasing popularity and impact of RISC-V-based processors and relatively small amount of research regarding side-channel and fault-injection attacks in RISC-V architectures. A description of the architecture and the design considerations can be found in [16]. The full specification of the ISA can be found in [17]. The approach selected to demonstrate results of the research was QEMU [18], due to prior knowledge of the platform by the author, broad documentation, and community support.

LLVM is a project containing tools and libraries that can be used to build highly optimised and robust compilers and language toolchains [19], [20]. The backbone of the project is LLVM IR, an intermediate representation that is designed to be a portable, high-level assembly [21].

The standard architecture of the compiler consists of three main blocks: frontend, optimiser, and backend. The frontend performs tasks connected with the syntax and semantics of the input language: lexical analysis, syntax analysis (parsing) and semantic analysis. The optimiser step can perform machine-independent optimisations, for example, constant propagation or dead code elimination. The backend step consists of target-specific optimisations, like instruction scheduling, replacing instructions with faster ones [22].

Compilers built using the LLVM provide a high degree of freedom and robustness with regard to input language and target architectures, thanks to the decoupling of the three aforementioned steps. The language-specific frontends, that emit LLVM IR, can use the same optimisation pipeline. The target-specific backend, can then process the IR and perform machine-specific operations. The flow diagram of the process is shown in figure 1.

Related work on fault-injection countermeasures

The two main categories of fault-injection countermeasures are fault resilience, stemming from the research on the reliability and robustness of software and hardware solutions, and fault detection, which is used mostly in security research.

The main approach to the fault resiliency problem is the introduction of redundancy. The most common is the instruction-level redundancy presented in, for example, the paper by Moro et al. [23], where the instruction-level redundancy approach for ARM Thumb-2 was formally verified. An important notion in the context of instruction redundancy is the idempotency of the instruction, meaning executing the instruction twice leads to the same machine state. The instruction-level redundancy approach has been presented and improved in various papers, e.g. Barry et al. proposed in [24] a LLVM-based solution of application of the redundancy, improving on the assembly-based method of Moro et al., and Chen et al. [25] proposed usage of the SIMD instructions to achieve the redundancy with reduced performance overhead.

In article [26], Schilling et al. proposed a variety of countermeasures in the context of control-flow integrity and stateful CFI branching. The article mentions implementation based on the LLVM project, but no publicly available implementation is available.

The article by Oh et al. [27] introduced Error Detection by Duplicated Instructions (EDDI), where the instruction-level redundancy is augmented with the result checks. In the article by Reis et al. [28], there has been presented the usage countermeasures (EDDI, Error Correcting Codes, Extended Control Flow checks).

The article by Richter-Brockmann et al. [29] presents a universal framework for verification of the (primarily hardware) fault-injection countermeasures.

Finally, a very recent report by the German Federal Office for Information Security [10] provides a survey on the topic of hardware fault attacks on microcontrollers, classification, and survey on the software and hardware countermeasures.

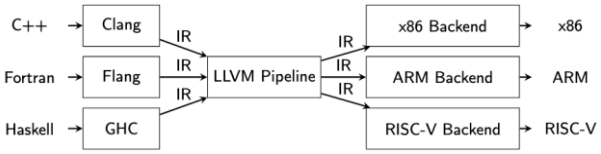

Fig.1. Frontend-optimiser-backend LLVM architecture

Currently implemented solutions

The manual implementations of the countermeasures have been applied in various projects. Some of the open-source examples include:

* wolfSSL - the procedures regarding verification of signatures used in the boot process are manually hardened by using platform-specific assembly [30],

* OpenTitan [31] - implementation of platform-specific "laundering" and barrier primitives, which prevent the compiler from tracking the connections of the value to other values in context, and therefore optimising the hardened code away,

* MCUboot [32] - implementation of various countermeasures, e.g. global fragile call counter, conversion of integers to pair (*value*, *value xor mask*).

One of the automated approaches to the application of countermeasures via binary rewriting was presented in the paper by Kiaei et al. [9], where the compiled application is disassembled and using the faulter-patcher framework the countermeasures are applied to places, where the faulter has managed to alter the outcome of the program.

The language and intermediate-level approaches have been implemented in various projects, one of the most widely known is the Cogito project [33] that is built on top of the LLVM toolchain. The research on the project led to publication of many papers (e.g. mentioned earlier work by

Barry et al. [24]), but the source code and resulting tools are not released.

One of the most complete and publicly available tools is CompaSeC project [34], in which various countermeasures (i.a. EDDI, SWIFT, RASM [35]) can be applied by the LLVM passes. When compared to the solution presented here the main differences is focus on the low-level (machine-dependent) transformations and the use of external configuration files or command-line arguments to specify the scope of application of the countermeasures.

Fault model, solution design and implementation

An important design consideration, that needs to be carefully thought through, is the threat/fault model. The internal mechanisms of the fault injection attacks are very difficult to research and understand [36], [13], [37], but outcomes observed can be, i.e. single/multiple instruction skip, bit flip, random byte, or permanent fault.

The fault model assumed in this project is the single instruction skip, as it's one of the most often observed outcomes in electromagnetic fault-injection and voltage glitching attacks [23].

The solution presented in this work consists of two main parts: modification of the LLVM and extending the Rust frontend to allow specifying the components requiring hardening. The LLVM modification can also be used as a standalone tool, e.g., when using another front-end language.

LLVM Pipeline - to successfully implement the automatic countermeasure instrumentation, it is imperative to select a proper place to insert the transformation. The general architecture of the LLVM backend from previous section could be split into lower-level components. Figure 2 presents is a more granular view of the pipeline. The middle-end consists of the machine-independent optimizations and code generation unit, which emits MIR -- the machine intermediate representation, which is a high-level SSA representation of the target assembly. The backend is built from the machine-dependent optimization and the final assembly emission unit.

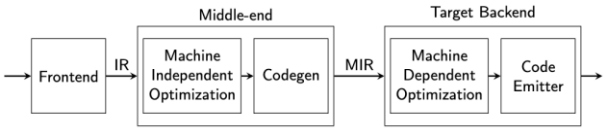

Fig.2. LLVM pipeline

LLVM Passes - the structure of the pipeline can be inspected on an even lower level (figure 3). The optimization passes, are general optimizations, that are part of the LLVM Optimization Pipeline (*opt* tool). The code related to those passes can be found in Analysis, Transforms folders. The

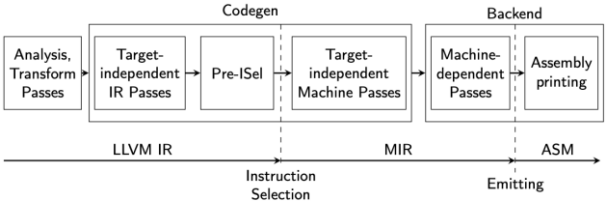

Fig.3. LLVM Pass within pipeline

CodeGen part of the pipeline consists of the target-independent IR passes, the *preISel* passes and the target-independent machine passes (working on MIR). The instruction selection is the pipeline step, where the LLVM IR is transformed into the MIR. After that, the machine-dependent passes are executed (e.g. register allocation). The last step of the pipeline is the materialization of the MIR into the target machine assembly.

Modification of Rust - majority of the changes required for the Rust language are connected with updating the underlying LLVM project, as well as allowing propagation of the required attributes.

Rust language allows using two kinds of attribute: inner and outer. The former applies to the element it was defined inside of (function, crate), while the latter is applied to the next element. Example of attribute usage is presented in figure 4.

The approach implemented in this work provides function-level redundancy. The motivations for selecting

```
1 // Inner attribute applies to whole crate
2 #![no_std]
3
4 // Outer attribute applies to next function
5 #[inline]
6 fn f(a: u32): u32 {
7     ...
8 }
```

Fig.4. Example attribute usage

such countermeasure are: simplicity of implementation, medium additional memory (2 return value slots and double redundancy of input parameters, that could be further reduced) and computational overhead (double execution of computed function and several load/store and comparison instructions).

The countermeasure works by inserting the Prologue, Interlude, Epilogue, and Failure blocks and clone of the function body. The general diagram for the countermeasure operation is presented in figure 5. The Prologue block contains an allocation of the memory for the return values and the function parameters for the first execution. The copying of function parameters is necessary, as functions can modify their parameters. After the allocation steps, the parameters are then copied to the proper slots, and the execution of the function body can start. The Interlude block is responsible for storing the return value of the first execution, and allocation of memory for parameters for the second execution, and copying them. The Epilogue block handles the return value from the second execution, retrieves the result of the first computation, and only if those values

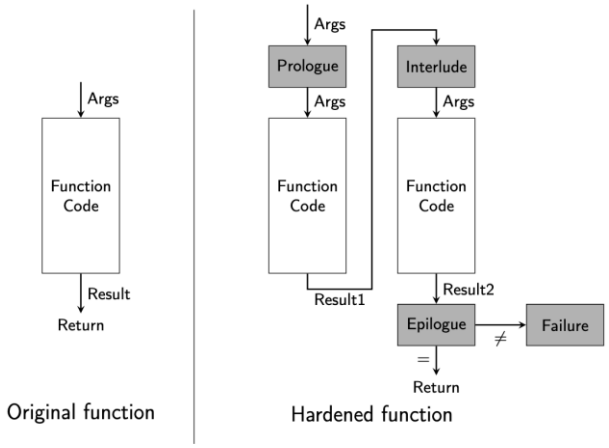

Fig.5. Application of the proposed hardening countermeasure

match, returns, or else the Failure block is executed, which is the block responsible for handling the error.

The important step prior to implementation is to select the list of assumptions. The most important ones are:

* Hardening only pure functions -- the hardened function cannot have any side effects. The situation of applying the hardening to impure function is an undefined behaviour.
* Supported types -- Due to the limited scope of the project, only support for integer and floating-point types is implemented.

The process of implementation requires making changes in a few places across the project. All the paths mentioned later are relative to the *llvm* catalogue in the root folder, which contains the code of the core LLVM tools and libraries.

Modifying the LLVM IR - the first step of implementation is the modification of the LLVM IR, which allows specifying the attribute on a function. The selected attribute name was *firv_harden* (The project was named **FIRV**, **F**ault **I**njection in **R**ISC-**V**).

Introduction of the attribute - the LLVM attributes are defined in the *include/llvm/IR/Attributes.td* file. The following definition was added:

```
def FirvHarden: EnumAttr<"firv_harden", [FnAttr]>;
```

The meaning of the statement is: define *FirvHarden* to derive from the EnumAttr subclass; the tag is *firv_harden* and it's a function attribute.

Adding the bitcode identifier - it's required to specify the bitcode identifier for the attribute. It's done in the *include/llvm/Bitcode/LLVMBitCodes.h* header file. The *AttributeKindCodes* enum should be extended with the desired attribute id:

```
enum AttributeKindCodes {
  ...
  ATTR_KIND_FIRV_HARDEN = 100,
...};
```

Handling the miscellaneous operations - the former changes entail a need to implement the handling cases in a couple of spots: bitcode reader, bitcode writer, and code extractor.

The required changes are: adding a case for the attribute in *BitcodeReader.cpp*, in the *getAttrFromCode* function, updating the *getAttrKindEncoding* function of *BitcodeWriter.cpp* module, and changing the function *constructFunction* in the CodeExtractor.cpp module. The last function is responsible for the extraction of the code fragments and placing them in a new function. The required change comes from the need to specify whether the attribute can or should not be added to the newly extracted function. In the case of this attribute, it's imperative to propagate it.

Attaching the Pass - before implementing the pass, it is required to select the type of the pass (Module or Function) and its placement in the pipeline. According to the LLVM documentation Function passes are required to satisfy 2 conditions:
* Optimization (or transformations) are organized globally, i.e. one function at a time.
* The pass doesn't cause the addition or removal of any functions in the module.

As both of those conditions are satisfied in the case of the transformation being added, hence the FunctionPass can be used.

Selection of the place in the pipeline offers following choices: optimization pass (Analysis/Transform), CodeGen pass, Target-specific pass.

Creating the header file - the first step of attaching the pass is the creation of the header file for the pass -- *FirvHarden.h*. The header is the place, where usually the declarations of the functions and classes are taking place.

```
1 #[no_mangle]
2 extern "C"
3 fn sw_f(a: i32, b: i32) -> i32 {
4     let x = a + b;
5     let y = a - b;
6
7     return x * y;
8 }
```

Fig.6. Countermeasure evaluation source file

The minimal declaration of a functional FunctionPass requires specification of the pass ID, constructor and an overridden *runOnFunction* method.

Pass initializations - to use countermeasure pass, it is required to provide hooks allowing the creation and initialization of the pass. The former is declared in the Passes.h header:

```
FunctionPass *createFirvHardenPass();
```

The latter is declared in the *InitializePasses.h* header:

```
void initializeFirvHardenPass(PassRegistry&);
```

Pass registration - before the pass can be used in LLVM, it needs to be registered in the LLVM's pass manager. The chosen pipeline placement is in the *preISel* block (see figure 3). Due to the pass not being an optimization pass, but exactly the opposite, it's executed in the latest moment possible. This allows all previous optimizations to work, while ensuring the countermeasure instrumentation is not affected by them. The pass registrations for the CodeGen step are happening in *Target\-Pass\-Config*. This class is the base class for the target-specific pass config, and the targets can override the specified methods (customary, still calling the base version of the method, but not required to). The pass is added in the *addISelPrepare* method, which is executed just before the Instruction Selection process. Due to the gradual introduction of the new Pass Manager [38], it's advised to prepare for the eventual transition. Similarly to the legacy pass manager, the pass is registered in the ISel Prepare step, which is the last step before the Instruction Selection step.

Implementing the Pass - the implementation of the pass is done in the *FirvHarden.cpp*. For the file to be compiled and linked into the library it needs to be added to *CMakeLists.txt*. The first part of the file consists of the required definitions, initialisation, and creation of the supplemental classes.

The main method for the pass *runOnFunction()* is presented in Appendix 1. The return value of that function indicates if the function has been modified in any way. The checks in lines 2 and 7 verify if the transformation should be performed on the function -- is the attribute present and if the function return type is supported, respectively.

After that, the actual transformation steps are executed: adding store/load for arguments (line 13), cloning of the function body (line 17), creation of prologue (line 21), creation of fail block (line 23) and return pad block (line 24), creation of epilogue (line 26) and interlude (line 32), and finally replacing the return instructions to storing the result in the respective slot and jump to next block (lines 34, 35). Full implementation of the pass can be found in the repository *firv-llvm-project* (see section *Code Repositories*).

Results

The test function used to evaluate the countermeasure efficacy is presented in the figure 6. The test cases were prepared by creating a base file (*src/res.rs*), and then creating a copy (*src/harden.rs*) with the *firv_harden* attribute

applied to the function. The code can be executed using the *make rust-asm* command, with the *RUSTC* and *SRC* arguments passed (*make rust-asm RUSTC=... SRC=...*).

The comparison of the resulting assembly of unhardened and hardened cases are presented in figure 7 and figure 8. Lines 2-5 are responsible for setting up the stack frame. In lines 6-13, the arguments for the first execution are prepared, and lines 14-17 are responsible for execution of the computation and storing the result. Lines 18-25 are responsible for setting up the arguments for the second execution, performed in lines 26-29. In lines 30-32, the results of the computations are compared. The block *.LBB1_1* (line 33) is the failure block, consisting of the instruction *unimpl*, causing the hardware interrupt. Lines 37-39 perform secondary comparison of the results. Lines 40-46 are responsible for preparing the return value and cleaning up the stack.

```
1 sw_f:
2     add a2, a0, a1
3     subw a0, a0, a1
4     mulw a0, a2, a0
5     ret
```

Fig.7. Generic (non-hardened) test function

```
 1 sw_f:
 2     addi sp, sp, -32
 3     sd ra, 24(sp)
 4     sd s0, 16(sp)
 5     addi s0, sp, 32
 6     mv a2, sp
 7     addi sp, a2, -16
 8     sw a0, -16(a2)
 9     lw a2, -16(a2)
10     mv a3, sp
11     addi sp, a3, -16
12     sw a1, -16(a3)
13     lw a3, -16(a3)
14     add a4, a3, a2
15     subw a2, a2, a3
16     mul a2, a4, a2
17     sw a2, -20(s0)
18     mv a2, sp
19     addi sp, a2, -16
20     sw a0, -16(a2)
21     lw a0, -16(a2)
22     mv a2, sp
23     addi sp, a2, -16
24     sw a1, -16(a2)
25     lw a1, -16(a2)
26     add a2, a1, a0
27     subw a0, a0, a1
28     mul a0, a2, a0
29     sw a0, -24(s0)
30     lw a0, -20(s0)
31     lw a1, -24(s0)
32     beq a0, a1, .LBB1_
33 .LBB1_1:
34     unimp
35     unimp
36 .LBB1_2:
37     lw a0, -20(s0)
38     lw a1, -24(s0)
39     bne a0, a1, .LBB1_
40     lw a0, -20(s0)
41     lw a1, -24(s0)
42     addi sp, s0, -32
43     ld ra, 24(sp)
44     ld s0, 16(sp)
45     addi sp, sp, 32
46     ret
```

Fig.8. Hardened test function

Code repositories - The code and implementation of the proposed solution can be found in the FIRV project public GitHub repository [39] which is a fork of the official LLVM project with the changes described in this chapter. The solution was also incorporated in the fork of the Rust language. Rust project uses its specific LLVM fork. All above and the supporting code: e.g. the scripts for building and running the RISC-V code in the QEMU simulator are also placed in the public GitHub repository of the FIRV project.

Conclusions and future work

This project demonstrates a proof-of-concept automated countermeasure to fault-injection attacks by applying the theoretical protection method to a widely used compiler toolchain. The research is mainly focused on the RISC-V architecture, but because of the relative similarity of the other RISC architectures, the results of the research can be applied to other hardware targets like x86 or ARM.

The application of the Software-Implemented Hardware Fault Tolerance (SIHFT) countermeasures is a very influential approach in hardening devices against fault-injection attacks (like EM/laser pulse, power glitching), especially in the case of COTS devices, where the modification of the hardware device is not feasible.

The proposed approach of function-level redundancy is a countermeasure allowing for the detection of a single fault. Function-level redundancy is a high-level solution, in the sense that it is not tied to specific platform details, so the proposed approach could be used (perhaps with minor adjustments) in other architectures like ARM or MIPS, or even in a CISC architecture like x86. The successfully implemented solution provides a general framework for the introduction of additional fault injection countermeasures.

The implementation, supporting files, and scripts have been published in the GitHub repositories and are made public which allows the implementation of other countermeasures in the future and presents an opportunity for further development of the project.

In this section, there are presented the topics that show future work could address the following research paths aiding in further development of the FIRV project; the first path to focus on was the integration with the frontends of the compilers. The most important parts of the hardening are happening in the optimisation pipeline and target-specific backend, while the frontend code is only responsible for passing the attributes in a format that will be understood by the LLVM pipeline. The second path is surveying the topic under different fault models, e.g. multiple skips or bit-flipping. Another related avenue of research, could be the implementation of additional side-channel attack protection mechanisms, for example, hardening against timing attacks, advanced CFI protection methods, or instruction-level redundancy methods. The goal of this work was to provide a proof-of-concept of the implementation of instrumentation for the selected solution and to provide a survey on the approach and possible pitfalls in implementing such solutions.

Last, but not least, an important feature for potential industrial usage would be implementing support for complex return types, like structures or tuples. Such types are widely used in languages like Rust or C++.

***Authors**: mgr inż. Szymon Wróbel, Wydział Informatyki i Telekomunikacji, Katedra Podstaw Informatyki, ul. Janiszewskiego 11/17, 50-372 Wrocław, E-mail: Szymon.Wrobel@pwr.edu.pl;*
dr inż. Krzysztof Kępa, Politechnika Wrocławska, Wydział Informatyki i Telekomunikacji, Katedra Telekomunikacji i Teleinformatyki, ul. Janiszewskiego 11/17, 50-372 Wrocław, E-mail: Krzysztof.Kepa@pwr.edu.pl;.

REFERENCES

[1] Kocher P.C., Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems, Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96, (1996), 104–113
[2] Kocher P., Jaffe J., Jun B., Differential power analysis, Wiener, M.(ed.) Advances in Cryptology - CRYPTO' 99, (1999), 388–397
[3] Genkin D., et al., Ecdsa key extraction from mobile devices via nonintrusive physical side channels, Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS'16 (2016), 1626–1638
[4] Kim T., Shin Y., Thermalbleed: A practical thermal side-channel attack, IEEE Access 10, (2022), 25718–25731
[5] Kühnapfel N., et al., Em-fault it yourself: Building a replicable emfi setup for desktop and server hardware, In 2022 IEEE Physical Assurance and Inspection of Electronics (PAINE), (2022), 1-7
[6] Gomina K., et al., Power supply glitch attacks: Design and evaluation of detection circuits, in 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), (2014) 136–141
[7] Krachenfels T., et al., Evaluation of low-cost thermal laser stimulation for data extraction and key readout, J. Hardw. Syst. Secur. 4(1) (2020), 24–33
[8] Solanki S., Kaur M., Design and verification of fault tolerance ip core using sihft technique, In 2017 International Conference on Current Trends in Computer, Electrical, Electronics and Communication (CTCEEC), (2017), 860–863

[9] Kiaei P., Breunesse C.B., Ahmadi M., Schaumont P., Woudenberg J.v., Rewrite to reinforce: Rewriting the binary to apply countermeasures against fault injection, 58th ACM/IEEE Design Automation Conference (DAC), (2021), 319–324
[10] BSI, A study on hardware attacks against microcontrollers, Tech. rep., BSI (2023)
[11] Standaert F.X., Introduction to Side-Channel Attacks, Springer US, (2010), 27–42
[12] Spreitzer R., et al., Systematic classification of side-channel attacks: A case study for mobile devices, IEEE Communications Surveys & Tutorials, 20(1) (2018), 465–488
[13] Bar-El H., et al., The sorcerer's apprentice guide to fault attacks, Proc. of the IEEE, 94(2) (2006), 370–382
[14] Gravellier J., et al., High-speed ring oscillator based sensors for remote side-channel attacks on fpgas, In 2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig), (2019), 1–8
[15] Tang A., Sethumadhavan S., Stolfo S.J.: CLKSCREW: exposing the perils of security-oblivious energy management, In Kirda, E., Ristenpart, T. (eds.) 26th USENIX Security Symposium, USENIX Security 2017, USENIX Association (2017), 1057–1074
[16] Patterson D.A., Hennessy J.L., Computer Organization and Design RISC-V Edition: The Hardware Software Interface, Morgan Kaufmann Publishers Inc., (2017)
[17] Waterman A., Asanović K., The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213. Tech. rep., RISC-V Foundation, (2019)
[18] QEMU project, source code repository, https://github.com/qemu/qemu (accessed 2023).
[19] Apple, developer tools (snapshot from 23.04.2011). https://web.archive.org/web/20110423095129/https://developer.apple.com/technologies/tools/
[20] LLVM users, https://llvm.org/Users.html, (2022)
[21] LLVM project. https://www.llvm.org/, (2022)
[22] Aho A.V., et al., Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., (2006)
[23] Moro N., et al., Formal verification of a software countermeasure against instruction skip attacks, Journal of Cryptographic Engineering, 4(3) (2014), 145–156
[24] Barry T., Couroussé D., Robisson B., Compilation of a countermeasure against instruction-skip fault attacks, In Proceedings of the Third Workshop on Cryptography and Security in Computing Systems, CS2'16, Association for Computing Machinery (2016), 1–6
[25] Chen Z., et al., A compiler approach to mitigate fault attacks via enhanced simdization, In 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC) (2017), 57–64
[26] Schilling R., Werner M., Mangard S., Securing conditional branches in the presence of fault attacks, In 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). (2018), 1586–1591
[27] Oh N., Shirvani P., McCluskey E., Error detection by duplicated instructions in super-scalar processors, IEEE Transactions on Reliability, 51(1) (2002), 63–75
[28] Reis G., et al., Swift: software implemented fault tolerance, In International Symposium on Code Generation and Optimization, (2005), 243–254
[29] Richter-Brockmann J., et al., Fiver - robust verification of countermeasures against fault injections, IACR Trans. on Cryptographic Hardware and Embedded Sys., (2021), 447–473
[30] wolfSSL: Secure boot and glitching attacks, https://www.wolfssl.com/secure-boot-glitching-attacks/, (2022)
[31] lowRISC, OpenTitan, code repository https://github.com/lowRISC/opentitan, (accessed 2023)
[32] https://github.com/mcu-tools/mcuboot, (accessed 2023)
[33] COGITO, project COGITO ANR-13-INSE-0006-01, http://www.cogito-anr.fr, (accessed 2023)
[34] Geier J., et al., CompaSeC: A compiler-assisted security countermeasure to address instruction skip fault attacks on RISC-V, In 2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC), (2023), 1–7
[35] Vankeirsbilck J., et al., Random additive signature monitoring for control flow error detection, IEEE Transactions on Reliability 66(4) (2017), 1178–1192
[36] Dumont M., Lisart M., Maurine P., Electromagnetic fault injection: How faults occur, In 2019 Workshop on Fault Diagnosis and Tolerance in Cryptography (FTDC), (2019), 9-16
[37] Breier J., Hou X., How practical are fault injection attacks, really? IEEE Access, 10 (2022), 113122–113130
[38] Eubanks A., The New Pass Manager, https://blog.llvm.org/posts/2021-03-26-the-new-pass-manager/, (2021)
[39] Wróbel Sz., Project FIRV, (2023), https://github.com/firv-comp

Appendix 1: Main function of the implemented pass

```
1 bool FirvHarden::runOnFunction(Function &Fn) {
2   if (! Fn.hasFnAttribute(Attribute::FirvHarden ) ) {
3         return false;
4   }
5
6   Type * RetType = Fn.getReturnType();
7   if (! isH arden ingSupportedForType (RetType)) {
8         errs () << " Firv ␣ Hardening ␣ is ␣ not ␣ supported ␣
          for ␣type ␣ " << * RetType << " \ n ";
9
10         return false;
11  }
12
13  StoreArgsAndLoad(Fn) ;
14
15  std::vector<BasicBlock*> OriginalBBs ;
16  std::vector<BasicBlock*> ClonedBBs ;
17  CloneBasicBlocks(Fn, OriginalBBs, ClonedBBs);
18
19  AllocaInst * FirvAI1 = nullptr;
20  AllocaInst * FirvAI2 = nullptr;
21  CreateFirvPrologue(Fn, FirvAI1, FirvAI2) ;
22
23  BasicBlock * FailBB = CreateFailBB(Fn);
24  BasicBlock * ReturnBB = CreateReturnBB(Fn,FirvAI1, FirvAI2);
25
26  auto EpilogueBB = CreateFirvEpilogue(Fn, FirvAI1, FirvAI2,
          FailBB, ReturnBB) ;
27
28  if (!EpilogueBB) {
29        return false;
30  }
31
32  auto InterludeBB = CreateFirvInterlude(Fn , ClonedBBs) ;
33
34  ReplaceReturns (OriginalBBs, FirvAI1, InterludeBB);
35  ReplaceReturns (ClonedBBs, FirvAI2, EpilogueBB);
36
37  return true;
38 }
```