UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR IT-SICHERHEIT

# MAMBO-V: A RISC-V Port of MAMBO for the MicroWalk Framework

*MAMBO-V: Eine RISC-V Portierung von MAMBO für das MicroWalk Framework*

**Bachelorarbeit**

im Rahmen des Studiengangs
**IT-Sicherheit**
der Universität zu Lübeck

vorgelegt von
**Christopher Peredy**

ausgegeben und betreut von
**Prof. Dr.-Ing. Thomas Eisenbarth**

mit Unterstützung von
Jan Wichelmann,
Florian Sieck

Lübeck, den 11. Dezember 2021

## Abstract

Side channel attacks are an active threat for cryptographic programs. Especially, microarchitectural side channels can be remotely exploited like cache attacks have shown in the past. Software is usually hardened against these side channels using a constant time implementation, but this is an error-prone task. MicroWalk is a binary analysis framework that can locate non-constant time behavior in x86 binaries by comparing execution traces. Traces are originally generated using Pin, a DBI tool for x86 only.
RISC-V is a new free and open source ISA and rapidly gaining momentum in the industry. For binary analysis of corresponding binaries we propose MAMBO-V, a port of the low overhead DBI tool MAMBO for the RISC-V architecture. We explain optimization techniques that were ported to reduce the performance overhead. Additionally, a plugin for MAMBO-V was developed to generate traces for external constant-time analysis with MicroWalk. With this new set of tools we were able to verify the existence of a known side channel leakage in base64 decoding routines in OpenSSL on RISC-V.

## Zusammenfassung

Seitenkanalangriffe stellen eine aktive Bedrohung für kryptographische Programme dar. Besonders mikroarchitekturelle Seitenkanäle, welche ohne physischen Zugang ausgenutzt werden können, sind betroffen. Um dagegen vorzugehen wird Software meist mit einer constant-time Implementierung gehärtet, doch dies ist ein fehleranfälliger Prozess. MicroWalk ist ein Tool zur Binäranalyse, welche dazu eingesetzt werden kann, einen Seitenkanal in compilierten x86 Programmen aufzuspüren. Dafür werden sogenannte traces miteinander verglichen, welche durch Instrumentierung mit einem DBI-Tool erzeugt werden können.
RISC-V ist eine neue freie Architektur mit neuem Befehlssatz, welche mittlerweile auf viel Interesse in der Industrie und der Forschung stößt. Für die Analyse entsprechender Binärdateien stellen wir MAMBO-V, ein RISC-V Port des performanten DBI-Tools MAMBO, vor. Dabei gehen wir auf verschiedene Optimierungstechniken ein, um auch MAMBO-V performant zu gestalten. Zusätzlich haben wir eine Erweiterung für unser Tool entwickelt, mit der wir traces erzeugen können und auf constant-time Verhalten analysieren können. Mit diesen neuen Tools finden wir dann auch einen Seitenkanal in base64 Umwandlungsfunktionen von OpenSSL auf RISC-V.

## Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Hilfsmittel angefertigt zu haben.

Lübeck, 11. Dezember 2021

## Acknowledgements

An dieser Stelle möchte ich die Gelegenheit nutzen, um mich in erster Linie bei Jan Wichelmann und Florian Sieck für die Betreuung meiner Arbeit zu bedanken. Ihr habt mich immer unterstützt. Ich habe mich bei euch sehr gut aufgehoben gefühlt, wodurch mir die Arbeit viel Spaß bereitet hat. Ich möchte mich auch bei Prof. Eisenbarth und dem ITS für die Unterstützung und die Möglichkeit bedanken, dass ich dieses Thema bearbeiten konnte. Danke auch für die Bereitstellung des Entwicklerboards. Vielen Dank auch an meine Freunde und Kommilitonen, für eure Hilfe!

# Contents

# 1 Introduction

Side channel leakages in software evolving from the underlying microarchitecture are called microarchitectural side channels. They include the exploitation of physical behavior, such as power consumption or caches, of a device. The MicroWalk framework [Wichelmann et al., 2018] is a binary analysis tool to detect microarchitectural leakages in compiled binaries. It focuses on remotely exploitable side channels exposed by the microarchitecture such as memory or cache access patterns as well as the control flow itself. Software can be hardened against microarchitectural side channels using constant-time programming techniques, which means using microarchitectural resources in a secret-independent fashion. To evaluate a binary, MicroWalk collects traces of memory accesses and control flow changes from multiple runs of a program with different inputs and compares them. In case of a non-constant time behavior, the results are input dependent differences in the system state. An attacker might then be able to learn what potential secret input led to what system state. We use the analysis with MicroWalk to locate microarchitectural leakages and evaluate their severity.
Because leakages occur at binary level, the analysis is heavily specialized for specific architectures. MicroWalk currently only supports x86 binaries, but we want to be able to analyze binaries for other architectures like RISC-V as well.

RISC-V is an Instruction Set Architecture (ISA) which is increasing in popularity over the last years. It is open and free, which makes it interesting for companies in the embedded industry who want to develop custom silicon to fit their needs. But the ISA is not limited to embedded devices. Due to the modular design, RISC-V cores can also be implemented in high performance processors for general purpose computers or servers. However, the use of RISC-V in security critical areas requires a secure hardware implementation and also secure software.
In our work, we concentrate on microarchitectural side channels and leakages introduced by input dependencies. Source code compiled for x86 may not leak, while the same code compiled for RISC-V could leak information. Therefore, our approach is to use binary analysis to check for input dependent behavior.

MicroWalk generates the traces using Pin [Luk et al., 2005] which is only available for IA-32 and x86-64 platforms. Instead of porting Pin to RISC-V we decided to port an-

other Dynamic Binary Instrumentation (DBI) tool MAMBO [Gorgovan et al., 2016] mainly because it is open source and lightweight. We propose MAMBO-V, a MAMBO port for RISC-V (RV64GC) platforms.
MAMBO [Gorgovan et al., 2016] is a DBI tool developed for ARM. It is a highly optimized and lightweight DBI engine that can run most typical Linux programs with little overhead. Because MAMBO was developed for ARM specifically, it offers good optimizations for RISC-like architectures, since ARM is considered one of them. Furthermore, ARM and especially ARM 64-bit share many similarities with RISC-V, so the port of MAMBO does not require huge architectural changes to the DBI tool.

A performance evaluation of the current MAMBO-V implementation shows, that it struggles with some heavy system call workloads. On the other hand, workloads that are similar to cryptographic functions perform very well. MAMBO-V includes two optimization techniques that also speed up repetitive workloads. For RISC-V binary analysis with MicroWalk we developed a MAMBO-V plugin, that generates traces and outputs them in the exact same format as the Pin tool does. This approach does not require any adjustments of MicroWalk to support the traces by MAMBO-V.
We analyzed OpenSSL key decoding functions with MAMBO-V and MicroWalk, and found two functions which leak information about the decoded (secret) key.

## 1.1 Our Contributions

- **MAMBO-V:** A low overhead DBI framework for RISC-V.
- **Tracer:** A MAMBO-V plugin to generate execution traces with MAMBO-V in a MicroWalk compliant format.
- Improvements in translation of atomic sequences in MAMBO and MAMBO-V.
- Confirmation of a known leakage in base64 key decoding functions in OpenSSL binary on RISC-V.

# 2 Background

## 2.1 Instruction Set Architecture

An ISA is a low level hardware abstraction layer, that defines one of the most important interfaces between hardware and software [Patterson and Hennessy, 2017] [Asanović and Patterson, 2014]. The interface usually consists of an instruction set, available registers, memory constraints like code and data aligning and many more definitions. Hardware vendors can design microarchitectures which implement an ISA, so software, which is compliant to the same ISA, can be executed on this chip and behaves as expected by the programmer. There can be multiple implementations of one ISA but all are able to run identical software. A result is, that software must not be developed for one specific chip or device anymore, it can be developed for a platform.

## 2.2 RISC-V

RISC-V is a new open and extensible ISA started 2010 by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman at UC Berkley [Waterman et al., 2011]. Since 2019 RISC-V is owned by a non-profit organization named RISC-V International. It mainly consists of various members from large tech companies like Google, Huawei and Western Digital to a large number of chipmakers and OEMs like the Raspberry Pi Foundation, Nvidia and NXP just to name a few that participate into the development of the RISC-V ISA [RISC-V International, n.d.].

### 2.2.1 License

The RISC-V ISA is published under the BSD open source license and free to use. In contrast to proprietary ISAs like ARM (containing A32, T32 and A64), no expensive and potentially restrictive licenses are required to build a compatible System on a Chip (SoC). In case of ARM, just a few big companies have a license to actually design new ARM cores, most others are restricted to use ARMs pre-designed cores which drastically reduces flexibility. Asanović argued:

> While instruction set architectures (ISAs) may be proprietary for historical or business reasons, there is no good *technical* reason for the lack of free, open ISAs[.] [Asanović and Patterson, 2014, P. 1]

Just like open standards and open source software, an open ISA, one of the most important interfaces, can revolutionize the industry. This license model would enable a real market of processor designs which could lead to greater innovation via competition, fewer errors and lower cost in development resulting in more affordable hardware [Asanović and Patterson, 2014]. The openness is one of RISC-V's key advantages and the main reason for the great worldwide interest [RISC-V International, n.d.].

### 2.2.2 Technical Overview

The RISC-V architecture design aims to combine the needs of Internet of Things (IoT) (mostly small cheap devices with network capabilities), personal mobile devices (Smartphones, Tablets) and Warehouse-Scale Computers (WSCs) all in one ISA [Asanović and Patterson, 2014] [Waterman, 2016]. A base-plus-extension ISA was designed to reach that goal. All implementations are required to support the base ISA called RV32I. It is ratified since 2019 and specifies that a RISC-V core must have at least 32 32-bit general purpose and an additional program counter register like shown in figure 2.1. It also specifies 40 instructions a processor must be able to execute. RV32I mainly consists of load and store instructions, unconditional and conditional branches, logical and a few basic arithmetic instructions like additions and shifts. To support 64-bit registers, the optional standard extension RV64I can be added, which increases the instruction count by 15 to a total of 55. It adds instruction that explicitly operate on 32-bit values[1] in 64-bit registers.
All base instructions have a fixed length of 32 bit. For hardware integer multiplications, floating-point arithmetic or other hardware features, additional extensions can be implemented in a microarchitecture. Table 2.1 summarizes the ratified extensions of the RISC-V specification version 20191213 [RISC-V International, 2019], all supporting a register width of 32-bit and 64-bit.

A base-plus-extension ISA like this allows chip designers to customize their design to their needs and given specification. Small IoT systems may only need RV32IMAC compliant cores like the SiFive E31 core for a typical microcontroller. General purpose cores for personal computers and mobile devices like smartphones may additionally include floating-point arithmetic and the Zicsr and Zifencei extensions. The small base ISA there-

[1]For example `ADDW` is added which only operates on the lower 32 bits of the registers producing a signed 32-bit result.

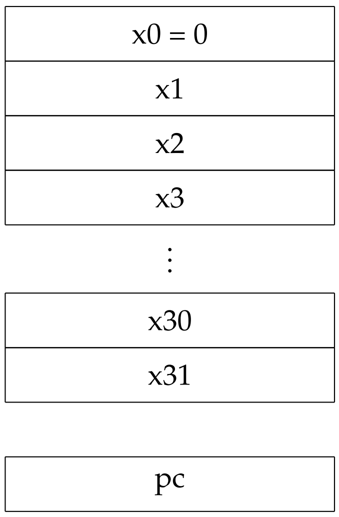


Figure 2.1: RISC-V general purpose integer registers and the program counter. Register `x0` is hardwired to zero.

fore gives designers a high degree of freedom to build efficient cores while still being compliant to the RISC-V standards for portability. A dedicated ISA specification describes the RISC-V privileged architecture and provides machine mode (most privileged mode), supervisor and hypervisor level instructions and contains features such as privilege levels and virtual memory to run operating systems [RISC-V International, 2021b].
The base-plus-extension approach contributes to the design goal of an ISA suitable for nearly any computing device. Generally, design decisions were made in a way to avoid over-architecturing. Portions of the ISA should not benefit single implementations in expense to others [Waterman, 2016].

Toward the end of 2021, over 40 new extensions were ratified [McMahon, 2021] to provide further specialization possibilities for RISC-V cores. Among them is a set of cryptographic extensions [RISC-V International, 2021a] introducing instructions for cryptographic algorithms such as AES, SHA2, SM4 and SM3. These extensions open up the possibility to build cryptographic accelerators within RISC-V cores. The `Zkt` extension guaranties, that certain already available instructions are implemented in a way, to have data-independent latency to eliminate observable timing related side channels. Power related side channels or cache related microarchitectural side channels for example are not addressed explicitly. Another newly ratified extension is the RISC-V "V" Vector Extension [RISC-V International, 2021c]. It adds Single-Input-Multiple-Data (SIMD) instructions for accelerating vector and matrix operations to speed up for example graphics and AI applications.
Standard extensions like these still follow the RISC-V theme with adding a low amount

| Extension | Description |
|---|---|
| I | Base integer instruction set |
| M | Standard extension for integer multiplication |
| A | Standard extension for atomic instructions |
| F | Standard extension for single-precision floating-point |
| D | Standard extension for double-precision floating-point |
| Q | Standard extension for quad-precision floating-point |
| C | Standard extension for compressed 16-bit instructions |
| Zicsr | Standard extensions for control and status register instructions |
| Zifencei | Instruction fetch fence |

Table 2.1: Table of ratified standard extension in RISC-V ISA version 20191213 [RISC-V International, 2019]

of lightweight instructions with high scalability [Waterman, 2016] [RISC-V International, 2021a], so they can be implemented in small wearables with area-efficient cores as well as in powerful computing devices such as servers.

### 2.2.3 RISC-V and ARM

Due to the novelty of RISC-V, the ISA can learn from previous mistakes in other RISC designs. One example is the initial Alpha ISA leaving out too much by not having byte and half-word memory manipulation instructions. So byte and half-word manipulation take multiple instructions slowing down certain applications and emulations. An example for including too much are the shift options for ARM instructions like `ADD` (register-shifted register) combining an addition with a register shift operation to one operand. Instructions like this increase the complexity of the pipeline but only achieve an advantage in code size on certain operations. RISC-V is designed with these issues in mind, so it has byte and half-word load and stores and only dedicated shift instructions as a compromise to higher average code density and simpler pipelines.

ARM is also considered a Reduced Instruction Set Computer (RISC) architecture. Although, the encoding of ARM instructions is mostly different to RISC-V and the instructions itself often significantly differ as well, both ISAs have a few higher level characteristics in common. For example, both are load-store architectures, so only designated `LOAD` and `STORE` instructions access memory directly while arithmetic instructions only operate on CPU registers. An exception are some atomic read-modify-write instructions included

```
RV64I:
    BEQ     x10, x11, offset        ; branch if equal

A64:
    CMP     x0, x1                  ; compare and update Z flag
    B.EQ    #offset                 ; branch if Z flag set
```

Listing 2.1: Comparison of conditional branch between RV64I and A64

in both instruction sets[2].
ARM and RISC-V instructions are either 32 bit (A32, A64, RISC-V base and most standard extensions) or 16 bit (T32, RV32C, RV64C) long. A64 is a true *fixed*-length ISA while RISC-V extensions like the "C" extension can relax this constraint on RISC-V platforms.

But there is also a fundamental difference between ARM and RISC-V effecting especially conditional branches. The RISC-V base ISA and all current standard extensions do not make use of a flag register or Condition Code Register (CCR). For example, arithmetic instructions do not propagate sign or zero flags. Consequently, as shown in listing 2.1, a conditional branch instruction on RISC-V contains a jump and also a comparison. This reduces code size without increasing the number of instructions, but it increases the complexity of the conditional branches and reduces the jump range to $\pm 4$ KiB [McMahon, 2021]. Conditional jumps not comparing to zero need two separate instructions on ARM, a compare and a jump.

To sum it up, there are similarities and differences between both architectures. They share more similarities than compared to a Complex Instruction Set Computer (CISC) architecture like x86 but RISC-V is not just a newer version of ARM. As shown, it follows other concepts and goals having the potential to revolutionize the industry.

## 2.3 Dynamic Binary Instrumentation and MAMBO

Dynamic binary instrumentation, is a form of modifying binary code of an application during runtime. Advantages compared to static binary instrumentation are, that only executed code is instrumented, and the binary is not required to be modified in place. This avoids recalculating branch offsets and dealing with overwritten instructions. An advantage for the static variant is that it is not necessary to execute the binary, so it can run platform independently, while DBI requires to be executed on the target platform.

MAMBO [Gorgovan et al., 2016] is a DBI tool developed for AArch32 and AArch64. In

[2]On RISC-V atomic instructions come with the standard extension for atomic instructions "A"

contrast to popular DBI tools like Pin [Luk et al., 2005], DynamoRIO [Bruening et al., 2012], and Valgrind [Nethercote and Seward, 2007], which are initially designed for the x86 architecture and were later extended to support ARM, MAMBO was originally designed and developed for ARM. ARM is a RISC architecture and very different to the x86, so optimizations of a DBI engine regarding x86 may have no positive or even a negative effect on RISC platforms.

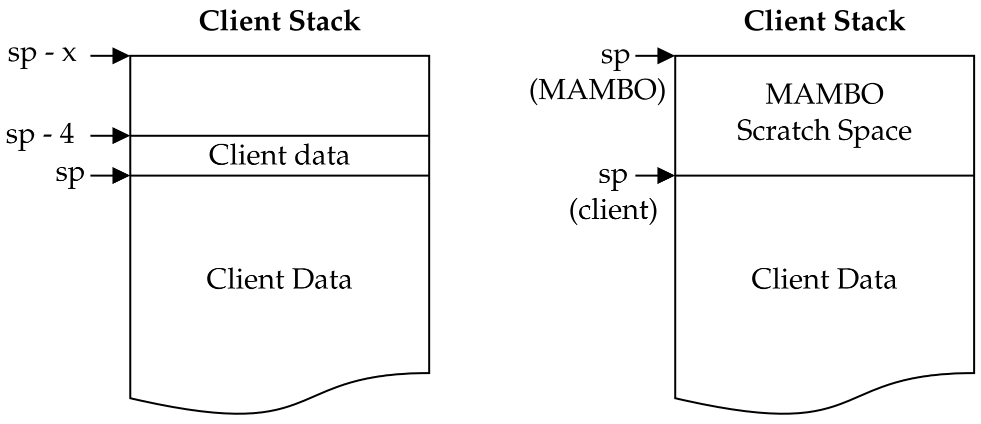
Most current existing DBI engines introduce a high execution overhead, making binary analysis on smaller embedded platforms more of a challenge than necessary due to limited performance and memory available. On SPEC CPU2006 and PARSEC 3.0 benchmarks, MAMBO introduces just small overheads [Gorgovan et al., 2016].

Main features of MAMBO are:

- ARM (AArch32 and AArch64) support
- Low translation and execution overhead
- Behavioral transparency
- Plugin interface
- Small codebase

### 2.3.1 Behavioral Transparency

MAMBO is designed to be able to correctly execute typical workflows. This means, that MAMBO can only run binaries which fully follow the platforms Application Binary Interface (ABI). This is the case for compiler generated binaries without inline assembly and code that does not rely on undefined behaviors. Applications that do not hold to these constraints may still work with MAMBO under certain circumstances but are not expected to execute correctly. Specific ARM ABI violations like storing data outside the interval delimited by the stack base and the stack pointer [ARM Ltd., 2020, 6.2.1] will definitely result in a runtime error, because MAMBO is using it for scratch space during context switch and translation as illustrated in figure 2.2. Despite these restrictions, MAMBO is capable of running many unmodified GNU/Linux applications.



(a) Client stack with data outside the stack interval (not ABI compliant).

(b) Client stack on context switch. MAMBO overrides data outside the previous stack interval.

Figure 2.2: Client stack before and during context switch, where MAMBO overrides data outside the clients stack data interval limited by sp.

### 2.3.2 MAMBO Internals

Like many DBI tools, MAMBO is executed in the same process space as its client, which is the program or library function that is analyzed. This reduces the overall overhead, because no Inter Process Communication (IPC) between engine and client is needed, and it gives MAMBO the opportunity to control and modify the client from its very first instruction. Another approach would be to attach MAMBO to the running client process, but this comes with the downside of potentially missing a part of the client's execution. Running in the same process requires, that MAMBO loads the client on its own, so an ELF loader is included that supports both statically and dynamically linked binaries. MAMBO controls and modifies the application code by scanning and translating at basic block granularity at runtime. A basic block is a single entry, single exit block of sequential binary code. The scanner searches the blcok, that is going to be executed next, for its exit, which is usually a branch or another control transfer instruction, and translates the basic block. The translation takes place within the same instruction set, so A32 code is translated to A32 code directly, and written to a code cache. The code cache contains a number of translated basic blocks that are modified during translation to be able to run inside the code cache. So the translation alters jump targets and instructions that depend on the program counter. Additional plugins can interfere with the translation process and instrument code. For example, a plugin that counts load instructions could insert code to increment a counter every time a load instruction occurs. When executed, a basic block in the code cache either transfers control over to another block in the code cache or to the DBI engine that initiates the scan and translation of the next basic block. Further, MAMBO maintains control over the client by also wrapping system calls and signals. Some need to be intercepted and modified, like the system call `brk` which modifies the data segment size. It is used for memory allocation, but MAMBO manages the data of the client itself, so it emulates this system call.

Some DBI tools like Valgrind prefer to translate the code into an intermediate representation to perform their modifications and translating them back to run from a code cache. The advantages in portability of this approach come with high performance penalties and a possible loss in information depending on how well the intermediate representation fits the source ISA.

A fast and lightweight DBI engine includes optimizations to speed up the translation and the execution of the translated code. Most overhead is produced by the context switch between client code and the DBI engine, so reducing the amount of context switches and the complexity of the translating routines will provide a significant performance boost. Here the code cache comes in handy, so translated basic blocks can be reused without a redundant translation. MAMBO provides a hash table of translated basic blocks in the code cache, so translated blocks can be found fast [Gorgovan et al., 2016].

#### Code Cache

Modifying code inside a binary is a tricky task and can lead to serious performance penalties. For example, extending code in place requires all following code to be moved to create space for the additional instructions and a lot of offsets need to be updated. A better solution is to write the modified code into a different place called the code cache. MAMBO's code cache consist of multiple memory blocks of a fixed size. Each block is intended to store one Translated Basic Block (TBB). If a TBB exceeds the size of one block, multiple code cache blocks can be linked together. The size and the number of the blocks are statically defined and can not change during execution. As soon as all blocks contain translated code, the code cache is considered full. To store additional TBBs to a full code cache, occupied blocks must be freed. Instead of wasting time to find the best block to free and updating all direct links to that block, MAMBO simply flushes the entire code cache.

#### Hash Table

A direct advantage of a statically defined code cache is, that single blocks can be referenced easily by linearly incrementing the pointer by the block size. To be able to reuse code cache blocks and to benefit optimizations like direct-branch linking (2.3.3) and the inline hash lookup (2.3.4), MAMBO uses a hash table [Gorgovan et al., 2016]. When a basic block was translated and written to a block in the code cache, the hash table is updated to include the source address of the client and the address of the corresponding block in the code cache, as shown in figure 2.3. The hash table effectively records which code cache block is assigned to which basic block of the client. The position of this assignment in the hash table is determined by a simple hash function that can be implemented with a single `AND` instruction on ARM.

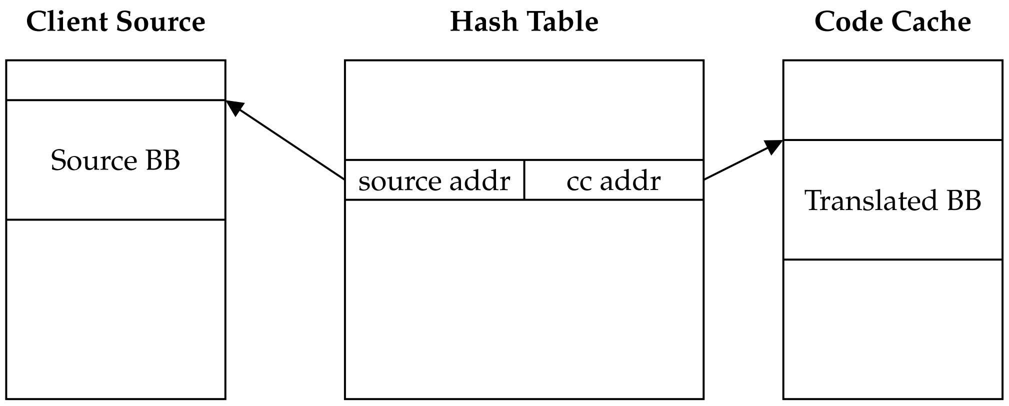
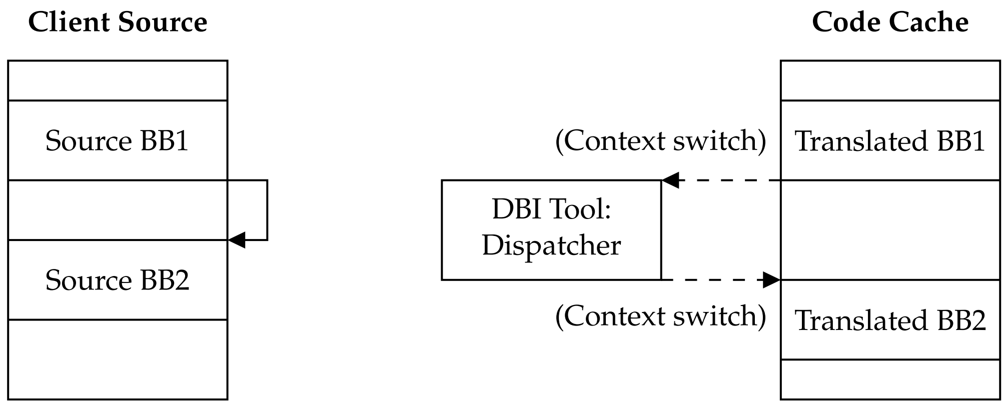


Figure 2.3: Hash Table connects the source basic block with its translation in the code cache.

### 2.3.3 Direct Branches

Starting with the optimization of the simplest form of jumps: unconditional direct branches. An unconditional direct branch is basically a naive jump from the current address to a target address described with a relative offset encoded in the instruction. Conditional direct jumps are very similar, with the only difference being that the jump is performed only if a condition is met. Because the execution of the client code takes place inside the code cache, the code at the target address must be translated and written to a block in the code cache. The translation routine replaces the original direct branch with a call to a dispatcher, so the invoked dispatcher routine can find and jump to the translated target block. The dispatcher will first check if the basic block was already translated and if not, it will perform the translation and write the result to the code cache. Once a previously translated block is found or a new translation is performed, the dispatcher resumes execution at that new block. The next time the two basic blocks are executed, the entire procedure repeats.

Figure 2.4 shows this naive approach where Source BB1 contains a direct jump to Source BB2, and they are translated to Translated BB1 containing the overwritten call to the dispatcher which resumes the execution to Translated BB2.



(a) Source basic block 1 exits with a jump to source basic block 2.

(b) TBB1 calls DBI tool which then jumps to TBB2. Two context switches are required.

Figure 2.4: Direct branch translation without optimization

### Direct-Branch Linking

The redundant invocation of the dispatcher to look up and jump to the same address every time can be optimized easily. At the first time the dispatcher is called to resolve one direct branch, it overwrites the dispatcher call with a direct jump to the translated target in the code cache. That way, both basic blocks are directly linked together and execute without any context switch after the first time. Figure 2.5 shows the code cache after the dispatcher inserted the direct link.

## 2.3.4 Indirect Branches

Indirect branches are jumps whose target is computed at runtime. The target address is usually passed in a register, so in contrast to direct jumps, the jump target can not be determined by just decoding the instruction. This also effects the translation, because the jump target is not known at translation time as well. So again, a call to the dispatcher is inserted which has the task to determine the target address at runtime, but this time it can

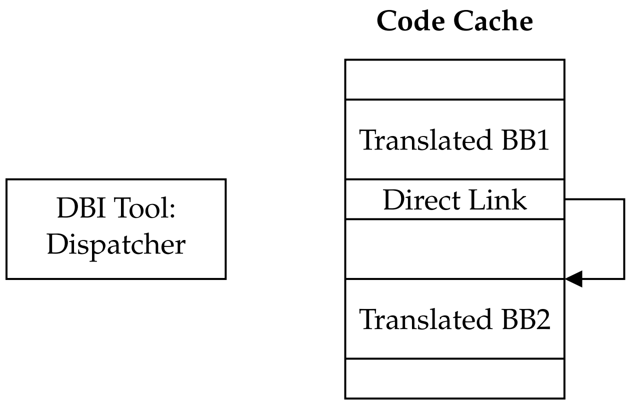


Figure 2.5: Direct-branch link inserted to directly jump to TBB2.

not do the direct-branch linking because the target address is unknown and can change during execution.

#### Inline Hash Lookup

Similar to the procedure with direct branches, for indirect branches the dispatcher is invoked to determine the address of the translated basic block for the target block and translates it, if it is not available in the code cache yet. Inserting a direct link is not sufficient, so a hash lookup routine replaces the indirect branch instead. It reduces the context switch overhead, if the target is already translated [Gorgovan et al., 2016]. This means the Indirect Hash Lookup (IHL) is part of the translated basic block. It consists of an efficient implementation of the same lookup routine that already is part of the dispatcher. If an existing block is found for the corresponding target address, the routine directly jumps to that block, otherwise the dispatcher will be called for translation of the target block. The inline lookup prevents a high overhead due to a context switch to the dispatcher, if a translation of the target block is already present in the code cache. Additionally, encoding the hash lookup into the translation allows the processor to handle branch target prediction for every translated indirect branch individually [Gorgovan et al., 2016].

### 2.3.5 More Optimizations

MAMBO provides additional advanced optimization techniques to further increase the performance of the DBI engine. This includes eliding unconditional direct branches for more efficient instruction cache and code cache usage [Gorgovan et al., 2016] and super blocks (traces), which are single entry and multiple exit blocks merging frequent se-

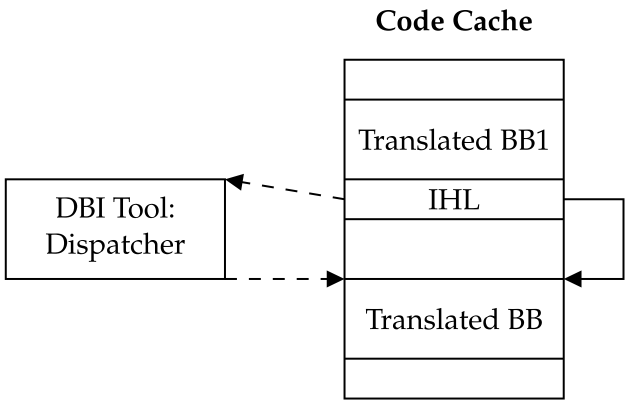


Figure 2.6: Inline hash lookup appended to TBB1 that jumps to a dynamically determined target TBB if found, the dispatcher is called otherwise.

quences of basic blocks together [Gorgovan et al., 2018] [Callaghan et al., 2020]. There are also further optimizations to indirect branches called Indirect Branch Inlining (IBI) where indirect branch targets are predicted and directly linked, skipping an IHL if the prediction is correct [Callaghan et al., 2020].

## 2.4 MicroWalk

MicroWalk [Wichelmann et al., 2018] is a software analysis framework created to automatically find microarchitectural side channels. Its development was motivated by the threat of remotely exploitable side channels such as cache attacks [Bernstein, 2005] in cryptographic implementations. A common countermeasure against those attacks is software hardening by a constant-time implementation, so that it uses microarchitectural resources in a secret-independent fashion. But software hardening is an error-prone task. Following a white-box model, the MicroWalk framework analyzes binaries and locates non-constant time behavior that result in potential leakage of secret information to support the development of constant time implementations [Wichelmann et al., 2018].

The framework runs a dynamic binary analysis with Pin [Luk et al., 2005] to generate execution traces, that is currently limited to x86/x64 binaries. However, to support other architectures, traces generated by external DBI tools like MAMBO can be supplied for analysis. Chapter 4 describes this approach in more detail.

### 2.4.1 Microarchitectural Leakages

Leakages in software evolving from the underlying microarchitecture are microarchitectural side channels. They are created by components and features of the hardware, like caches or branch prediction. For example, data caches provide low memory latencies, but at the same time, they create a timing side channel, which leaks information about data accesses [Bernstein, 2005]. It can be exploited by measuring data access times. If the access latency is low, it means a data block was already accessed before, otherwise the latency would be high. On a system providing shared libraries to multiple applications, an attacker could collect information about data accesses of other applications through this microarchitectural side channel.

Microarchitectural side channels also depend on the ISA and the used compiler, because of the variety of instructions in a processor and the usage of them in the binary defined by the compiler. A certain implementation may be translated to constant time code by a AArch64 compliant compiler, but a RV64GC compiler creates a non-constant time binary due to the availability of other instructions. Even on the same architecture, the compiler generated binary can be different to the source code because of several optimizations [Simon et al., 2018]. Therefore, MicroWalk analyzes software binaries [Wichelmann et al., 2018] rather than source code.

### 2.4.2 Mutual Information

Mutual Information (MI) is a term in information theory, that describes the amount of information of one variable which can be measured through the observation of a random second variable. In the side channel context, MI can be redefined as a measure of the amount of leakage from a secret variable through the observation of the side channel information [Wichelmann et al., 2018]. Basically, it measures how much information is leaked through a side channel.

### 2.4.3 Analysis Procedure

MicroWalk assumes a strong attacker, who can choose the inputs or secrets and is able to monitor and modify the execution of a client application [Wichelmann et al., 2018]. Figure 2.7 is a high level control-flow graph to show the basic analysis pipeline. The analysis starts with random generation of test cases, which are inputs (secrets) such as secret keys for an encryption. In the next step, the test cases are being executed with a DBI tool (Pin

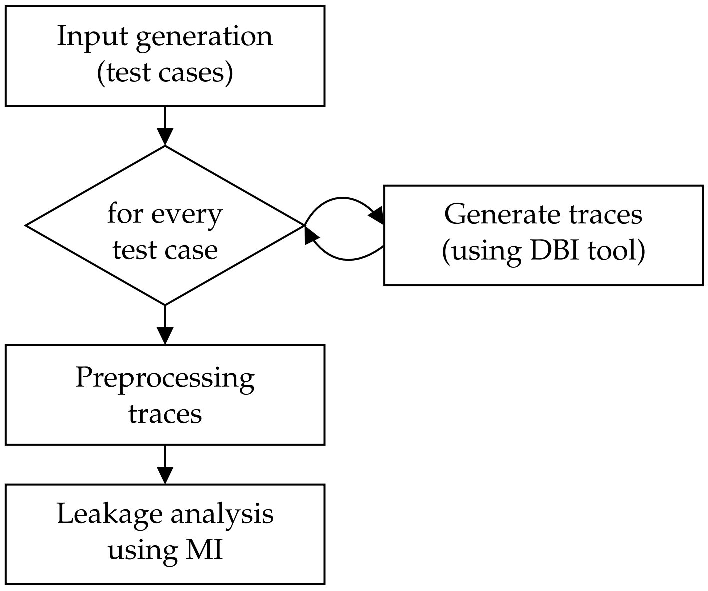


Figure 2.7: Control-flow graph of MicroWalk.

in case for x86) to generate traces for [Wichelmann et al., 2018]:

- Memory allocations
- Jumps and branches (control flow)
- Memory reads and writes (data flow)
- Stack operations

The generated traces correspond to internal states of the application. When the trace generation is done for all test cases, MicroWalk goes into preprocessing. The traces are prepared for comparison in the last step. For example, absolute addresses are replaced with relative ones to reduce noise. Finally, the preprocessed traces are compared with each other. The traces of a truly constant-time implementation are expected to be indistinguishable from each other as different secrets do not change the application states that were observed. Opposingly, a non-constant time behavior with secret dependent memory accesses or branches would lead to detectable differences between the traces. In case there is a observable dependency between the internal states and the secrets from the test cases, the binary could be vulnerable to microarchitectural side-channel attacks. Further analysis using MI also determines the locations of non-constant time behaving code and calculates the amount of leaked information [Wichelmann et al., 2018].

A high MI value means, that a strong attacker can precisely assign the secret inputs to the corresponding system states. Hence, the application would leak secret data over side channels.

# 3 MAMBO-V

A widely used countermeasure against microarchitectural leakages is software hardening by developing constant-time code. Since the compiler is completely restructuring the program during compiling, the program's binary should be analyzed for constant-time behavior rather than the source code. The program binary heavily depends on its target architecture and can only be executed there, so traces can only be generated on the target platform. Due to the growing interest in RISC-V platforms and development of RISC-V exclusive software, the interest in security on RISC-V platforms is also increasing. At the beginning of our project, no DBI tool had official support for RISC-V. Building a DBI tool from scratch requires a lot of time in development and a deep understanding of the target architecture and the operating system. The advantage of this approach is, that the tool can be created directly for the target architecture and can benefit from various optimizations specific to that architecture leading to a lower overhead. Another approach, with the advantage of a lower development overhead, is to port and customize an existing DBI tool targeting a similar architecture. The less different the architecture is to RISC-V, the better the fit and the more opportunities to include architecture dependent optimizations. A performant and lightweight DBI tool is important for RISC-V platforms, because it allows dynamic binary analysis on low performance embedded devices, which is also a target area for RISC-V processors. To put it all together, our requirements for a base DBI tool for porting are:

- Built for similar architecture to RISC-V
- Lightweight on system resources

MAMBO [Gorgovan et al., 2016] is a DBI tool that fulfills our requirements. It is originally developed to run on embedded systems with a Linux-based operating system and supports the A32, T32 and A64 instruction sets. The ARM architecture is also considered a RISC architecture and shares similarities with RISC-V in multiple aspects. MAMBO features a small codebase and many optimizations for an efficient translation and low execution overhead.

## 3.1 Features

MAMBO-V is a port of MAMBO to add support to run on RISC-V platforms. It includes all features required to achieve behavioral transparency, and it includes those optimizations which are most important for reducing overhead.

Feature overview:

- Supports RV64GC platforms and binaries
- RISC-V ELF loader
- Behavioral transparency
- Fully ported plugin Application Programming Interface (API)
- Translation optimizations:
  - Direkt-branch linking
  - Inline hash lookups

MAMBO-V currently only supports RV64GC platforms which means, platforms with support for the RV64I base instruction set and the extensions M, A, C, F, D, Zicsr, Zifencei. A popular RISC-V application processor, that meets the RV64GC specification, is the SiFive U54. Binaries targeting processors with additional extensions like the SiFive U74 featuring RV64GBC[3] cores or the SiFive P270 compatible with RV64GBCV[3] will likely run as well as long as no additional jump or branch instruction are added, but we did not test it. If an application manipulates its control flow with instructions that MAMBO-V is not aware of, it will lose control over the application, because code is not executed in MAMBO-V's code cache anymore. A RV64GC processor is also the minimum requirement, because the included assembler code only compiles for at least RV64GC.

MAMBO-V includes a slightly modified ELF loader, compared to MAMBO, capable of loading RISC-V ELF files. It is used to load a RISC-V executable into the running MAMBO-V process, allowing it to access all the client's memory and controlling the execution from the very first instruction. The implementation mainly differs in some macro definitions.

MAMBO-V follows the same transparency goal as the original MAMBO, behavioral transparency. This level of transparency is sufficient to perform binary analysis on software and to find non-constant time behavior. The level of transparency does not limit the kind of analysis that can be performed, it limits the set of software that can be analyzed.

[3]"B" was the name of the draft of the RISC-V Bit-Manipulation ISA-extensions

The full plugin API and all architecture independent helper functions are supported by MAMBO-V providing a rich interface for MAMBO plugins. Albeit most API functions are architecture independent, plugins are not guarantied to be. For example, a small plugin for counting branches can be implemented in a portable way, so it compiles and executes for ARM 32 and 64 bit as well as for RISC-V (with MAMBO-V) out of the box. More complex plugins, that write instructions directly to the code cache without the use of the helper functions, are not portable out of the box and must be implemented for each architecture separately. Leaving the API open in a way, that plugins can write every code they want to the code cache, comes with the cost of portability but has the advantage, that a wide variety of analysis methods can be implemented as a plugin. The current helper functions actually add an architecture abstraction layer as a result of having the same function available for all architectures, but their main purpose is to simplify and speed up the plugin development.

## 3.2 Implementation Differences

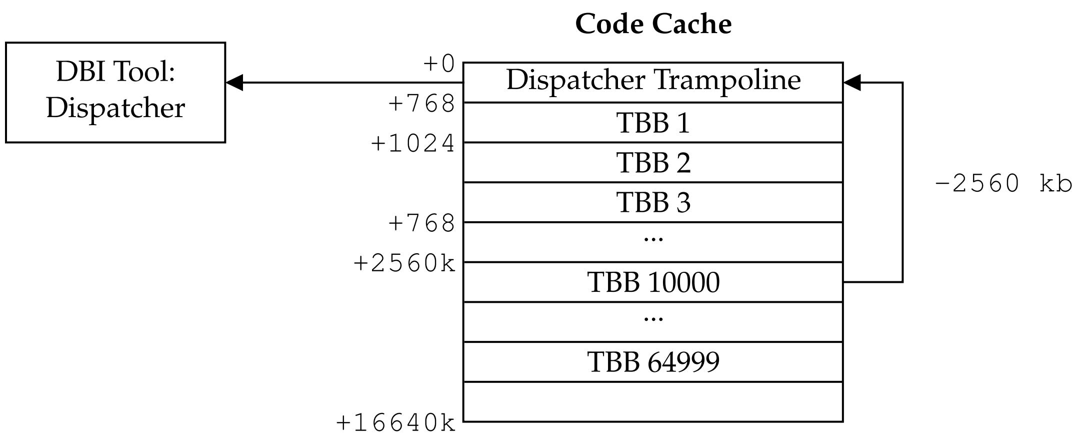
The differences between ARM and RISC-V introduce some differences and new challenges in developing MAMBO-V. Different instruction encodings lead to larger or smaller immediate values and offsets for jumps, discrepancies in the behavior of hardware implementations and a different programmer model all introduce challenges in implementing a port to a different architecture. In the following, we explain some of the challenges we faced during development and present our solutions.

### 3.2.1 Jump Range

RISC-V and ARM do not have direct branch instructions that take an absolute address. Only offsets can be passed due to encoding limitations. Because of different instruction encoding schemes, the highest offset range is achieved with ARMs A64 `B` (Branch) instruction. It takes 26 bits of a signed offset that is multiplied by 4 because of the 4 byte alignment in AArch64 resulting in a $\pm 128$ MiB jump range [ARM Ltd., 2021]. The RISC-V direct jump with the highest possible offset is the `JAL` (Jump and Link) instruction. It encodes only 21 bits for a signed offset that is only doubled because of a finer 2 byte alignment. Therefore, direct jumps in RISC-V have a maximum jump range of $\pm 1$ MiB.

If the superblock optimization option (ARM only) is disabled, the code cache consists of 65000 blocks each 256 byte in size by default. The entire code cache takes about 16.6 MB in total. Figure 3.1a is an example for a code cache filled with TBBs where block number

10000 contains a jump to the dispatcher trampoline. The dispatcher trampoline backups the client's state, transfers control to the DBI tool and restores the client state again, when the DBI tool returns. It is located among other trampolines at the beginning of the code cache. In a naive implementation, the translator would insert a Jump-and-Link (`JAL`) instruction to jump 2560 kB backwards as shown in figure 3.1b, but the instruction can only encode an offset of approximately 1049 kB at maximum in both directions.



(a) Reaching the dispatcher trampoline from TBBs.

```
tbb_10000:
    [...]

tbb_10000_exit:
    LI  x10, target_address
    LI  x11, basic_block_nr
    JAL dispatcher_trampoline       ; ERROR
```

(b) Naive port using direct jumps.

Figure 3.1: MAMBO's direct jumps not suitable for jumps to the dispatcher trampoline on RISC-V.

There are multiple ideas to fix this issue.

- Shrink down the code cache to 4096 blocks
- Distributed trampolines every 8192 blocks
- Use indirect jumps

The cheap fix would be to limit the number of code cache blocks to 4096, because `JAL` can jump back 1 MiB. Unfortunately for this approach, with 4096 blocks the code cache will be full and flushed very often, which would have a large negative impact in performance.

```
C.J .+10        ; Jump over dword

.dword  target      ; Hard-coded jump target

AUIPC   reg, 0      ; Load target value
LD  reg, -8(reg)
```

Listing 3.1: Indirect jump to any static location in the 64-bit address space.

Another approach without needing to change the code cache size, is to place a trampoline to jump to the dispatcher trampoline every 8192 blocks, so the TBB can jump over the nearest trampoline to the beginning of the code cache. The additional memory $m$ consumed by the trampolines can be calculated as follows, where $cc\_size$ is the total number of code cache blocks and 8192 the maximum range of `JAL` measured in code cache blocks:

$$\begin{aligned} m_{trmp} &= (\lfloor cc\_size/8192\rceil - 1) * 256B && (3.1)\\ &= (\lfloor 65000/8192\rceil - 1) * 256B && (3.2)\\ &= 1792B && (3.3)\end{aligned}$$

The additional memory used is always a multiple of the size of a code cache block and grows linear with $cc\_size$. In our case it would be 1792 bytes which correspond to 7 code cache blocks.

At the first glance, replacing all direct jumps to the dispatcher with indirect ones has a much higher memory overhead. In the worst case, all TBBs have a conditional exit with two jumps to the dispatcher trampoline. The inserted indirect jump shown in listing 3.1 takes 18 bytes of memory while a naive `JAL` would only need 4 bytes. The total additional memory required is calculated with $(cc\_size - 3)$ TBBs because the dispatcher trampoline occupies the first 3 blocks:

$$\begin{aligned} m_{idj} &= (cc\_size - 3) * 14B * 2 && (3.4)\\ &= 64997 * 14B * 2 && (3.5)\\ &= 1819916B && (3.6)\end{aligned}$$

Because we decided to implement this approach, we changed the translation of a conditional branch to work with only one indirect jump instruction, to reduce the memory overhead to the half $\approx 910kB$. But we found out, that the actual memory overhead is much lower. More memory is only needed, if a new code cache block must be allocated due to the additional 14 bytes. Hence, this approach uses additional memory only if the

```
prepare:
    LI  x10, 1
loop:
    LR.D    x11, (x20)
    BNE x11, x0, loop
    SC.D    x12, x10, (x20)
    BNE x12, x0, loop
```

Listing 3.2: Lock acquire loop.

previous basic block with the direct jump occupies between 240 and 256 bytes of the code cache block, which is very rare. According to some samples without active plugins, over 70% of the TBBs were filled up to 50% and only very few ones were filled over 75%. So we expect the actual memory overhead to be very low or even non-existent in most cases.

### 3.2.2 Atomic Loop

RISC-V and ARM provide atomic instructions. Atomic instructions are for multiprocessor and process synchronization to ensure exclusive memory operations. They are often used when dealing with software locks that manages a resource that can or should only be used by one party (thread or process). Listing 3.2 is an example implementation of a lock acquisition loop in RISC-V. It loads the lock with the specialized load-reserved (`LR`) instruction, which marks the beginning of an atomic sequence. The next instruction checks, whether the lock has the value 0 and jumps back to the beginning if not. In case the value is 0, the lock can be set. The store-conditional (`SC`) instruction stores a non-zero value back to the lock only, if the reservation of the lock by the previous `LR` is still valid. The reservation is invalidated, when a write operation occurs between the last `LR` and the `SC`, for example if another thread tries to set the same lock at the same time. The `SC` instruction ensures, that only one of multiple acquisition attempts will succeed, to avoid race conditions. The last conditional branch in listing 3.2 checks, whether the acquisition of the lock was successful, so the resource can safely be used. If `SC` fails, it will try again until it eventually succeeds.

#### Architectural Differences

The atomic sequence is bounded to strict constraints defined by the architecture. On RISC-V those `LR/SC` loops are called constrained `LR/SC` loops. They are allowed to encapsulate at most 16 instructions beginning from the last `LR` and they can only contain instructions from the base instruction set excluding some instructions like loads, stores, backward jumps and indirect jumps. Sequences, that live up to the constraints will eventually succeed. Unconstrained sequences are not guaranteed to succeed at all. Depending on the implementation, they can be deadlocks where the `SC` never succeeds [RISC-V International, 2019]

It is similar on the ARM aritecture. Lock acquisition loops can be implemented the same way but the restrictions of the hardware are different. In contrast to RISC-V, the number of instructions is not restricted in the ARM ISA but normal loads and stores, backward jumps and indirect jumps are [ARM Ltd., 2021, B2.9.5]. Unconstrained sequences, that do not meet the restrictions, may behave differently on different ARM implementations.

So on both architectures, atomic sequences containing explicit memory accesses (loads and stores) are considered unconstrained and their possibility of forward progress depends on the implementation of the device they run on. On the SiFive U54 core, an unconstrained lock acquisition loop with explicit memory access instructions causes the following `SC` to always fail, effectively creating a deadlock. But on a Raspberry Pi 3 Model B+, with a Broadcom BCM2837B0 SoC, we found out, that an unconstrained lock acquisition loop with explicit memory access instructions does not produce a deadlock.

MAMBO expects a behavior similar to the Cortex-A53. When the ARM implementation of the loop in listing 3.2 is translated, it will be split into two parts because there are two basic blocks divided by the branch in line 5 in listing 3.2. In MAMBO, every basic block begins with some loads to restore the client state in some registers. Executed on a Cortex-A53, the loop behaves as intended but on a SiFive U54 it would be translated to a deadlock. Hence, the translation of such atomic sequences needs to be redesigned to meet the specification instead of the behavior of a single hardware implementation.

### Better Translation of Atomic Sequences

We developed a lightweight solution to translate atomic sequences in MAMBO-V. It does not require the scanner to recognize atomic sequences and keep track of them, which would increase the overhead of the scanner only to handle sequences that occur very rarely. Our approach is to combine a software emulation of the atomic sequence and atomic instructions. The software emulation relaxes all the constraints of the hardware instructions but still requires atomic hardware features to avoid race conditions.

The translation takes place as shown in figure 3.2. It rewrites a load-release instruction (`LR.W` or `LR.D`) with a normal load instruction (`LW` or `LD`) and moves the read value to an available scratch register. The following code is translated as before until the store-conditional instruction and does not need to comply with most of the previous restrictions anymore. The store-conditional instruction (`SC.W` or `SC.D`) is translated to an atomic sequence using `LR` and `SC`. The `LR` loads the value again, which is compared with the value in the scratch register in the following instruction. In case there is a mismatch, the variable has changed since the first load of it and the program skips the store conditional to continue with the following code, which is usually a jump back to the first load like in listing 3.2. If both values match, the store-conditional is executed to atomically save the new value. The following code can handle the returned value just like in the original source.

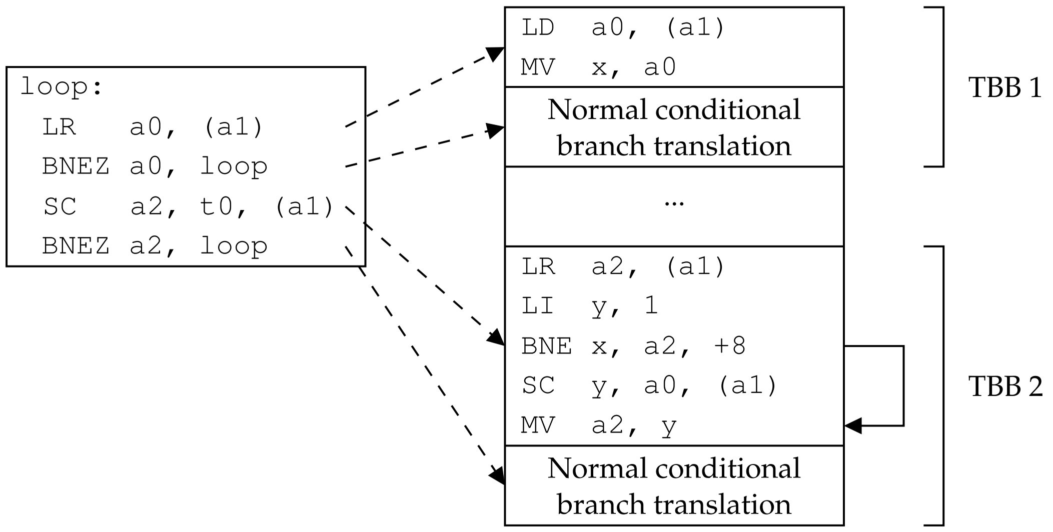


Figure 3.2: Translation of a `LR/SC` loop on RISC-V (MAMBO-V). The registers `x` and `y` are temporary scratch registers which need to be saved before use and restored after.

#### Implementation and Limitations

The current implementation of the translation of atomic sequences is an unreliable prototype, because it uses a hard-coded scratch register (`x`). It appears, that register `x31`, a volatile temporary, is rarely used, but the translator does not check if the register is actually unused and free when it saves the read value. If `x31` appears as a destination register inside the loop, it will be a deadlock again. Finding an unused register requires decoding a lot of code, which costs a lot of time, so a better implementation would rather save the value in the memory space of the DBI engine. In our tests, an atomic sequence was only discovered in some functions of the GNU standard C library, where the register `x31` is not used.

### 3.2.3 Global Pointer and Thread Pointer

In contrast to ARM, the RISC-V standard calling convention provides a global pointer `gp` and a thread pointer `tp`, which are assigned to the registers `x3` and `x4` respectively [RISC-V International, 2019]. Applications use the global pointer to access structures such as global variables and the global offset table. The latter for example saves information about the position of external library functions in dynamically linked and position independent binaries. MAMBO-V does not share the global pointer or thread pointer with the client, so MAMBO-V needs shadow registers for them, which save the values of the non-active context.

The context switch in MAMBO is originally a single-sided context switch, where only the context of the client is saved when entering the DBI context and restored when leaving again. A full context switch, exchanging all register values in both ways, was not necessary, because the DBI engine is actively calling them. But on an architecture with a global pointer and thread pointer register, MAMBO must save and restore its `gp` and `tp` on context switches, so the underlying client application cannot tamper with the context of its parent. Therefore, MAMBO-V exchanges the two pointers with the value in their shadow registers.

## 3.3 Performance Evaluation

Keeping the performance overhead as low as possible was one of our goals in developing MAMBO-V. Unfortunately, although the current implementation includes two major performance optimization techniques, it still performs bad on certain workloads.

#### Hardware

We evaluate the performance of MAMBO-V on a PolarFire SoC FPGA Icicle Kit [Microsemi, n.d.] development board by Microsemi. Besides the large selection of I/O ports such as Ethernet, USB, SD card slot and even a PCIe Gen2 port, it features a PolarFire SoC FPGA with 2 GB LPDDR4 RAM. The SoC packs a SiFive E51 (RVIMAC) microcontroller, 4 SiFive U54-MC (RV64GC) Linux capable application cores and a Field-Programmable Gate Array (FPGA) with 254 K logic elements in one package. SiFives U54-MC cores have a single issue in-order execution pipeline with 5 stages and a branch predictor with a 512-entry branch history table. The instruction cache has a size of 16 KiB [SiFive Inc., 2019].

The results can be compared with MAMBO running on a Raspberry Pi 3 Model B+. It has a Broadcom BCM2837B0 SoC with four Cortex-A53 cores, which also operate on in-order but dual-issue pipelines. The branch predictor features a 3072-entry pattern history prediction table, 6 times larger than on the SiFive U54-MC [ARM Ltd., 2018]. Cortex-A53 cores implement the ARMv8 [ARM Ltd., 2021] architecture and support the AArch64 execution state for 64-bit binaries. The BCM2837B0 comes with a 32 KiB L1 instruction cache.

Both platforms have similar microarchitectural characteristics, which are scaled differently. This is an important property for the following comparison. Widely differing features regarding the pipeline, branch predictor or instruction cache can have a significant impact on the performance penalty when running binaries in MAMBO or MAMBO-V. For example, if one platform would have support for out-of-order execution, but the corresponding MAMBO implementation would deploy code into the code cache that could barely be reordered, the relative performance penalty would be different to the same implementation in an in-order platform. In other words, we want to compare the implementation of MAMBO and MAMBO-V, not the performance of the hardware.

### 3.3.1 Benchmark MAMBO-V

Executing large benchmark suites like SPEC CPU2017 on the Icicle Kit turned out to be very challenging, because we ran into a kernel or hardware issue with the flash storage controller. And also executing benchmarks, like the one integrated in OpenSSL, which heavily relies on operating system signals were unable to run in MAMBO-V. It is prevented by an issue in signal handling. Therefore, for the current state of MAMBO-V, we needed to fall back to more basic benchmark techniques. We compared MAMBO and MAMBO-V based on the following workloads.

1. **Primes:** A small program that searches for all prime numbers inside a given interval. The source code can be found in listing 1 in the appendix. The goal of this benchmark is to generate a high workload while iterating over just a few basic blocks without disruptions from signals or system calls. We passed the interval 0 to 100000.

2. **G++:** The G++ compiler workload is similar to Primes but with more basic blocks and some system calls to modify content in the file system. We took measurements of G++ compiling the Primes program.

3. **SHA1-415:** A measurement of hashing a 415 MB file with SHA-1. Compared to the first two workloads, hashing a large file contains a very high number of system calls

to read the file from the file system.

4. **SHA1-1.3:** Similar to SHA1-415 but with a file only 1.3 MB in size.
5. **CoreMark:** A lightweight industry-standard benchmark tool to evaluate CPU performance [EEMBC, n.d.].

For Primes, G++, SHA1-415 and SHA1-1.3 we measured the CPU time with the command line tool `time`. Due to external factors like the underlying OS, the CPU time of multiple runs may vary, so we executed 20 runs of each benchmark for each platform. For each pair of benchmark and platform, the benchmark was executed 10 times without instrumentation and 10 times with default instrumentation using MAMBO or MAMBO-V respectively. The DBI tools were configured to run the same set of optimizations and did not include any optional plugins.

### 3.3.2 MAMBO and MAMBO-V comparison

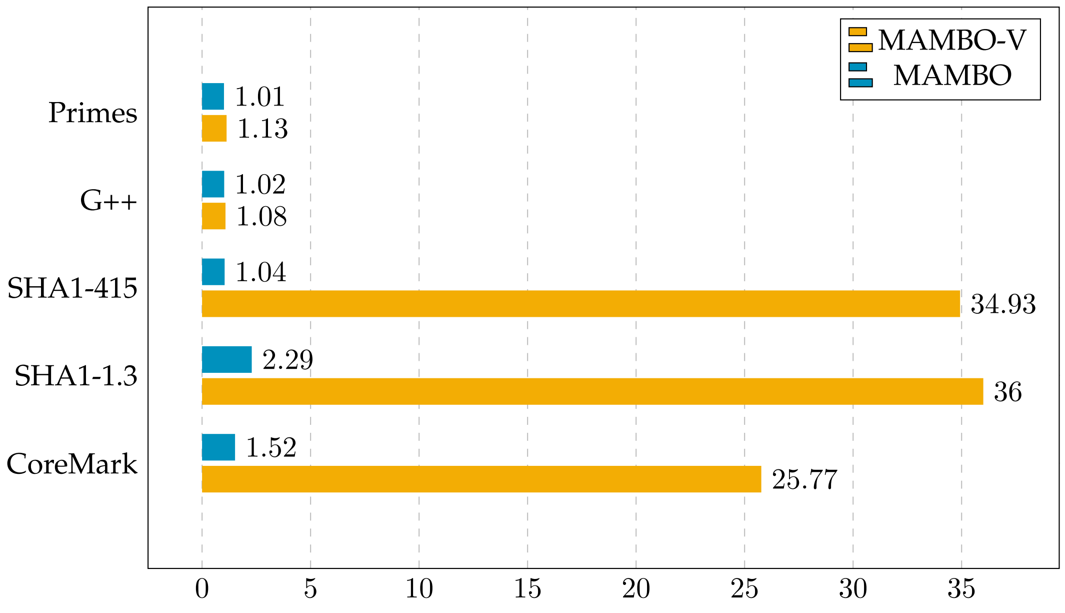


Figure 3.3: Performance slowdown comparison between MAMBO and MAMBO-V

Figure 3.3 shows the relative slowdown of the benchmarks when running in MAMBO and MAMBO-V. The performance of Primes and G++ remains nearly unchanged on both platforms. It shows, that the direct-branch linking and inline hash lookup optimizations were highly effective, because the slowdown factor of the first two benchmarks is already close to 1. The remaining benchmarks do not show such a low overhead for MAMBO-

V. One of the main reasons seems to be the high amount of system calls, especially in SHA1-415 and SHA1-1.3. The original MAMBO implementation still performs very well on these benchmarks, so the RISC-V port needs to be better optimized for reaching the same level of performance.

Nevertheless, workloads generated by cryptographic functions are very similar to the Primes and G++ loads, which already perform great. Hence, MAMBO-V is already a good choice for crypto analysis, as demonstrated in chapter 4.

# 4 Trace Plugin and Evaluation

To be able to use MAMBO-V for constant-time analysis with MicroWalk, we developed a trace generation plugin simply called *Tracer*. Its purpose is to instrument a program using MAMBO-V to write out traces, just like the Pin tool in MicroWalk does. The generated trace files are already in the format expected by the preprocessor module of MicroWalk, so they can be directly used for analysis. In the following, we will call the analyzed program the *client*.

In this chapter, we explain the trace generation by our MAMBO-V plugin and show a complete analysis run of a key decoding function from the OpenSSL [OpenSSL Foundation Inc., n.d.] library.

## 4.1 Trace Generation for MicroWalk

Execution traces for analysis with MicroWalk contain 4 types of logged operations:

1. **Memory allocation and deallocation:** Tracer instruments the functions `malloc`, `alloc`, `realloc` and `free`, so they create one trace entry when called and an additional one when they return. The first trace entry logs the arguments passed to the corresponding function and the second entry logs the returned value, which is the memory address (except for `free`).

2. **Jumps and branches:** All types of branches present in RV64GC are instrumented by the Tracer plugin. These are

   - conditional branches (`BEQ`, `BNE`, `BLT`, `BGE`, `BLTU`, `BGEU`, `BLTU`, `C.BEQZ`, `C.BNEZ`)
   - unconditional direct branches (`JAL`, `C.J`, `C.JAL`)
   - and unconditional indirect branches (`JALR`, `C.JR`, `C.JALR`).

   The information that is logged is the source and the target address in the original binary, whether the conditional branch was taken and if it was a jump, a call or a return. With this information, the control flow of the application can be reconstructed with a high level of detail.

3. **Memory reads and writes:** Also instructions that read or write memory (all loads and stores), are instrumented. The generated traces contain the address of the load or store instruction, the target address in memory and the data size.

4. **Stack pointer modifications:** Finally, instructions which actively modify the stack pointer are instrumented to generate a trace containing the new stack pointer. It provides the possibility to track stack allocations for analysis.

Generating a great amount of traces will create a large overhead, slowing down the trace generation and analysis and filling the memory with gigabytes of traces. Therefore, MicroWalk provides the ability to work with a subset of traces generated only from the interesting parts of the client's code. Tracer supports the selection of interesting binary images using a whitelist. Code, that belongs to other images not defined as interesting, will not be instrumented by the plugin. For example, to analyze an implementation of an encryption function in OpenSSL, the OpenSSL library image named `libcrypto.so.1.1` would be defined as interesting whereas standard C libraries are not interesting and should not produce traces at all.

To reduce the overhead even more, the function to be analyzed is part of a wrapper module which is effectively the client binary for MAMBO-V. It provides two function templates, one for initializing the needed components (e.g. the crypto library) and one for executing the actual test cases. In contrast to the execution of the test cases, the initialization is executed and traced only once for all test cases.

## 4.2 OpenSSL Analysis on RISC-V

To verify that MAMBO-V and the Tracer plugin are suitable to provide traces for MicroWalk, we repeat an experiment which exposed a microarchitectural leakage in key-decoding functions of cryptographic libraries [Sieck et al., 2021]. We demonstrate, that our new tools are able to identify the same leakages that were previously found in x86 binaries using Pin.

### 4.2.1 PEM Decoder Leakage

The Privacy-enhanced Electronic Mail (PEM) format is often used to store or exchange cryptographic information such as keys. Because the payload is base64 encoded, cryptographic libraries need to decode PEM files to access the included data. Across multiple libraries, the analysis with MicroWalk showed that the implementation of the base64 de-

```
_EXPORT _NOINLINE void RunTarget(FILE* input)
{
    RSA *rsa = PEM_read_RSAPrivateKey(input, nullptr, nullptr, nullptr);
    if(rsa == nullptr)
        ERR_print_errors_fp(stderr);    // Error handling

    RSA_free(rsa);
}
```

Listing 4.1: Test case function loading a public key from a file.

coding is not constant-time. The non-constant time behavior is a result of an implementation using lookup tables (LUTs). A high resolution cache attack can exploit the discovered side channel and reduce the security guarantees by a significant amount [Sieck et al., 2021].

### 4.2.2 Analysis of Key Decoding in OpenSSL

The non-constant time implementation of the base64 decoder refers to the source code not being hardened, so we can expect the same leakage in binaries compiled for other architectures like RISC-V. Our target is OpenSSL version 1.1.1i compiled for and executed on a RV64GC compatible platform. From the results of previous analysis of OpenSSL on x86 [Sieck et al., 2021], we expect the highest possible leakages to be in the `EVP_DecodeUpdate` and `EVP_DecodeBlock` functions. The former executes preparation tasks on the string and the latter performs the actual LUT-based decoding.

For the trace generation part, 512 RSA keys, each with a length of 2048 bit, are randomly generated. These keys form our set of test cases. A wrapper module calls the `PEM_read_RSAPrivateKey` function on every test case to load and decode the private key as shown in listing 4.1. We run the wrapper as a client for MAMBO-V with the Tracer plugin enabled, so it will generate a collection of traces for every iteration of the function in listing 4.1. Note, that only the trace generation with MAMBO-V is required to run in a RV64GC environment. All other steps in the MicroWalk pipeline, test case generation, preprocessing and analysis, are architecture independent.

Finally, all 512 traces can be passed to MicroWalk's preprocessor and analyzer to get a leakage estimation based on MI and detected non-constant time behavior. As expected before, the results show a high leakage potential in `EVP_DecodeUpdate` and `evp_decodeblock_int`. In the analyzed OpenSSL release, the latter is a static function that implements the LUT-based decoding. `EVP_DecodeUpdate` calls the static function directly instead of `EVP_DecodeBlock` which is just a simple wrapper.

However, MicroWalk shows heavily key dependent differences in execution states, which can result in an exploitable side channel. Please refer to table 1 in the appendix for a detailed list of the results.

### 4.2.3 Experiment Conclusion

The analysis on x86 with 4096 test cases resulted in a maximum leakage of 12 bits [Sieck et al., 2021] and on RISC-V our results with 512 test cases show a maximum of 9 bits. The leakage estimation is upper bounded by the logarithm of the number of test cases. Having 512 of them and the same number of corresponding traces, 9 bits of information are needed to select one test case out of 512 that corresponds to a given trace. With 4096 test cases, the upper bound is 12, so both experiments reach that boundary for the same routines.
Since the analysis of OpenSSL key decoding with MAMBO-V on RISC-V showed the same results as the analysis with Pin on x86, we can conclude, that MAMBO-V and the Tracer plugin have the potential to be a Pin replacement on RISC-V.

# 5 Conclusions

As industry and academia are showing a growing interest in RISC-V, the need for powerful and versatile debug and profiling tools also increases. Additionally, interest is also growing for automatic software analysis tools for security analysis. MicroWalk is such a tool to identify microarchitectural side channels in binaries by collecting traces and analyzing differences using MI. Previously, the trace generation was limited to x86 binaries and hosts. With this work we add a new supported architecture by developing one of the first DBI tools for RISC-V which generates execution traces for later analysis with our MAMBO-V plugin (Tracer).
The proposed DBI tool MAMBO-V, is a RISC-V port of MAMBO. Our implementation contains direct-branch linking and inline hash lookup optimizations as a first step to a low overhead DBI tool, that can compete with MAMBO. One of the challenges during implementation was the need to develop a new method of translating atomic sequences, which can also be beneficial for MAMBO. However, the performance evaluation showed, that optimization work is left to reach our goal of a low overhead DBI tool for RISC-V, similar to what MAMBO is for ARM.
The demonstration of MAMBO-V supplying MicroWalk with traces of OpenSSL on RISC-V, successfully exposed leakages. Therefore, we can conclude that MAMBO-V is capable of proper binary analysis and tracing non-constant time behavior. Our experiment also shows, that RISC-V binaries expose side channels as well, so they also need to be checked for non-constant time behavior.

## 5.1 Future Work

Currently, using MicroWalk with MAMBO-V requires a lot of work compared to MicroWalk's original convenient pipeline. The setup of the target function and the test cases is done by hand, as well as the execution of MAMBO-V with the Tracer plugin to generate traces. From then on, the traces can be inserted to the MicroWalk pipeline for preprocessing and analysis. So, the next step would be to fully integrate MAMBO-V into the MicroWalk pipeline for fully automatic analysis.

### 5.1.1 Open Problems

#### MAMBO-V Issues

During performance evaluation described in section 3.3, we discovered heavy performance hits with certain applications that often issue system calls. The examination of debug logs on a test run with p7zip showed, that the execution becomes slower and slower with every executed system call. A normal benchmark run finishes in less than 10 minutes, but the instrumented benchmark did not even finish after over 1 hour and did not look like it would in reasonable time.

Another issue we ran into is a bug in the signal handler, which was the reason for not having OpenSSL in the benchmark list. MAMBO inserts privileged instructions in the code cache which are not allowed to execute in user mode. As a result, the kernel throws an illegal instruction signal back to the source. MAMBO catches this signal and modifies the translation at that location in some way. Porting this behavior to RISC-V was difficult because MAMBO encodes additional information in the illegal instruction, where for RISC-V multiple instructions must be used. Apparently the implementation is not correct and needs to be reworked.

#### Microarchitectural Analysis

We showed, to a great extent, how we find and analyze non-constant time behavior from secret dependent memory accesses and control flow changes. But what about timing side channels? Can there even be a timing side channel if the memory and control flow changes are constant-time? Yes, for example, the execution latency of the division instruction depends on its operands. On a SiFive U54 the latency is defined to be between 3 and 64 cycles. MicroWalk currently does not perform or provides analysis methods for time measurement, so this might stay undetected.

## 5.2 Related Work

Using binary analysis tools like DBI to analyze software binaries is not a novel technique. There are quite a few powerful DBI engines out there that can be used for monitoring, debugging, performance optimization, security analysis and more. MAMBO is such a DBI tool. Other popular tools are Intel Pin [Luk et al., 2005], DynamoRIO [Bruening et al., 2012] and Valgrind [Nethercote and Seward, 2007]. As they are mainly developed for the

x86 and x86-64 architecture, they would introduce a higher overhead to the analysis of RISC-V binaries than MAMBO. DBI tools can also be used for data flow analysis with taint-tracking to prevent data leaks in an untrusted environment [Brahmakshatriya et al., 2019].
Using DBI tools for security analysis was discussed some time ago [D'Elia et al., 2019]. Currently, security analysis based on Pin or MAMBO is limited to software that does not try to escape from the DBI system nor changes its behavior after the detection of a DBI engine. This mainly affects malicious software which is not in the scope of MicroWalk. Nevertheless, this possible vulnerability should still be considered when analyzing unknown software.
Leakage detection and localization in software binaries for microarchitectural leakages is also the main subjects of DATA [Weiser et al., 2018]. Like MicroWalk, both analyze at runtime (dynamic analysis), generate execution traces with Pin and analyze the traces to find control-flow leaks and data leaks. Because their trace generation is based on the DBI engine Pin, the tools are limited to architectures supported by Pin which currently are IA-32 and x86-64.
Static approaches to verify constant time implementations, without executing the target itself, can have better code coverage but are more complex to develop. The better code coverage results from all code in the binary being instrumented, even when they are never executed. Static binary analysis tools like CacheAudit [Doychev et al., 2013] or ct-verif [Almeida et al., 2016] do not have any runtime information to take into account during the reconstruction of the target's control flow. It decreases the accuracy and does not scale well to large program binaries. Further challenges are calculating indirect branches and analyzing self modifying and dynamically generated code. ct-verif works with a LLVM representation of the machine code, rather than actual machine code. The additional loss in precision introduced by the translation to LLVM assembly would not affect our constant time properties, because the differences preserve control-flow paths and memory-access patterns [Almeida et al., 2016]. With access to the source code of the target, a symbolic execution approach would also be possible for binary analysis.

On embedded platforms with microcontroller SoCs, the previously mentioned DBI systems are infeasible due to the lack of an operating system. For binary instrumentation on embedded platforms (firmware analysis), the following approaches already have been proposed:

- Full system virtualization (firmware re-hosting)
- Partial virtualization

- Linker-based instrumentation

In case of a full system virtualization (called re-hosting) [Gustafson et al., 2019] [Fasano et al., 2021], the entire microcontroller is virtualized in a virtual machine. Hence, the analysis runs in software, which provides flexibility and can be parallelized at low cost. But this is difficult to achieve since all peripherals and physical interactions need to be simulated, and the environment highly depends on the hardware. Every device would need an accurate virtual machine because of the integrated nature of embedded devices and the high diversity of hardware and software across devices.
Partial virtualization methods like PROSPECT [Kammerstetter et al., 2014] and Avatar [Zaddach et al., 2014] are solving this problem by forwarding peripheral I/O to the real device. This approach uses on chip debugging interfaces like JTAG, but they are usually not exposed anymore in commercially released devices for integrity reasons.
The linker-based approach was introduced by Harzer Roller [Bogad and Huber, 2019] and does not require any virtualization. The target function symbols in an object file are modified and relinked by the linker. But instead of linking the functions as in the original binary, the linker relocates function calls to instrumentation code. With this technique, the control flow of function calls and returns statements can be instrumented without source code knowledge (black box).

# References

[Almeida et al., 2016] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. (2016). Verifying constant-time implementations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 53–70, Austin, TX. USENIX Association.

[ARM Ltd., 2018] ARM Ltd. (2018). *Arm Cortex-A53 MPCore Processor Technical Reference Manual*. DDI 0500J (ID012219).

[ARM Ltd., 2020] ARM Ltd. (2020). *Procedure Call Standard for the Arm Architecture*. Release 2020Q2.

[ARM Ltd., 2021] ARM Ltd. (2021). *Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile*. ARM DDI 0487G.a (ID011921).

[Asanović and Patterson, 2014] Asanović, K. and Patterson, D. A. (2014). Instruction sets should be free: The case for risc-v. Technical Report UCB/EECS-2014-146, EECS Department, University of California, Berkeley.

[Bernstein, 2005] Bernstein, D. J. (2005). Cache-timing attacks on aes. Technical report, Department of Mathematics, Statistics and Computer Science, The University of Illinois at Chicago.

[Bogad and Huber, 2019] Bogad, K. and Huber, M. (2019). Harzer roller: Linker-based instrumentation for enhanced embedded security testing. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, New York, NY, USA. Association for Computing Machinery.

[Brahmakshatriya et al., 2019] Brahmakshatriya, A., Kedia, P., McKee, D. P., Garg, D., Lal, A., Rastogi, A., Nemati, H., Panda, A., and Bhatu, P. (2019). ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 1–15, New York, NY, USA. Association for Computing Machinery.

[Bruening et al., 2012] Bruening, D., Zhao, Q., and Amarasinghe, S. (2012). Transparent dynamic instrumentation. *SIGPLAN Not.*, 47(7):133–144.

[Callaghan et al., 2020] Callaghan, G., Gorgovan, C., and Luján, M. (2020). Optimising dynamic binary modification across 64-bit Arm microarchitectures. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, pages 185–197, New York, NY, USA. Association for Computing Machinery.

[D'Elia et al., 2019] D'Elia, D. C., Coppa, E., Nicchi, S., Palmaro, F., and Cavallaro, L. (2019). Sok: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 15–27, New York, NY, USA. Association for Computing Machinery.

[Doychev et al., 2013] Doychev, G., Feld, D., Kopf, B., Mauborgne, L., and Reineke, J. (2013). Cacheaudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 431–446, Washington, D.C. USENIX Association.

[EEMBC, n.d.] EEMBC (n.d.). CoreMark is an industry-standard benchmark that measures the performance of central processing units (CPU) and embedded microcontrollers (MCU). `https://www.eembc.org/coremark/`. Online; accessed 2021-12-09.

[Fasano et al., 2021] Fasano, A., Ballo, T., Muench, M., Leek, T., Bulekov, A., Dolan-Gavitt, B., Egele, M., Francillon, A., Lu, L., Gregory, N., Balzarotti, D., and Robertson, W. (2021). Sok: Enabling security analyses of embedded systems via rehosting. In Cao, J., Au, M. H., Lin, Z., and Yung, M., editors, *ASIA CCS '21: ACM Asia Conference on Computer and Communications Security, Virtual Event, Hong Kong, June 7-11, 2021*, pages 687–701. ACM.

[Gorgovan et al., 2016] Gorgovan, C., d'Antras, A., and Luján, M. (2016). MAMBO: A Low-Overhead Dynamic Binary Modification Tool for ARM. *ACM Transactions on Architecture and Code Optimization*, 13(1):14:1–14:26.

[Gorgovan et al., 2018] Gorgovan, C., d'Antras, A., and Luján, M. (2018). Optimising Dynamic Binary Modification Across ARM Microarchitectures. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 28–39, New York, NY, USA. Association for Computing Machinery.

[Gustafson et al., 2019] Gustafson, E., Muench, M., Spensky, C., Redini, N., Machiry, A., Fratantonio, Y., Balzarotti, D., Francillon, A., Choe, Y. R., Kruegel, C., and Vigna, G. (2019). Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 135–150, Chaoyang District, Beijing. USENIX Association.

[Kammerstetter et al., 2014] Kammerstetter, M., Platzer, C., and Kastner, W. (2014). Prospect: Peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 329–340, New York, NY, USA. Association for Computing Machinery.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200.

[McMahon, 2021] McMahon, K. (2021). RISC-V International Ratifies 15 New Specifications, Opening Up New Possibilities for RISC-V Designs. `https://riscv.org/announcements/2021/12/riscv-ratifies-15-new-specifications/`. Online; accessed 2021-12-03.

[Microsemi, n.d.] Microsemi (n.d.). MPFS-ICICLE-KIT-ES | Microsemi. `https://www.microsemi.com/existing-parts/parts/152514`. Online; accessed 2021-12-11.

[Nethercote and Seward, 2007] Nethercote, N. and Seward, J. (2007). Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100.

[OpenSSL Foundation Inc., n.d.] OpenSSL Foundation Inc. (n.d.). OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`. Online; accessed 2021-12-11.

[Patterson and Hennessy, 2017] Patterson, D. A. and Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

[RISC-V International, 2019] RISC-V International (2019). *The RISC-V Instruction Set Manual*. Volume I: User-Level ISA. Editors Andrew Waterman and Krste Asanović.

[RISC-V International, 2021a] RISC-V International (2021a). *RISC-V Cryptography Extensions Volume I*. Version 1.0.0-rc6.

[RISC-V International, 2021b] RISC-V International (2021b). *The RISC-V Instruction Set Manual Volume II: Privileged Architecture*. Version 20211203.

[RISC-V International, 2021c] RISC-V International (2021c). *RISC-V "V" Vector Extension*. Version 1.0-rc2.

[RISC-V International, n.d.] RISC-V International (n.d.). History of RISC-V. `https://riscv.org/about/history/`. Online; accessed 2021-10-11.

[Sieck et al., 2021] Sieck, F., Berndt, S., Wichelmann, J., and Eisenbarth, T. (2021).

Util::Lookup: Exploiting key decoding in cryptographic libraries. *arXiv:2108.04600 [cs]*. arXiv: 2108.04600.

[SiFive Inc., 2019] SiFive Inc. (2019). *SiFive U54-MC Manual*. Version 19.08p0.

[Simon et al., 2018] Simon, L., Chisnall, D., and Anderson, R. (2018). What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 1–15.

[Waterman, 2016] Waterman, A. (2016). *Design of the RISC-V Instruction Set Architecture*. PhD thesis, EECS Department, University of California, Berkeley.

[Waterman et al., 2011] Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K. (2011). The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA. Technical Report UCB/EECS-2011-62, EECS Department, University of California, Berkeley.

[Weiser et al., 2018] Weiser, S., Zankl, A., Spreitzer, R., Miller, K., Mangard, S., and Sigl, G. (2018). DATA – differential address trace analysis: Finding address-based side-channels in binaries. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 603–620, Baltimore, MD. USENIX Association.

[Wichelmann et al., 2018] Wichelmann, J., Moghimi, A., Eisenbarth, T., and Sunar, B. (2018). MicroWalk: A Framework for Finding Side Channels in Binaries. *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 161–173. arXiv: 1808.05575.

[Zaddach et al., 2014] Zaddach, J., Bruno, L., Francillon, A., and Balzarotti, D. (2014). Avatar: A framework to support dynamic security analysis of embedded systems' firmwares.

# Appendices

| | Location | Estimated leakage (bits) |
|---|---|---|
| (static) | `evp_decodeblock_int <+0x2E>` | 9 |
| (static) | `evp_decodeblock_int <+0x98>` | 9 |
| (static) | `evp_decodeblock_int <+0xB6>` | 9 |
| (static) | `evp_decodeblock_int <+0xCC>` | 9 |
| (static) | `evp_decodeblock_int <+0xE4>` | 9 |
| | `EVP_DecodeUpdate <+0x94>` | 9 |
| | `BN_bin2bn <+0x58>` | 3.963 |
| | `BN_bin2bn <+0x20>` | 3.951 |
| | `ASN1_get_object <+0x106>` | 2.952 |
| | `ASN1_get_object <+0x16>` | 2.937 |
| | `ASN1_get_object <+0x3C>` | 2.937 |
| | `ASN1_tag2bit <+0x420>` | 2.937 |
| | `RSA_padding_check_PKCS1_OAEP <+0x30>` | 2.822 |
| | `RSA_padding_check_PKCS1_OAEP <+0x38>` | 2.822 |
| | `RSA_free <+0x9E>` | 2.602 |
| | `RSA_free <+0x92>` | 2.322 |
| (static) | `evp_decodeblock_int <+0x66>` | 1.013 |
| (static) | `evp_decodeblock_int <+0x86>` | 1.013 |
| (static) | `evp_decodeblock_int <+0x9C>` | 1.013 |
| (static) | `evp_decodeblock_int <+0xBA>` | 1.013 |
| (static) | `evp_decodeblock_int <+0xDO>` | 1.013 |
| | `EVP_DecodeUpdate <+0x7E>` | 1.013 |
| | `EVP_DecodeUpdate <+0x132>` | 1.013 |
| (static) | `sanitize_line <+0x34>` | 1.013 |
| (static) | `sanitize_line <+0x50>` | 1.013 |
| (static) | `sanitize_line <+0x54>` | 1.013 |

Table 1: List of results from MicroWalk where the leakage estimation is greater than 1. Function names of static functions were determined by reverse engineering one of its non-static parent functions and comparing to the source code.

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main(int argc, char **argv) {
    long long low, high, i, prime;
    bool isPrime = true;

    low = std::atoll(argv[1]);
    high = std::atoll(argv[2]);

    cout << "Interval:␣" << low << "␣-␣" << high << endl;

    while (low < high) {
        isPrime = true;
        if (low == 0 || low == 1) {
            isPrime = false;
        }
        else {
            for (i = 2; i <= low / 2; ++i) {
                if (low % i == 0) {
                    isPrime = false;
                    break;
                }
            }
        }

        if (isPrime)
            prime = low;
        ++low;
    }

    return 0;
}
```

Listing 1: Primes: A small prime generation benchmark.