DAMAS: Control-Data Isolation at Runtime through Dynamic Binary Modification

Camille Le Bon, Erven Rohou, Frédéric Tronel, Guillaume Hiet

DAMAS: Control-Data Isolation at Runtime through Dynamic Binary Modification

Camille Le Bon, Erven Rohou
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
camille.le-bon@inria.fr
erven.rohou@inria.fr

Frédéric Tronel, Guillaume Hiet
CentraleSupelec, Inria, Univ Rennes, CNRS, IRISA
Rennes, France
frederic.tronel@centralesupelec.fr
guillaume.hiet@centralesupelec.fr

***Abstract*—Memory corruption attacks have been a major issue in software security for over two decades and are still one of the most dangerous and widespread types of attacks nowadays. Among these attacks, control-flow hijack attacks are the most popular and powerful, enabling the attacker to execute arbitrary code inside the target process. Many approaches have been developed to mitigate such attacks and to prevent them from happening.**

One of these approaches is the Control-Data Isolation (CDI) that tries to prevent such attacks by removing their trigger from the code, namely indirect branches. This approach has been implemented as a compiler pass that replaces every indirect branches in the program with a table that leads the control-flow to direct hard-written branches. The drawback of this approach is that it needs the recompilation of the program. In this paper we present an approach and its implementation, DAMAS, a framework capable of deploying protections on a running software and use runtime information to optimize them during the process execution. We implemented a coarse-grain CDI protection using our framework and evaluated its impact on performance.

***Index Terms*—control-data isolation, dynamic binary modification, binary rewriting**

1. Introduction

Memory corruption bugs are one of the oldest problems in computer security that still persists. Programming languages such as C or C++ that allow the programmer to manually manage memory are prone to this kind of bugs. Unfortunately, these languages are among the most used languages in the industry [1]. Those bugs are entry doors for malicious users to alter the behavior of a program or to even take full control of its control-flow [2], [3], [4].

There are many ways to take advantage of a memory bug and attack a vulnerable program. Szekeres et al. proposed a model of memory corruption attacks [5]. This model presents every possible scenario that may lead to a memory corruption attack as well as security policies that can prevent these attacks to happen. Implementations of some of these policies are active by default on modern systems because of their negligible impact on the performance of programs, such as W$\oplus$X or address-space layout randomization. Due to the presence of such protections, some attack vectors have become rare in real attacks, such as the injection of shellcodes. However, these protections only make attacks harder to perform and require higher skills from the attacker to circumvent them but they are not able to ensure a complete memory safety [6], [7].

Control-flow hijack attacks are among the most common and dangerous memory-corruption-based attacks. They allow the attacker to take full control of the control-flow, thus permit the execution of arbitrary code. Currently, no protection can entirely erase the threat of this type of attacks. According to Szekeres et al. [5], a complete implementation of the control-flow integrity (CFI) policy should prevent the program from diverging from the control-flow decided by the programmer. Nonetheless, all implementations of CFI have been defeated as of today [6], [7]. The root issue that permits a process to diverge from its original control-flow is the presence of indirect branches in its code. These kind of branches are omnipresent and take several forms: return instructions, indirect calls, indirect jumps, etc. A solution to control-flow hijacking could be to remove every indirect branch as proposed by the control-data isolation policy (CDI) [8]. This approach transforms every indirect branch instruction into a direct one, replacing them by sleds of comparison/jump pairs.

Most implementations of these approaches are implemented as a compilation pass and use the control-flow graph (CFG) built by the compiler to generate the appropriate protection code. Binary-based CFI implementations rely on their ability to recover the initial CFG of the program. Hence the protection may be very coarse in order to preserve the semantics of the program and avoid false positives. The trade off between the overall safety added and the overhead in performance introduced by the protection is usually pretty bad.

Solutions that use runtime information are not perfect either. These solutions monitor the process execution in order to recover information about the CFG of the program. However this monitoring has two main pitfalls. First the completeness of the CFG depends highly on the code coverage of the monitored execution of the process. Second, if the monitored execution is itself deceived by an attacker, malicious CFG edges may be considered valid by the protection mechanism. Moreover, the protection is added ahead of time in order to patch the target binary. This approach introduces difficulties to patch the binary file such as generating assembly code that respects the assumptions made in the original code as exposed by

Wang et al. [9], such as the relative position of sections according to each other and the consequences of this position in terms of offsets in the code.

1.1. Contributions of this work

The main issue with the solution proposed by Arthur et al. [8] is the need to recompile the program to deploy the protection. COTS software thus cannot benefit from this solution. Moreover, the need for a process to be stopped and restarted to deploy a security solution may be an undesired constraint. In order to respond to these problems, we propose in this paper an approach using dynamic binary modification (DBM) to deploy protections on running processes. Our approach allows to add protections, optimize them and even remove them at runtime without the need to restart the process.

Using our framework, we implemented a protection solution greatly inspired from CDI that uses runtime information to reduce its impact on the performance of the target process.

In order to enforce a robust CFI solution, a complete CFG needs to be recovered. Nonetheless, the lack of compile-time information such as the source code or the CFG prevents us from knowing exactly what the semantics of the program is. The recovery of a complete CFG from the binary code alone without executing the program first to discover dynamic paths is a difficult task [10]. The most famous binary-analysis frameworks such as IDA Pro [11], ANGR [12], Ghidra [13], BAP [14] or Radare2 [15] are not able to generate complete enough CFG to ensure that a perfectly valid execution of a program never escapes the computed CFG.

The goal of this work is to adapt the control-flow-isolation policy to make it work on running processes only using a disassembly of their binary code and runtime information such as the mapping of memory. We make the following contributions:

- Given precise function boundaries in a program binary, we propose an approach to enforce a protection scheme based on control-data isolation without the need for the sources nor to recompile the program.
- Our implementation of this approach, DAMAS, can attach to a running process and deploy protections without the need to restart it. We evaluated our solution in order to measure its impact on the performance of the instrumented programs.
- We propose multiple optimizations to incrementally minimize the impact of our solution on the target process performance.

2. Our approach

The goal of our approach is to prevent a program from diverging from its intended control-flow. To do that, unlike CFI, we do not check at runtime that an indirect branch is valid, but we get rid of indirect branches.

In this article, we call a *jump* the control-flow instruction `jmp` and *branch* any kind of control-flow instruction whether it is a jump, a call or a return instruction, as described precisely in Table 1. We chose these terms for the sake of consistency, clarity and to agree with the rest of the literature that calls *indirect branch* any kind of branch whose operand is not an immediate.

Generic name	Category	Examples (x86_64 ISA)
branch	jump	`jmp [rip+0xd7f9a]` `jmp [0x4c5580+rax*8]`
	call	`call [rax]` `call [r15+rbx*8]`
	return	`ret`

TABLE 1. OUR TERMINOLOGY OF INDIRECT CONTROL-TRANSFER INSTRUCTIONS

Just like control-data isolation [8], indirect branches are replaced by comparisons to valid potential target addresses and a corresponding direct branch. The program must compare the computed branch target addresses one-by-one to the hard-coded potential target addresses until it finds one that corresponds. A sled (for a call instruction) would be organized like the following piece of code:

```
if (fptr == addr1)
  call addr1;
else if (fptr == addr2)
  call addr2;
else if (fptr == addr3)
  call addr3;
```

In the original CDI, a different sled is associated to each branching site. Indeed, the potential targets explicited by a specific sled are the ones defined by the CFG. Thus, every sled in the resulting program is supposedly potentially different. The advantage of this technique is that both forward — call and jump instructions — and backward — return instructions — edges can be protected in the same fashion. Our approach is a bit different. As explained in Subsection 2.2, we cannot build sleds as precise as the compile-time CDI approach and thus we use the same sled for several branching sites. In fact, since our approach relies essentially on the disassembly of the binary code instead of a CFG, a lot of contextual information is missing. For instance, a call instruction could only target a handful of functions according to the source code. But because we do not have this information, we will consider that this instruction can target the start of any function in the address space.

This difference between the original CDI and our dynamic CDI is important. In our approach, instead of replacing every branch site with a reasonably small sled, we create equivalence classes of branches that are redirected to the same bigger dispatch code. As described in Subsection 2.2, the structure of the code responsible for dispatching branch instructions is a bit more complicated than sleds and goes beyond their initial scope. For this reason, we call our structure a *dispatch table*. The precise organization of a dispatch table is described in Figure 1.

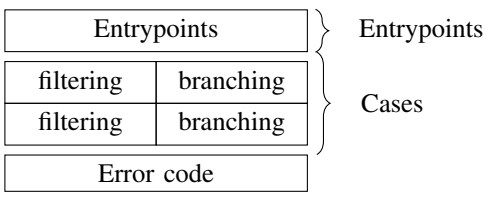

Figure 1. Structure of a dispatch table

In order to make the target process use these dispatch tables, we must modify its binary code. The modification of binary code, especially if it is already being executed, is tedious and requires precision. As a rule of thumb, one cannot really shift code without taking the risk to break the branch instructions nearby. Indeed, direct branching instructions and memory accesses in x86–64 use offsets from the program counter as operands. Moving an instruction or a whole basic-block may make these instructions point to the wrong location.

Moreover, it would not be possible to merely use trampolines to jump to code caves containing the code representing a given sled. Some indirect branches are encoded using fewer bytes than what is needed to encode a jump to a trampoline. For instance (in x86 assembly code), a call to a register is encoded by two bytes while a direct call to a given address needs at least five bytes. As a consequence, not only the indirect branching instructions must be modified, but the whole binary code as well. Since the layout of the code must be modified, instructions accessing memory locations must be translated as well to ensure that they still target the same address as before. For this reason, it is particularly difficulty to modify branching instructions in the code without pushing all the subsequent instructions farther, raising the need to allocate more memory to store the whole program's code.

2.1. Relocation of the code

Because the whole `.text` section of the program needs to be rewritten, we decided to allocate enough contiguous memory in the target process to contain the new code and relocate each basic-block individually in this new section we call `.secure_text`. Relocating each block individually allows our tool to give enough room between each block to be able to translate its last instruction afterwards without messing with the following basic-block. Moving the whole code to another place in memory also allows the user to entirely remove the protection and redirect the process back to its original code, restoring the original performance as well. However, this possibility has not been thoroughly explored and is beyond the scope of this paper.

During the relocation process, the instructions inside each basic-block is also translated in order to preserve their original semantics. Instructions such as `mov` and `lea` are modified so that their operands target the same memory location as in the original code. Note that all instructions rewritten in a basic block take up the same amount of space, except the last branching instruction.

2.2. Removal of indirect branches

As stated in the original CDI paper, indirect branches cannot simply be replaced by direct branches since they can target multiple locations and the target of an indirect branch is only decided at runtime. Hence they are replaced by sleds as described at the beginning of this section.

However, we have access neither to the source code nor compile-time information and thus in general we cannot build a complete yet minimal set of valid targets for the indirect branches within the binary code of the program. For this reason, the set of valid targets is much larger than those computed by a compile-time CDI. Moreover, since the code is relocated, the dispatch tables must not only match the branches operands with possible target addresses, but it must also redirect the branching to the relocated code corresponding to the matched address. For example, suppose that a function at address 0x1024 has been relocated at address 0x2048. The corresponding entry in the dispatch table will be the following:

```
cmp rax, 0x1024
je [cs:0x2048]
```

Depending on the nature of the branch — a function call, a return or just a jump — the sled part of the table is not built the same way. The register used in comparisons with addresses is not the same for function calls and return instructions. Indeed, for function calls, `rax` is preferred as it is not used for arguments and is caller-saved, while for returns, `rdi` is preferred because `rax` is used as the return value and must be preserved at all costs. According to the System V AMD64 ABI, the register `rdi` is call-clobbered, meaning that modifying it before returning from the function should not be a problem. For more information, translation of indirect calls is discussed in Subsection 2.2.1, then translation of indirect jumps is discussed in Subsection 2.2.2 and finally translation of return instructions is discussed in Subsection 2.2.3.

Moreover, if each call site had its own dispatch table composed of every possible function, the memory cost of our approach would be unacceptable. Consequently, we use the same dispatch table for every call sites in the program. However, as a consequence, call instructions with different arguments use the same dispatch table. The comparison of the target address is thus more complicated since this address can be stored in different registers, for example. This problem is easily circumvented by prepending the dispatch table with several entrypoints as shown by Figure 1. There must be one entrypoint per call-instruction kind of operand found in the program. The following listing is an example of entrypoints for a table replacing `call` instructions using `rcx`, `rbx` or `-0x18[rbx]` as operand. This example is made up for the sake of explanation, but a very small program could produce this list of entrypoints.

```
_entrypoints:
  ; entry for call rcx
  mov rax, rcx
  jmp _table_start

  ; entry for call rbx
  mov rax, rbx
  jmp _table_start

  ; entry for call -0x18[rbx]
  mov rax, -0x18[rbx]
  jmp _table_start
_table_start:
  ...
```

These entrypoints copy the operand of the original call instruction into a dedicated register called *preferred register* in the rest of this article. Once the preferred register is set, the process jumps to the beginning of the sled part of the table to perform the comparisons.

2.2.1. Translation of indirect calls. Indirect calls can theoretically target the beginning of every function in the address space of the program. These functions are not only the ones defined in the main binary file of the program but also functions defined in libraries.

The lack of a CFG is a big issue here. It is impossible to retrieve a precise and complete inter-procedural CFG from the static analysis of the binary code in the general case. To avoid false positives, we assume that each indirect call only target the beginning of a function in the program. For this reason, there is only one dispatch table for the whole program and every indirect call instruction is redirected to this table.

Moreover, some call instructions may target an entry in the PLT section, which cannot be relocated as-is due to its dynamic nature. When the code is relocated, some entries may not have been visited yet and the corresponding branch instruction may not have been overridden by the dynamic loader. If the PLT were relocated and the copy used by the process after the relocation, a call to an uninitialized entry would trigger a branch to an unknown location. Indeed, the instruction being relocated, the branch operand would not be an offset to the loader anymore but an offset to an undefined location. In order to prevent that, the PLT is not relocated and its entries are considered valid targets.

A possible solution would be to force the runtime linker to resolve all the symbols at the moment the code begins to be instrumented. The PLT would be complete and its content could be relocated and translated like the rest of the program. We did not explore this solution for this article however.

Finally, another kind of calls occur sometimes that causes troubles to the dispatch table. It is possible that a call instruction targets a function from a shared library but instead of going through the PLT, the call targets directly the function code. This usually happens when the call corresponds to a C++ method call. The following listings give an insight of the phenomenon:

Listing 1. Code in the library

```
struct Class {
  void (*method)();
};

void say_hello() {
  printf("Hello!\n");
}

struct Class class_new() {
  struct Class obj;
  obj.method = say_hello;

  return obj;
}
```

Listing 2. Code in the main binary

```
struct Class my_obj = class_new();
my_obj.method();
```

Here, the pointer `obj.method` contains the actual address of `say_hello`. So calling `my_obj.method` means performing an indirect call to this address directly, bypassing the PLT. Since libraries are often loaded lazily, we cannot add the address of functions located in shared libraries into the dispatch table. Indeed, during the relocation process and when the tables are built, it is possible that some libraries are not loaded yet.

The solution we use in our implementation is to place a trap — `int 3` — at the beginning of the error-handling code at the end of the table. Our tool catches these traps and verifies whether the address in the preferred register is a valid target or not. To do that, our tool finds the library loaded at the address range that contains the given address and checks if this address corresponds to a function using the symbols in the library's ELF file. If the call target is valid, the instruction pointer is set to this address before resuming the process execution, otherwise, the execution continues with the error handling. In our implementation, the code used to handle errors is simply a call of the system call `exit` with a return code 0x42. Nonetheless it could be possible to implement a more complex solution to test more precisely if the given address is in fact a valid target — e.g. for JIT compiled code — but such error-recovery solutions are beyond the scope of this paper.

In conclusion, indirect calls are replaced by direct calls that target the right entrypoint to the call dispatch table. This table consists essentially of pairs of comparison and branch instructions. If no case in the table matches the preferred register, an error code, placed after the cases of the table, is executed.

2.2.2. Translation of indirect jumps. Indirect jumps raise even more concern than indirect calls. While we can safely assume that calls must target the beginning of functions, indirect jumps can target any address in the program code. For this reason, we assume that an indirect jump can either:

- target any address in the basic blocks of the function to which the instruction belongs or;
- be a tail-call optimization, equivalent of a call.

The first hypothesis supposes that the binary code results from the compilation of a program written in a high-level language and not from some manually written assembly code. In that case, an indirect jump may be the result of the compilation of a high-level construct such as a switch statement. Such constructs do not make jumps go outside the function they are part of.

Moreover, since our approach relocates the basic blocks that compose the program, we can safely assume that the target address of a jump must be inside these blocks. This prevents a malicious user from targeting data that is stored in the middle of a function's code.

The second hypothesis states that a jump instruction can be in fact a call to another function. This is called a tail-call optimization. This optimization enables a function to call another function without modification of the stack and letting the callee return from both functions at once using only one `ret` instruction. To do that, a mere jump is used instead of a call instruction. This is particularly useful for functional programming languages that would overflow the stack quickly or spend a lot of time in successive `ret` instructions without such an optimization. In this case, the valid targets of the jump instruction are the first instruction of every function in the program.

These hypotheses allow our approach to have one jump dispatch table for every function in the main binary.

Every indirect jump of the program must be redirected to the dispatch table associated to the function it appears in.

Moreover according to these hypotheses, the structure of the jump dispatch tables is the following: every address in a basic block that is part of the same function as the jump instruction is a valid target and if no match is found, the error code of the table consists of a jump to the main call dispatch table in order to handle tail-calls.

```
_entrypoints:
  ; ...
_table_cases:
  cmp rax, 0x1024
  jl _after_case_one
  cmp rax, 0x1035
  jge _after_case_one
  ; target inside this block
  ; here comes the redirection

_after_case_one:
  ; more cases

_error_code:
  jmp _main_dispatch_table
```

This listing gives an insight of how the jump dispatch tables are organized. In this example, the function has a basic block in the address range 0x1024 to 0x1035. The preferred register is tested against the block's boundaries and if the target address is within these boundaries, the jump can be redirected.

While the preferred register could be tested against the address of every single instruction of the function, we opted instead for a bound check. This trades a little security for performance since a lot fewer comparisons and jumps are performed this way. However, a jump into the middle of an instruction will be considered as valid.

When the preferred register matches a case in a jump dispatch table, the redirected address must be computed using an offset from the beginning of the basic block. Let $bb_{original}$ be the start address of the matching basic block in the `.text` section and bb_{reloc} be the start address of the same basic block in the `.secure_text` section. For a target address $target$, the relocated branch target $target_{reloc}$ is the following:

$$target_{reloc} = target - bb_{original} + bb_{reloc}$$

This computation is done at runtime inside the dispatch table code and an indirect jump to the computed address is performed. The validity of this indirect jump is ensured by the previous comparisons performed against the preferred register. The value of the indirect jump operand is computed right before the jump and is not entirely controlled by the user. The user may control the value enough to target the wrong instruction inside the matched basic block. But this weakness is inherent to our approach and is a consequence of our lack of a precise control-flow graph. Even individual comparisons between the preferred register and the address of each instruction in the basic block could not eliminate this threat. Nonetheless, the previous checks ensure that the preferred register contains an address inside the matched basic block, limiting greatly the possibilities of the attacker.

Since only the last branching instruction of the relocated basic blocks can be modified, the offset of an instruction from the beginning of a block stays the same between the original code and its relocated counterpart. However, we make the hypothesis that instructions in the form `mov reg, [rip+offset]` will not need to be rewritten with an offset that is big enough to prevent the instruction to be correctly encoded. Indeed these instruction access memory addresses relative to the instruction pointer. However, since we relocate the instruction itself, the offset to access the same memory location is not the same any more. As long as this offset can be encoded as a 32-bit value, it can be easily replaced in the instruction binary code.

This limitation could be easily circumvented by making such instructions terminator of basic blocks. Indeed this would allow a proper translation of the instruction without messing with the block's layout. The instructions after this `mov` would be part of a subsequent independent basic block.

2.2.3. Translation of return instructions. Without a precise CFG, we consider that return instructions may target any of the instructions located after a call. Therefore, there is only one return dispatch table for the whole program and every `ret` instruction is replaced by a direct jump to this table. To enhance the security, a shadow stack can be added to the current protection to ensure that the provided return address is the expected return address. We did not add one in our implementation for the sake of simplicity, however.

Unlike the previous tables, the return dispatch table has only one entry point. Indeed, return instructions do not return to an address given as operand but rather retrieve the target address from the stack. Consequently, the preferred register — `rdi` for the return dispatch table — is set using the stack pointer `rsp`. Aside from this particularity, the return dispatch table is organized in the same fashion as the call dispatch table and compares the preferred register with addresses considered valid branching targets.

The real difference between the call dispatch table and the return dispatch table is the set of valid addresses. While the call dispatch table compares the preferred register with addresses from the original code — i.e. from the `.text` section — the return dispatch table compares it with addresses from both the original code and the relocated code. Indeed, our approach can be applied to a program that is already running.

If the target process is executing protected code and calls a function, the return address of this function will be an address inside the `.secure_text` section. This scenario is the most common when a process is protected. However, if our tool has not yet been attached and the target process calls a function, its return address will be in the original `.text` section. If our tool is attached during the execution of the function, it means that this function will continue to execute its modified version inside the `.secure_text` section. When the function will return, it will look for its return address inside the return dispatch table. If we did not store the return-sites from the original code inside the return dispatch table, returns to these addresses would be considered invalid and would prevent the process from running a semantically correct execution.

The solution to this problem is to have two cases per return-site in the table. One is the return-site in the original code, the other is its counterpart in the relocated code. In both cases, the return instruction will target the return-site inside the `.secure_text` section, preventing the process from escaping our protection by returning to its original control flow.

In the following listing, we show a subset of a return dispatch table where a valid return-site address is 0x2048. The corresponding instruction in the original code is located at 0x1024. As we can see, both return-site addresses appear in the table and both cases redirect the control flow to 0x2048.

```
  cmp rdi, 0x2048
  jne _after_case_0x2048
  jmp [cs:0x2048]

_after_case_0x2048:
  cmp rdi, 0x1024
  jne _after_case_0x1024
  jmp  [cs:0x2048]

_after_case_0x1024:
```

At the beginning of the treatment of a return instruction — i.e. translating it to a branch instruction to the return dispatch table —, the return address is popped from the stack into the `rdi` register. At the moment the process enters the return dispatch table, the stack pointer has the correct value as expected after a return instruction.

3. Implementation

We implemented our approach as a command-line tool named DAMAS. DAMAS is written mainly in RUST with a module in PYTHON and a shell script. Its disassembly module is written in PYTHON and uses the ANGR [12] framework. This framework has proven to recover a satisfactory part of the total binary code of programs [10], [9] while being easy to use.

We use a shell script to assemble binary snippets from x86-64 assembler codes. RUST has many libraries available to assemble code inline, but they are relatively complex to use while we only need to assemble simple snippets. For this reason, we wrote this little script that uses GNU `as` to generate a little shared library and then displays it in the standard output. The RUST module responsible for assembly captures this output and lets the rest of DAMAS use it as needed.

DAMAS is based on a library we wrote for DBM, called Sorry [16]. This library is very inspired from Padrone [17], a lightweight DBM framework that only takes control of the target process when needed. Unlike heavier frameworks like DynamoRIO [18] or Pin [19], Padrone and Sorry do not instrument the whole code, only adding the necessary overhead to the process performance.

In order to generate and retrieve statistics about the use of the dispatch tables injected in the target process, we added counters to the tables that are incremented each time the corresponding case is matched.

The purpose of these counters is twofold. First, it enables us to visualize how the tables are used. We can determine precisely if the distribution of the cases reached by the process follows a distribution similar to a uniform distribution or if branching instructions target a specific subset of the potential target addresses. Secondly, it allows us to discriminate the specific cases that the branching instructions target the most. Using this information, we can optimize our tables to minimize the number of superfluous instructions executed to perform a branch — i.e. make the dispatch table reach the actual branch instruction faster.

4. Optimizations

The addition of redirection tables in the code to replace a single branch instruction eventually introduces an overhead in the performance of the process. This overhead worsens as a table grows. In its naive form, a table is structured linearly, putting each case one after the other. In order to reach the 400th case of a table, the process must pass by the 399 previous ones. In large programs, such as web servers, this becomes a real concern.

Consequently, we have designed dynamic optimizations to change the layout of a table during the execution of the process in order to make it faster to traverse. We have focused our effort in two main optimizations: sort of cases and a tree representation. Both approaches are detailed in the following subsections.

These optimizations are applied during the runtime of the target process using dynamic binary modification. We use a dynamic profiling of the use of the tables to find the appropriate optimization scheme in a similar way as `FittChooser` [20].

4.1. Case sort

The tables used in our approach are very large and contain many cases since we over-approximate the possible targets of each branch. This ensures that the process will continue to execute normally and never encounter a false positive. However, a lot of cases in the tables will never be matched because the semantics of the program would never trigger them. As a consequence, some cases may become privileged targets of branches and some other cases will never be reached. Unfortunately, the most matched cases are not necessarily the ones that appear at the beginning of the table. This is especially true for the call dispatch table whose first cases correspond to the PLT entries which may not be the most indirectly-called functions. Figure 2 (top) illustrates the distribution of the return table for sqlite3: the most frequent entry (7,199,290 hits) requires 3,200 comparisons before a match, and most cases are never matched. After sorting (bottom), entries with high probability are compared first.

To circumvent this problem, we profile the use of the tables in order to identify the most used cases and sort the table so that they appear at the beginning of the table, making them much faster to reach. Once a table has been sorted it is re-injected into the target process.

4.2. Tree representation of tables

The first and naive form we designed for dispatch tables was to put each case one after the other ordered by the original address of the branch target — i.e. the hardwritten address we compare the register to, that is the

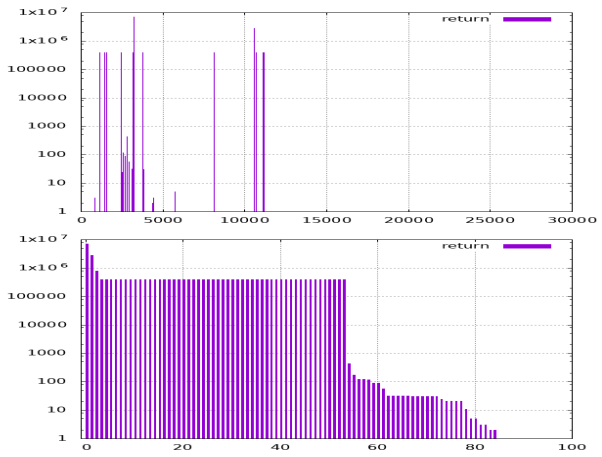

Figure 2. Distribution of return table for sqlite3, unsorted and sorted. The x-axis represents the distinct return instructions. Note the y-logscale, and the different range on the x-axis: the bottom graph is a zoom on executed return instructions. All others are never executed.

address the branch should have targeted in the absence of DAMAS. While this ordering may be not optimal as described in the previous subsection, the use of the table may not be as unbalanced as some extreme cases. For instance, the call dispatch table may be relatively equally used and no really preferred targets may be identified. In this case, a simple sort of the table is not sufficient.

For this reason, we totally changed the layout of our table, giving it the form of a balanced binary tree. Instead of traversing the whole table linearly, the process can navigate faster through the tree and reach the appropriate case in less time — $O(\log_2 n)$ instead of $O(n)$.

When DAMAS is first deployed on a process, no runtime information about branching targets is available. Consequently, using this tree representation as the default representation for tables allows better performance. During the execution of the process, if some table shows a use that clearly privileges cases over the other, a sorted linear representation can be injected to replace the original tree representation, refining further the performance overhead induced by our approach.

5. Limitations

DAMAS is a prototype developed in the context of our research, hence it comes with some limitations. First of all, the library it relies on, Sorry, makes most of its manipulations in the target process using the C interfaces of system calls such as `malloc`, `mprotect`, etc. As a consequence, DAMAS can only be deployed on programs that are dynamically linked and use the C library. For instance, programs written in Go or Pascal would not be supported. Fortunately, the C library is linked by many programs written in common languages including C itself, C++, OCaml, and RUST. Further development of Sorry and DAMAS could easily lift this issue, replacing references to the C API to calls to the actual system calls. In addition to this problem, it appears that C++ exceptions are not handled properly, limiting even more the programs supported by our tool.

Second, many big programs still crash because of segmentation fault while under the protection of DAMAS. Even though we do not know the precise cause of these crashes, we expect them to be solved with a more thorough debugging of DAMAS, hopefully promoting it from research prototype to an actually usable tool.

Moreover, programs using JIT compilation cannot be protected. JIT code does not exist in the binary file of the program by definition, preventing DAMAS from disassembling it and make it part of the known binary code of the program. Therefore, no dispatch table may contain references to JIT compiled code, leading to false positives. Moreover, analyzing the code after each JIT code generation could give an attacker the possibility to abuse DAMAS and make it accept code they crafted through the JIT compiler. While a solution to these concerns could be found, it goes beyond the scope of this paper.

6. Results

The impact of DAMAS on performance has been measured against multiple programs. The testing platform consists in an Intel Xeon CPU E5-1603 v4 @ 2.80 GHz running a Fedora 34 with Linux kernel 5.11.15-300.fc34.x86_64. Every benchmark has been run inside a Docker container that contained: DAMAS, the target process, the benchmark client such as the command `time` or `ApacheBench` and any necessary dependencies.

We chose a diverse set of programs to make the measurements with the intention of showing the different impacts DAMAS has on performance according to the kind of application it is protecting. Our dataset contains CPU-intensive programs such as compression software GZIP and BZIP2, server applications like the HTTP server NGINX and the MQTT server MOSQUITTO as well as TCC — the tiny C compiler — and the SQL database engine SQLITE3.

The compiler TCC has been benchmarked twice with two different inputs. First, we used the source code of SQLITE3 which is an amalgamation of C code into three files. Then, as explained in detail in Subsection 6.1, since the execution time was too short, we used the source code of SSHD as input for a more complete evaluation.

Moreover, since a lot of work in NGINX is done in shared libraries instead of inside the main binary, we compiled another version of the software with most of its dependencies linked statically, only leaving the C library and the dynamic loader as dynamic linked libraries (as needed by DAMAS). This way, most of the treatments are performed inside the main binary, forcing DAMAS to relocate and translate this code in order to consider the impact of our solution on these parts of the software as well. This build of NGINX is referred to as NGINX-STATIC.

In these benchmark scenarii, DAMAS is attached to the target process when it reaches the `main` function and stays attached during the whole execution of the process. This way, we can consider the impact of DAMAS on the target process during its entire execution. We can have more trustworthy, precise and reproducible measurements with this method than with an evaluation of arbitrary parts of the execution. Only the target process execution time is evaluated, the time spent by DAMAS is not considered. Indeed, DAMAS is supposed to be a protection that can be added to a running software and that most of its preparation procedures can be done in parallel to the target

Program	Reference time (s)	Damas (linear) time (s)	Damas (linear) overhead	Damas (tree) time (s)	Damas (tree) overhead	Damas (presorted) time (s)	Damas (presorted) overhead
BZIP2	198.86	935.24	×3.7	226.70	+14%	217.44	+9%
GZIP	68.38	250.52	×2.66	94.15	+38%	81.15	+19%
NGINX	501.69	501.89	+0%	500.68	+0%	499.74	+0%
NGINX-STATIC	495.31	496.74	+0%	496.10	+0%	496.91	+0%
MOSQUITTO	4.50	19.89	×3.42	4.74	+5.33%	4.70	+4.44%
SQLITE3	1.32	626.61	×473.71	2.25	+70.45%	2.07	+56.81%
TCC							
sshd	2.38	29.04	×11.2	2.87	+20.58%	3.01	+26.47%
sqlite3	0.14	7.85	×56	0.28	×2	0.42	×3

TABLE 2. AVERAGE EXECUTION TIMES OF THE PROGRAMS OF THE DATASET AND OVERHEAD TO THE REFERENCE EXECUTION TIME.

execution, therefore the only reason why the program is stopped in our scenarii is for measurement purpose.

Some programs of our dataset — BZIP2, GZIP, TCC and SQLITE3 — have been benchmarked using the `time` command, since their purpose is to take input, realize a treatment, give output and exit. The servers however have been benchmarked using specialized clients in order to get more meaningful metrics such as time taken per request, latency, etc. NGINX has been benchmarked with APACHEBENCH [21] and MOSQUITTO with MQTT-BENCHMARK [22].

The Docker images were run several times to ensure the validity of our measurements. The tested program inside the Docker image has been executed four times per run. First, the program was run without protection for reference. The three other times, DAMAS was attached at the startup of the program and used respectively unsorted linear tables, tree tables and presorted linear table. In order to have presorted tables, the previous execution of DAMAS using unsorted linear tables logged the values of the dispatch-table counters at the end of the target process execution and sorted the cases of each tables. This configuration was then provided to sort the tables at the beginning of a new execution.

While it is possible to run DAMAS without counters, every execution uses them, even when they are technically useless — e.g. with the tree tables — to ensure that the naive linear table executions are not unfairly penalized by the counters. Nonetheless, the counters take the form of 32-bit integer arrays and an `inc` instruction per case in each table that is executed only when the case is matched, therefore the impact of counters on performance is negligible.

The average execution runtime and the overhead compared to the reference are shown in Table 2. The average execution times are arithmetic means of the sampled execution times and the overhead are computed as follows (and expressed as percentage for clarity):

$$overhead_{execution} = \frac{time_{execution}}{time_{reference}} - 1$$

6.1. Performance evaluation

As expected, DAMAS imposes a big overhead on the performance of the most CPU-intensive programs in our dataset, especially when it is deployed with the naive linear dispatch-table representation. This representation causes the execution time of these programs to be multiplied, such as TCC which was slowed down by a factor of 11×, or even the most extreme case, SQLITE3 with an slowdown of 473×.

These programs present a massive impact of DAMAS on their performance, as shown in Figure 3. The boxplots give the following pieces of information:

- the median;
- the lower and upper quartiles;
- the extreme line is 1.5× the interquartile range.

The slowdown disappears when a more optimized representation of dispatch tables is used. Typically, on BZIP2 and GZIP the tree representation allows to reduce the overhead in execution time to an average of respectively 14% and 38% while the more suited presorted tables enable a further reduction of the overhead to respectively 9% and 19%.

The large overhead observed in TCC when compiling SQLITE3 — 100% for the tree tables and 200% for the presorted tables — can be explained by the short execution time. It is possible that DAMAS' impact on performance is not perfectly linear and that a minimal non-compressible overhead occurs. For instance, function returns may have become a significant part of the execution of TCC in such a small execution. Measurements with very large compilation units can answer this question. We have also measured the impact of DAMAS on TCC when compiling SSHD that has a bigger codebase. As expected, the overhead introduced by DAMAS is much smaller.

It is nonetheless difficult to find other large projects that can be compiled with a command line in the same fashion as the following listing.

```
tcc -o executable *.c -ldependency
```

Indeed, modular compilation is not interesting in our testing scenario, we want only one long execution of TCC. Moreover, this compiler is renowned for its fast compilation time, making the task even harder.

6.1.1. NGINX and NGINX-STATIC. Unlike with the CPU-intensive programs, DAMAS did not impact NGINX's performance. Since this server is supposed to be very IO intensive, a very small overhead was expected as opposed to CPU-intensive that suffer much more from a large amount of additional branches.

However, this almost inexistent overhead raised our concerns. According to APACHEBENCH logs and manual verifications with a web browser, the server works as intended and serves the requested pages. Moreover, DAMAS

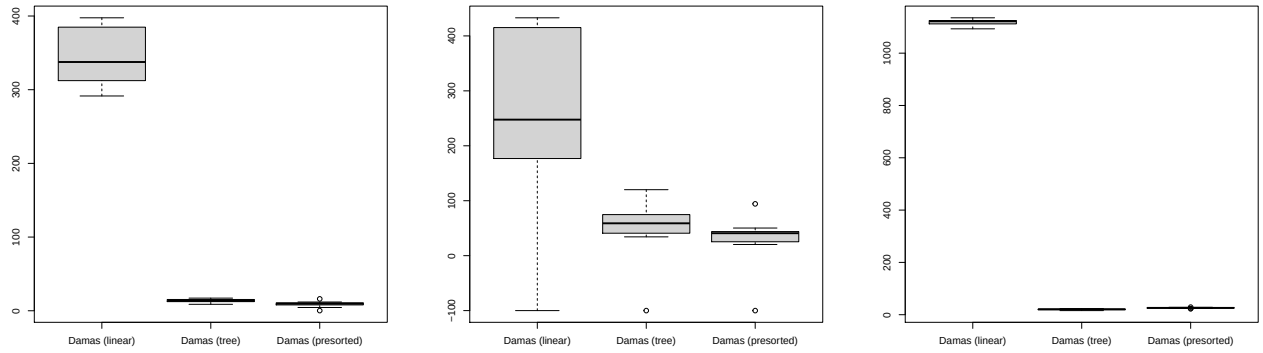

Figure 3. Overhead in execution times in percentage of the most CPU intensive programs. Left is BZIP2, middle is GZIP and right is TCC.

did not warn us about any indirect branch instruction that could not be translated — leaving an indirect branch inside the code that could circumvent our protection and branch back to the original control flow. In addition to that, we checked the whole execution control flow to be sure that the process never branches back to its original control flow and thus escapes our protection.

Finally, our results show no difference between NGINX and NGINX-STATIC both in terms of execution time of the reference and in terms of overhead. In conclusion, it seems that the impact of IO operations is big enough to completely hide the impact of DAMAS regardless of the difference in code coverage made by the static linkage of most libraries used by NGINX.

6.1.2. Tree representation and sorted tables. Some programs of our dataset benefit more from the tree representation of their dispatch tables from presorted linear tables and some do not. The reason is that the programs that benefit from presorted tables have a very biased use of the tables. Indeed only few cases of the most used tables match the preferred register during the execution of these programs, the other cases are often never matched, ensuring that a linear traversal of such a table is faster than a traversal of a binary tree for the entire table.

However, the biggest table — i.e. the ret dispatch table — in a process instrumented by DAMAS usually does not fit this description. Indeed, return instructions are always indirect by design, forcing the process to use the table for each call it made beforehand, whether it was direct or indirect. While indirect branches can have a very precise set of targets in a program — e.g. C++ virtual methods, switch statements, etc —, the set of every direct calls in the program usually induces a more balanced use of the return dispatch table.

7. Conclusion and future works

In this paper, we presented an approach against control-flow hijack attacks that can be deployed at runtime on a process. Our approach is based on control-data isolation [8] and aims to remove every indirect branching in the program code. The removal of these branches is done using dispatch tables that must compare an effective branch target address with valid potential target addresses and performing the corresponding direct branch.

A first implementation of our approach with a linear traversal of the dispatch tables showed an unacceptable impact on the performance of the target process. However, simple optimizations such as sorting the table in reverse order of use and a tree representation of these tables instead of the linear representation improved significantly the impact of our approach on performance.

In order to further reduce the impact of our solution, we will implement a dynamic monitoring of the process execution in order to get traces that will help choose the best representation for each table individually depending on how it is used. Each table of the process could have the most fitted representation unlike the current implementation.

For instance, a tree representation of tables by default for the tables improves the overall performance of the program compared to the unsorted linear representation. Nonetheless, the process may not target each case of one of the most used tabled uniformly and some cases of a table may be overused compared to the others. In such cases a sorted linear representation for this particular table may offer better performance. Moreover, the cases of the table will be sorted according to their use in the current execution of the program and not according to a previous execution, providing an even better fitting order of the cases of the tables.

In order to make the choice of representation, we could use the Bhattacharyya distance to estimate how close the use of a table is to a uniform distribution. For probability distributions p and q over the same domain X, the Bhattacharyya distance is defined as

$$D_B(p,q) = -\ln(BC(p,q))$$

where BC is the Bhattacharyya coefficient for discrete probability distributions defined as

$$BC(p,q) = \sum_{x \in X} \sqrt{p(x)q(x)}$$

Moreover, we will further enlarge the scope of programs our solution can support. First, we will improve the support of multithreaded applications by making DAMAS aware of the different threads of the target process instead

of relying blindly on the PID of the main thread. Threads of an already running process will have to be protected properly as well.

Second, the current experiments focus primarily on the impact of the dispatch tables and their different representations in terms of code. We would like to measure the impact of the relocations and table traversals on the branch predictor and in terms of cache misses, etc. Therefore, we will analyze further the execution traces of processes protected by DAMAS using performance counters of the CPU with the intention of reducing even more the impact of our approach on the performance of the program.

Finally, it would be interesting to implement a functionality to remove entirely our protection from the target process. Such a functionality would release all the memory allocated by DAMAS and redirect the execution back to the original control flow.

In conclusion, the effort will be put in priority on improving performance as well as the range of applications our solution can support to widen its potential adoption.

Acknowledgement

This work is partially funded by a grant from DGA — France's Ministry of Armed Forces — and takes place within the missions of the Pôle d'Excellence Cyber.

References

[1] TIOBE - The Software Quality Company, "TIOBE index for April 2021," https://web.archive.org/web/20210502041035/https://www.tiobe.com/tiobe-index/, 2021. [Online]. Available: https://web.archive.org/web/20210502041035/https://www.tiobe.com/tiobe-index/

[2] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, Mar. 2012. [Online]. Available: https://doi.org/10.1145/2133375.2133377

[3] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, p. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[4] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: A new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11. New York, NY, USA: ACM, 2011, p. 30–40. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[6] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[7] M. Payer and T. R. Gross, "String oriented programming: When ASLR is not enough," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, ser. PPREW '13. New York, NY, USA: ACM, 2013. [Online]. Available: https://doi.org/10.1145/2430553.2430555

[8] W. Arthur, B. Mehne, R. Das, and T. Austin, "Getting in control of your control flow with control-data isolation," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 79–90.

[9] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Krügel, and G. Vigna, "Ramblr: Making reassembly great again," in *NDSS*, 2017.

[10] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," *ArXiv*, vol. abs/2007.14266, 2020.

[11] SA Hex-Rays, "IDA Pro." [Online]. Available: https://www.hex-rays.com/ida-pro/

[12] F. Wang and Y. Shoshitaishvili, "Angr – the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 2017, pp. 8–9.

[13] National Security Agency, "Ghidra." [Online]. Available: https://ghidra-sre.org/

[14] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*. Springer, 2011, pp. 463–469.

[15] "Radare2." [Online]. Available: https://rada.re/n/

[16] "Sorry." [Online]. Available: https://gitlab.inria.fr/klebon/sorry

[17] E. Riou, E. Rohou, P. Clauss, N. Hallou, and A. Ketterlin, "PADRONE: a Platform for Online Profiling, Analysis, and Optimization," in *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, Vienne, Austria, Jan. 2014. [Online]. Available: https://hal.inria.fr/hal-00917950

[18] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *International Symposium on Code Generation and Optimization, 2003. CGO.*, 2003, pp. 265–275.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," *SIGPLAN Not.*, vol. 40, no. 6, p. 190–200, Jun. 2005. [Online]. Available: https://doi.org/10.1145/1064978.1065034

[20] A. A. Ap, K. Le Bon, B. Hawkins, and E. Rohou, "FITTCHOOSER: A dynamic feedback based fittest optimization chooser," in *2018 International Conference on High Performance Computing Simulation (HPCS)*, 2018, pp. 98–105.

[21] "ab: Apache HTTP server benchmark tool." [Online]. Available: https://httpd.apache.org/docs/2.4/en/programs/ab.html

[22] "MQTT benchmarking tool." [Online]. Available: https://github.com/krylovsk/mqtt-benchmark