Dam: A Practical Scheme to Mitigate Data-oriented Attacks with Tagged Memory based on Hardware

Mengyu Ma, Liwei Chen[*], Gang Shi
Institute of Information Engineering, Chinese Academy of Sciences
School of Cyber Security, University of Chinese Academy of Sciences
Beijing, China
{mamengyu, chenliwei, shigang}@iie.ac.cn

***Abstract*—The widespread deployment of unsafe programming languages such as C and C++, leaves many programs vulnerable to memory corruption attacks. With the continuous improvement of control-flow hijacking defense methods, recent works on data-oriented attacks including Data-oriented Exploits (DOE), Data-oriented Programming (DOP), and Block-oriented Programming (BOP) have been showed that these attacks can cause significant threat even in the presence of control-flow defense mechanism. Moreover, DFI (Date Flow Integrity) is a software-only approach for mitigating data-oriented attacks, while it incurs a 104% performance overhead. There are no suitable defense methods for such attacks as yet. In this paper, we propose Dam, a practical scheme to mitigate data-oriented attacks with tagged memory based on hardware. Dam is a novel approach using the idea of tagged memory to break data-flow stitching and gadgets dispatcher of generating data-oriented attacks rather than complete DFI. By enforcing security checking on memory access, Dam eliminates two requirements in constructing a valid data-oriented attack. We have implemented Dam by extending lowRISC, a RISC-V based SoC (System of a Chip) that implements tagged memory. And our evaluation results show that our scheme has an average performance cost of 6.48%, while Dam provides source compatibility and strong security.**

***Keywords*—memory safety, data-oriented attacks, tagged memory, hardware-assist.**

I. Introduction

Memory corruption vulnerabilities are still an overwhelming threat to computer systems and software due to the widespread deployment of C/C++ programs. Memory data can be divided into control-related and non-control-related data types from the functional perspective, so there are two main ways of memory corruption attacks including control-flow hijacking attacks [1], [2] and non-control-data attacks [3]. With the continuous improvement of control-flow hijacking defense methods such as Data Execution Prevention (DEP) [4], Address Space Layout Randomization (ASLR) [5], and Control Flow Integrity (CFI) [6]. Non-control data attacks are valued gradually. Recently, researchers have presented automatic generation of Data-oriented Exploits (DOE) [7], Turing-complete Data-oriented Programming (DOP) [8] and a new code reuse technique based on Block-oriented Programming (BOP) [9], so-called data-oriented attacks. It has been showed that these attacks can cause significant threat even in the presence of control-flow defense mechanisms. Moreover, there are no practical defense methods for such attacks as yet.

In contrast to control-flow hijacking attacks, data-oriented attacks tamper with data variables and data pointers of target programs to execute the malicious behavior. Because there are no changes about the address for transfer instructions, a common feature of data-oriented attacks is that they do not change the control flow of the target program. Therefore, data-oriented attacks not only can bypass mainstream defense mechanisms based on control flow integrity but also be as powerful as control-flow hijacking attacks. In addition, compared to traditional non-control-data attacks, data-oriented attacks do not rely on any specific security-critical data or functions (like system call parameters or printf-like functions). Instead, they reuse abundant data-oriented gadgets to build function payloads. It is more difficult to prevent data-oriented attacks because protecting all data variables will incur high-performance overhead.

Several previous works [8], [10] demonstrated that data-oriented attacks can be prevented soundly by Data Flow Integrity (DFI) [11]. DFI instruments the program to check whether source locations of writing operations are legal at the time of every reading operations. Besides, the whole memory safety [12], [13] is another solution that can effectively defend against data-oriented attacks. They aim to eliminate memory errors and prevent adversaries from gaining the ability to access memory illegally. However, almost all of these solutions almost suffer from high-performance overhead. Complete DFI has a 104% performance-overhead, and the average performance loss of SoftBound+CETS is 116% [8].

Considering the above-stated challenge for DFI and memory safety in preventing data-oriented attacks, we propose Dam, a practical scheme to mitigate data-oriented attacks with tagged memory based on hardware. Compared to complete DFI, we only focus on necessary conditions for launching a valid data-oriented attack. In this paper, we first propose a scheme using the idea of tagged memory to break the requirements in generating data-oriented attacks.

To summarize, our contributions are:

- *Requirements of Data-oriented Attacks*: We conclude two requirements in constructing a valid data-oriented attack, they are data-flow stitching and gadgets dispatcher.

[*]Corresponding author

- *Security Enforcement*: We propose two necessary rules to break the data-oriented attacks. Rule1 is that data pointers should point to legal memory locations, which we called a trusted set of data pointer objects (legitimate DPO). Rule2 is that critical variables should be defined by legal memory access operations, which we called legitimate decision variables (legitimate DV).
- *Low Cost*: We employ a hardware-assisted tagged memory architecture to accelerate runtime security checks of memory access, which can reduce performance overheads.

II. Background and Related Work

A. Data-oriented Attacks

Traditional non-control-data attacks corrupt a variety of data including user identity data, configuration data, user input data, and decision-making data [3], which causing privilege escalation or sensitive data leakage. However, the implementation of these attacks need some application-specific knowledge such as the precise memory address of target data and maybe indirectly affect the control flow of programs. For this, researchers proposed a new type of non-control-data attacks such as DOE, DOP, and BOP. This paper refers to them as data-orient attacks. Next, we will introduce the main ideas of these three attacks in detail.

1) Data-oriented Exploits (DOE): DOE [7] developed a new technique called data-flow stitching, which systematically finds ways to join data flows in the application to launch data-oriented exploits. The key idea of data-flow stitching is to efficiently search for a new data-flow edge set to stitch the source and target vertices, eventually able to manipulate the sensitive data result in information disclosure or privilege escalation. The algorithm of stitching first needs to check whether each memory vertex in the source flow and target flow pointed by another vertex. Then, to stitch the data flows, the algorithm uses the stitch-able pointers on the target flow to change the pointer to point to a vertex in the source flow. The Flow Stitch mentioned is the first tool to systematically generate data-oriented attacks without violating control flow integrity. And it is capable of exploiting automatically by using multi-edge stitch, which is also the focus of this paper.

2) Data-oriented Programming (DOP): DOP [8] demonstrated that non-control-data attacks are Turing-complete for the first time. It means that we can construct expressive non-control-data exploits for an arbitrary program in specific architecture. Similar to ROP, the key idea of DOP is to find data-oriented gadgets (short sequences of instructions in the program's control-abiding execution that enable specific operations simulating a Turing machine). The exploit process of DOP is roughly divided into two parts. First, attackers need to identify data-oriented gadgets to perform malicious goals (e.g., assignment, arithmetic, and conditional decisions). Second, the adversary needs to find dispatch gadgets to chain together disjoint function gadgets in an arbitrary code sequence. Note that there is also a selector controlled by this adversary through some memory errors to execute a subset of gadgets, which repeatedly corrupts the target variables to set up the execution of gadgets. Moreover, different function gadgets take the output of the previous gadget as input by selectively corrupting operable data pointers.

3) Block-oriented Programming (BOP): Compared with DOP, BOP [9] is a new type of code reuse technique that utilizes basic blocks as gadgets along with legal control flow path in a binary to launch data-oriented attacks. The significant contribution to BOP is the block-oriented programming compiler (BOPC). BOPC is the first compiler technique to automate the data-oriented attacks with the arbitrary memory write vulnerabilities. An adversary can automatically generate attack payloads using BOPC in theory. Similar to other data-orient attacks, there are also two main steps to leverage an exploit. First, attackers need to select functional blocks using BOPC. Unlike the concept of previous functional gadgets, functional blocks in BOP means that some basic blocks from the target program that implement individual SPL statements (BOPC provides an exploit programming language, called SPL). Second, BOPC searches dispatcher blocks to connect pairs of neighboring functional blocks, which enables to simulate the BOP chain to produce a payload that implements malicious behaviors.

B. Defense for Data-oriented Attacks

1) Data Flow Integrity: Data Flow Integrity (DFI) [11] first uses static analysis to generate the data-flow graph (DFG) that expresses variables of define-use relationship. Then, DFI instruments the program to check whether each memory location is defined by DFG before reading operations. Therefore, complete enforcement of DFI can defense all data-oriented attacks. However, the performance overhead of complete DFI cannot be applied in practice. Write Integrity Test (WIT) [14] is an optimization of DFI in a sense. It also needs a static policy generated by a compiler. Different from DFI, WIT instruments the program to check whether the specific object is legal before write operations instead of reading operations. But it still cannot be practical. Recently, HardScope [10] presented run-time scope enforcement to mitigate all currently know DOP attacks by enforcing compiler-time memory safety constraints. It protects some critical variables in C programs at function granularity. There is still an attack surface for data-oriented attacks.

2) Memory Safety: Because memory corruption attacks typically utilize memory errors to make a pointer out-of-bounds or dangling, which allows the attacker to gain the ability to read and write arbitrary memory addresses. Therefore, the memory safety aims to eliminate memory errors that can be exploited by an attacker to illegally access memory address. By checking boundary and matching identifier, SoftBound [12] and CETS [13] store metadata of each pointer (base, bound, lock, and key) to enforcement the complete memory safety. However, complete memory safety always suffers a high-performance overhead.

3) Heuristic Methods: Lite HAX [15] and OEI (Operation Execution Integrity) [16] aim to detect control-flow attacks as well as DOP attacks for embedded devices. Lite HAX tracks and records the fine-grained control- and data-flow events of executing programs at runtime with hardware supports. OEI enables selective verification of both control-flow integrity and critical-variable integrity by adding some comments from the programmer. These heuristic solutions often are required to trade-off security and performance. Although they sacrifice some security features and need a lot of manual work, they have an optimistic performance overhead. HDFI (Hardware-Assisted Data-flow Isolation) [17] just is an isolation mechanism that extends each memory unit with an additional tag. Since HDFI only supports one-bit tags in order to use easily, it supports very simple security policies. Dam provides critical data-flow integrity instead of the idea of isolation.

C. Tagged Memory Architecture

A technique usually called "memory tagging" may dramatically improve the situation if implemented in hardware with reasonable overhead [18]. The tagged memory mechanism augments each data word in memory with a small piece of extra metadata, a tag. lowRISC [19] is a not-for-profit project creating a fully open-sourced, Linux-capable, RISCV-based, 64-bit SoC (System-on-a-Chip), that aims to implement fine-grained memory access restrictions by using tagged memory.

Tagged memory provides a flexible security framework in the form of a design-time configurable number of tag bits to each 64-bit word in memory. These tag bits are copied along with the data word through the cache hierarchy, which means each word in the L1 data and L2 cache lines are augmented with additional tag bits. It adopts a multi-tree structure where leaf nodes at the tag table level hold the actual tags. Moreover, Fig. 1 illustrates the hardware architecture of lowRISC based on extensions to the Rocket core [20]. Tag check units (tagChck) and tag processing units (tagProc) are added to the L1 data and L2 cache pipeline along with each memory access, which can be enabled at run-time. Tags must also be associated with data in the main memory. For this implementation, tags are stored in a reserved area of physical memory. The tag cache also performs the role of the original TileLink converter.

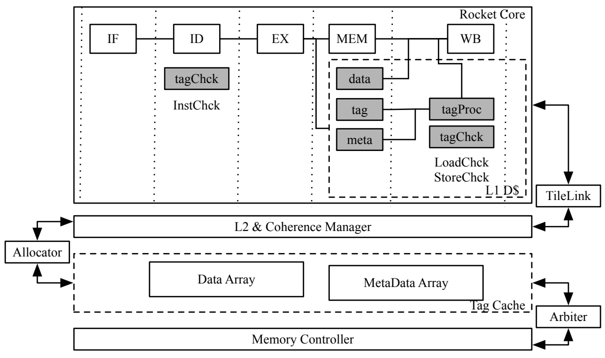

Fig. 1. Hardware architecture of lowRISC

There are some ISA extensions are provided to support the tagged memory according to the manual of lowRISC. Two new instructions have been added for loading and storing tags. And there are various tag functions supported in lowRISC to perform tagChck and tagProc. As shown in Table I (rd: destination register, rs1: 1st source register, rs2: 2st source register, imm: immediate number, address offset, DW: Double Word).

TABLE I
SUMMARY OF THE VARIOUS TAG FUNCTIONS SUPPORTED IN LOWRISC

Function	Description
ltag rd, imm(rs1)	load the tag associated with DW located at rs1 + imm to register rd.
stag rd, imm(rs1)	store the tag in rd to the DW located at rs1 + imm.
Load tag check	check if certain bits are set in the memory tag.
Store tag check	check if certain bits are set in the memory tag.
Load tag propagation	propagate tags from memory to the destination register.
Store tag propagation	propagate tags from rs2 to memory.

Moreover, RISC-V provides the riscv-tests [21] to count the number of CPU cycles consumed by defining the way the test program starts and ends execution. The test program will start execution at the first instruction after *RVTEST_CODE_BEGIN*, and stop when execution reaches the *RVTEST_CODE_END* macro, which is implicitly a success.

D. Threat Model and Assumptions

We describe our threat model and assumption in this section. The goal of this paper is to mitigate currently known data-oriented attacks according to characteristics of data-flow stitching or gadgets dispatcher. Therefore, we only focus on data-oriented attacks that utilizing possible gadgets inside executables and shared libraries. We assume that an attacker can freely and repeatedly interact with the program at the user-level and on purpose of generating data-orient attacks through one or more memory vulnerabilities. Nonetheless, the kernel services are trusted and they cannot be subverted. There are some security mechanisms such as DEP and NX provided by the OS, which makes the code pages are read-only to prevent code injection attacks. Perhaps the system also enforcement with CFI or other methods to prevent control-flow hijacking attacks. Besides, we assume that the results of static analysis obtained from protected applications are derived in a secure environment. Thus, the results of static analysis are trusted.

III. REQUIREMENTS IN GENERATING DATA-ORIENTED ATTACKS

Based on the description in section II-A, we summarize the characteristics and necessary conditions of data-oriented attacks. And we conclude two requirements in constructing a valid data-oriented attack, namely data-flow stitching and gadgets dispatcher. As shown in Fig. 2.

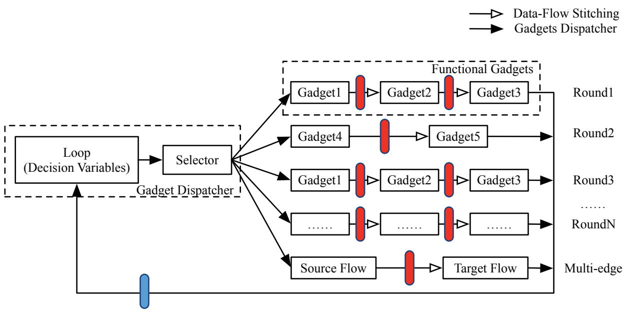

Fig. 2. The design model of Data-orient Attacks

A. *Data-Flow Stitching*

Data-oriented attacks need to manipulate data-flow paths to stitch the source and destination. The key to find multi-edge exploits is stitching data pointers to approach sensitive variables or perform functional goals. Specifically, DOE [7] is required to stitch different vertices. DOP [8] is required additional manual analysis to stitch data-oriented gadgets that enable attackers to construct malicious behaviors. The workable method is to find gadgets with dispatcher surrounding them as much as possible, which means that attackers need selectively corruption operand data pointers and decision variables. BOP [9] implements automated verification for the chain of the data-oriented gadget using BOPC, and the principle of data-flow stitching is the same as DOP.

We conclude that one requirement in generating data-oriented attacks is data-flow stitching using data pointers that can be controlled by attackers. As indicated by the red mark in Fig. 2. For this, blocking stitch of the data-flow path can prevent sensitive variables from being approached by adversaries, and blocking stitch of data-oriented gadgets can break the construction of payloads.

B. *Gadgets Dispatcher*

Also, data-oriented attacks need to schedule different functional gadgets since loops are necessary for attackers to repeatedly connect gadgets. Compared with data-flow stitching, gadgets dispatcher mainly used to perform Turing-complete simulation operations mentioned in DOP [8] and BOP [9].

We conclude that the other requirement in generating data-oriented attacks is gadgets dispatcher using the corrupted decision variables. As shown by the blue mark in Fig. 2. Thus, one solution for mitigating data-oriented attacks is to ensure the integrity of decision variables.

IV. Mitigation Scheme

We propose Dam, a practical scheme to mitigate data-oriented attacks with tagged memory based on hardware. Dam aims to provide low-cost and source-compatible security enforcement against data-oriented attacks. Different from the complete memory protection, this paper aims to provide a comprehensive detection mechanism with the support of hardware architecture rely on the common characteristics of data-oriented attacks.

The design framework of Dam as shown in Fig. 3. We first analyze the source code of target programs in compile time. Second, we perform the dynamic checking using security strategies including legitimate DPO (Data Pointer Objects) and DV (decision variables). Moreover, the overall framework is built on the tagged memory architecture to support our tag allocator. The allocator can convert security strategies to memory tags. And then we will introduce the rules of data-flow and the structure of tagged memory in detail.

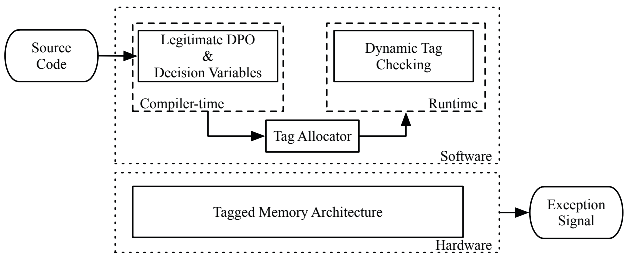

Fig. 3. The design framework of Dam

A. *Legitimate DPO (Data Pointer Objects) and DV (Decision Variables)*

To break the data-oriented attacks, we propose two necessary rules as follows:

- *Rule1*: Data pointer in the target program should point to legal memory locations. We call Rule1 as a trusted set of data pointer objects (legitimate DPO).
- *Rule2*: Decision variables in the target program should be defined by legal memory access operations. We call rule2 as legitimate decision variables (legitimate DV).

To deploy above two rules, we further explain the meaning of legitimate DPO and DV. And we take the code snippet of Fig. 4 as an example.

```
//p = &a, q = &b
char* swap(char **p, char **q){
    char* t = *p; //t = a
    *p = *q;      //a = b
    *q = t;       //b = t
    return t;     //ret a
}
void main(){
    char a1, b1;
    char *a = &a1;
    char *b = &b1;
    char *c = swap(&a, &b); //c = a
}
```

Fig. 4. Code snippet for the swap function

For legitimate DPO, we use the Andersen algorithm to solve the constraint relationship between different data pointers so that we can obtain the set of objects pointed by the data pointer. In the sample code above, there are six sets as follows, corresponding six legitimate DPOs. For instance, the pointer *a* in the main function points to variables *a1* and *b1*. Same as pointer *b, c, p, q*, and *t*.

1) main@a={main@a1, main@b1}.
2) main@b={main@b1, main@a1}.
3) main@c={main@a1, main@b1}.
4) swap@p={main@a}={main@a1, main@b1}.
5) swap@q={main@b}={main@b1, main@a1}.
6) swap@t={main@a1, main@b1}.

For legitimate DV, we identify to the last definition site of the decision variable using the def-use chain. Then, we mark this site as the legitimate DV. It is intuitive to infer that the pair of this decision variable and its last definition site is legal, and the other situations are illegitimate.

B. Tagged Memory Allocator

Our scheme verifies the consistency of tagged memory for critical variables (data pointers and decision variables in this paper). Inspired by HDFI [17], normal operations (tag1 = 0) access to critical variables is illegal, which will trigger an exception signal and terminate the program. Furthermore, is used to check the validity of the source operand.

The structure of tagged memory allocator as shown in Fig. 5. First, tag1 is a 1-bit tag that indicates whether the memory location is a critical operation. Similar to HDFI, tag1 distinguishes between secure and non-secure memory operations from an isolation perspective, which will greatly reduce the number of memory tag comparisons at runtime. Second, tag2 is a 31-bits, unique integer number to identify the legitimate DPO and DV. tag2 indicates the meaning of rule1 and rule2 mentioned in section IV-A.

The specific assignment of tags is as follows:

- If tag1 is 0, which means that the memory operation is normal, and tag2 will be set to 0.
- On the contrary, if tag2 is 1, which means that the memory operation is a location that can be utilized by data-oriented attacks. Thus, tag2 will be initialized as an identifier for constraining data-flow stitching and gadgets dispatcher. It means that:
  - For each legitimate of DPO, the tag2 of a pointer and objects pointed by this pointer will be set to the same value.
  - For each legitimate DV, the tag2 of a variable and the last definition of this variable will be set to the same value.

For the encoding of tagged memory, we use a method proposed by Delta Pointer [22]. This encoding allows for 32-bit tags, but the address space need not be limited to 32 bits. It depends on a trade-off between the maximum object size and the address space size, both by a factor of two.

C. Dynamic Updating and Checking with tagged memory

We perform security checks on every memory operation where tag1 is 1 during runtime. The security policy is to compare whether the tag2 of the DPO and DV is legal. A mismatch in the memory tag indicates that this DPO

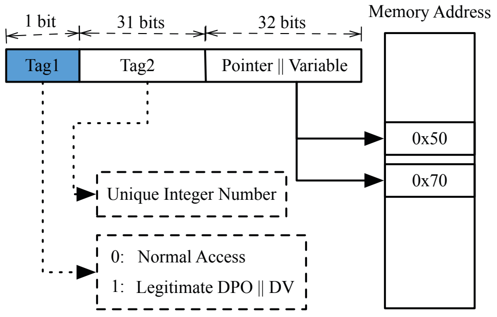

Fig. 5. The sturcture of tagged memory

and DV are corrupted by illegal memory access so that the hardware will automatically give an exception signal to trigger an interrupt. As shown in Fig. 6. In the figure on the left, operations of *b* and *d* are illegal because the tags do not match. In the figure on the right, operations of *b* and *d* change to be legal due to legitimate definition updates.

V. Implementation

This section we present some details and implementation of Dam for Linux on the lowRISC (version minion-0.4). The implementation uses the method of static analysis combined with dynamic detection based on hardware extensions. The process of implementation is divided into three steps. The first is to identify legitimate DPO and DV using static analysis techniques at compile time. The second is to perform compiler instrumentation for tagged memory allocator using the LLVM pass module. Finally, we need to modify lowRISC hardware extensions to support our tagged policy.

A. Identifying legitimate DPO and DV

Because there is no concept of the pointer in the binary file. We need to perform a source-level static analysis for programs. The compiler pass module works under LLVM 3.9 [23]. The function of the LLVM pass module is to identify legitimate data pointers and decision variables. Note that we choose loop variables as the main decision variables in our experiment because data-oriented attacks corrupt loop variables to perform gadgets dispatcher in the general case. To our knowledge, it is difficult to find necessary gadgets dispatcher without using loop variables in limited code space.

We choose to analyze data pointers and loop variables at the LLVM IR (Intermediate Representation) layer because it has already processed the source code once, which makes pointer operations easier to identify through several related APIs provided by LLVM. For loop variables, it is relatively simple to identify loop variables because of their unique representation in LLVM. Logical representations of loop variables can be identified and tracked on data-flow paths using define-use chains. For data pointers, we first obtain the pointer constraint from the LLVM IR code. Then, we employ Andersen algorithm to solve the constraint conditions based on the point-to analysis. Through the above method, we can get legitimate DPO and DV.

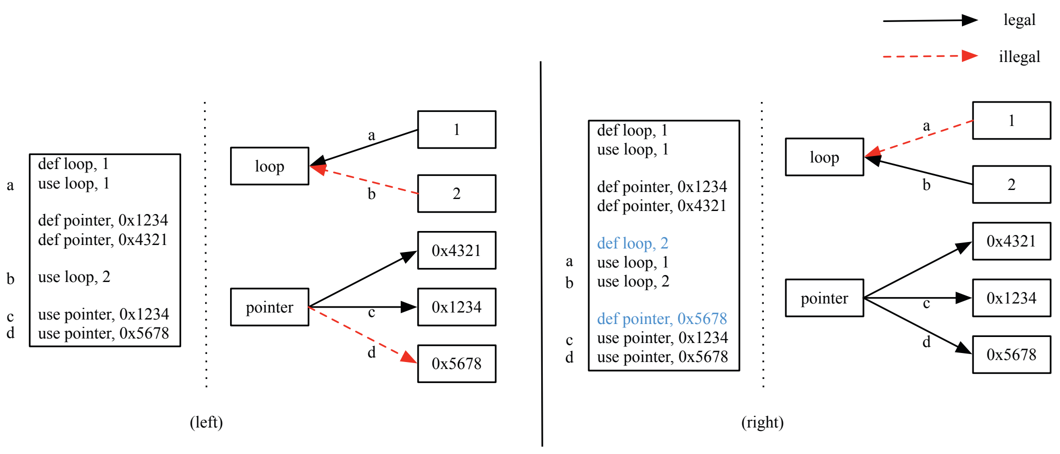

Fig. 6. Runtime def-use trace

B. Compiler Instrumentation

The goal of the compiler instrumentation is to perform security checks for DPO and DV. First, the code instrumentation needs to collection runtime operations on data-flow paths and decision sites. Second, we match the memory tags according to the policy mentioned by section IV-C. Different from static analysis, the code instrumentation is performed at the assembly level and IR level. Considering the development of an upstream RISCV-LLVM backend, we choose GCC compiler to assemble and link the assembly files compiled by the RISCV-LLVM pass module. Therefore, we use a wrapper script to handle the assembly file for compiler instrumentation. And then we employ GCC compiler to generate execute files on the 64-bit, RISC-V architecture.

To achieve the above solution, we need to add some comments into assembly files at sites of data pointers and decision variables. Then, we write a python script to replace per-inserted comments with tag updating and checking. Algorithm. 1 is the wrapper script handler. The legitimate DPO and DV are derived from section V-A.

C. Hardware modifications

The hardware architecture of our scheme is implemented on lowRISC, a baseline processor with 64-bit, 5-stage, in-order RVGC design. We use ISA extensions and tagged functions mentioned in section II-C to perform runtime checking by setting corresponding mask bits in the dedicated control and status register (CSR, tagctrl).

Moreover, we have enlarged the tag bits to support our tagged memory allocator by modifying the source code of lowRISC in Chisel. Although extended 32-bits tags increase memory overhead, our design employs the encoding of layout memory to trade-off this cost.

VI. Evaluation

In this section, we provide a series of simulation results for our scheme and evaluate the performance impact of hardware extensions. Moreover, we analyze the security of Dam from the vulnerable program mentioned in DOP [8].

Algorithm 1: Code instrumentation handler

```
Input: an assembly file with pre-inserted comments
Output: an assembly file with code instrumentation
while each sites do
    if definition of data pointer then
        object_tag1 = 1;
        object_tag2 = DPO_identifier;
    end
    if definition of loop var then
        var_tag = 1;
        var_tag2 = last_def_identifier;
    end
    if dereferencing of data pointer then
        object_tag1 ?= 1 : exception();
        object_tag2 ?= DPO_identifier : exception();
    end
    if dereferencing of loop var then
        var_tag1 ?= 1 : exception();
        var_tag2 ?= last_def_identifier : exception();
    end
end
```

A. Performance

We choose the riscv-tests [21], a unit tests for RISC-V processors, to evaluate the performance overhead of our prototype system by computing execution time of programs (the number of CPU cycles consumed on the host CPU). As described in section V. We firstly generate assembly files with comments that indicated data pointers and loop variables on the LLVM infrastructure (version 3.9). Then, we perform the code instrumentation using a wrapper script. Finally, we compile and link modified assembly files to generate executable files on the GCC compiler (version 7.2.0) without optimization.

Benchmarks contain ten different programs. We ran them on an emulator with the high-performance cycle-accurate verilator (the fastest free Verilog HDL simulator). The results are presented in Fig. 7. First, we counted the numbers of running cycles for benchmarks without any hardware extensions, called the baseline. Second, there are no additional memory operations besides tag processing and checking on lowRISC when the tag bits are set to zero. And we ran benchmarks with tag scheme provided by lowRISC, called the basic tag. Third, we ran tests with Dam and counted the number of CPU cycles, called Dam. In summary, the average runtime overhead of the basic tag is 4.54%, and the average runtime overhead of Dam is 6.48%. The number of CPU cycles consumed for benchmarks is shown in Table II.

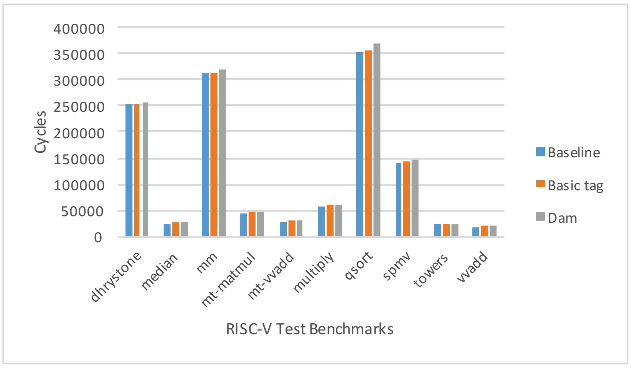

Fig. 7. Performance overhead

TABLE II
THE NUMBER OF CPU CYCLES CONSUMED FOR BENCHMARKS

Benchmarks	**Baseline**	**Basic tag**	**Dam**	**Basic overhead**	**Dam overhead**
dhrystone	251742	252936	255392	0.47%	1.45%
median	25248	26967	27351	6.81%	8.33%
mm	312178	314347	319108	0.69%	2.22%
mt-matmul	44855	45855	46712	2.23%	4.14%
mt-vvadd	27195	30829	31467	13.36%	15.71%
multiply	57982	59440	59878	2.51%	3.27%
qsort	353717	356910	369988	0.90%	4.60%
spmv	139205	144791	146764	4.01%	5.43%
towers	23667	24639	24706	4.11%	4.39%
vvadd	19012	20970	21904	10.30%	15.21%
average	-	-	-	4.54%	6.48%

B. Security

1) Data-oriented Attacks: Since there are no other real attack examples have appeared, we evaluate the security of Dam by examining an attack sample adapted from the FTP server mentioned in DOP [7]. We simplified the code of FTP Server to better describe the cause of the vulnerability. The code snippet is shown in Fig. 8, there is a memory error controlled by attackers to corrupt some local variables. The payload can be simulated by the input of attackers with dispatcher gadgets, selector and assignment gadgets.

This attack is divided into three steps. First, the adversary uses the buffer overflow to make the pointer *p* and *q* point to the address of *pms1* and *pms3* respectively with assignment gadgets. Second, dispatcher gadgets and selector make the function *systemaddr()* to be executed and return address of *system()* to the attacker. Finally, the attacker uses the assignment gadgets to configure function parameter and call the function *system("/bin/sh")*.

```
typedef struct _mystruct{void (*foo)();}
    mystruct;
mystruct *pms1, *pms2, *pms3;
void main(){
  int old_value, new_value, limit = 2;
  int *p = &old_value, *q = &new_value;
  pms1 -> foo = normal_function1;
  pms2 -> foo = normal_function2;
  pms3 -> foo = sensitive_systemaddr;
  while(limit--){          //dispatcher gadgets
    choose = limit + 1;
    read(STDIN_FILENO,&buf,512);//memory error
    if(choose == 1)              //selector
        pms1 -> foo();
    else if(choose == 2)
        pms2 -> foo();
    else
        set_string(&buf);
    *p = *q; //assignment gadgets (stitching)
  }
}
```

Fig. 8. Vulnerable code snippet with data-oriented gadgets

2) Defense by Dam: According to the mitigation scheme proposed by this paper. Legitimate DPO is *(p, &a)* and *(q, &b)* respectively. Legitimate DV is *(limit, 2_addr)*. Therefore, the target program will trigger a hardware interrupt in the case where pointer *p* points to *pms1* or variable *limit* defined by a *buffer*. Blocking dispatcher and stitching gadgets makes it difficult to find data-oriented gadgets that can be utilized. Although our scheme is not Turing-complete, it can effectively mitigate data-oriented attacks greatly. Dam mitigates all currently known data-oriented attacks.

C. Limitation

For security, we just examined the typically data-oriented attack mentioned in DOP [7]. We hold the point of view that it is safe if there are not enough gadgets can be found with a limited code size, which is similar to ROPecker [24]. From this perspective, we have indirectly proved that it is difficult to replicate the gadget search module in data-oriented attacks without data pointers and decision variables. In other words, Dam can effectively break the requirements in constructing a currently known data-oriented attack.

To further reduce runtime overheads, we need to reduce the number of tag checking. Watchdog Lite [25] indicate that their compiler prototype can eliminate 72% of temporal checks and

40% of spatial checks on average. Therefore, we can further reduce the performance overhead by removing unnecessary checks in theory. One of future works could focus on more optimizations about static analysis.

VII. Conclusion

In this paper, we propose Dam, a practical scheme to mitigate data-oriented attacks with tagged memory based on hardware. Dam is a novel approach using the idea of tagged memory to break the requirements (data-flow stitching and gadgets dispatcher) in generating data-oriented attacks. First of all, we develop two rules for constraining data-flow paths. Rule1 is that data pointers in the target program should point to legal memory locations, which we called a trusted set of data pointer objects (legitimate DPO). Rule2 is that decision variables in the target program should be defined by legal memory access, which we called legitimate decision variables (legitimate DV). Next, we obtain legitimate DPO and DV using static analysis techniques. Besides, we perform security checking based on tagged memory during runtime rely on our rules. Finally, the hardware will automatically trigger an interrupt in the case of mismatching of tagged memory. We have implemented Dam based on the open-source RISC-V Core lowRISC [19]. And our evaluation results show that our scheme has an average performance cost of 6.48%, while Dam provides source compatibility and effective defense for data-oriented attacks. However, there are some limitations need to be solved in future work, as described in section VI-C.

Acknowledgment

This work is partially supported by the National Natural Science Foundation of China (No. 61602469), and the Fundamental theory and cutting-edge technology Research Program of Institute of Information Engineering, CAS (Grant No. Y7Z0411105).

References

[1] 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, pp. 2:1–2:34, 2012.

[2] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 745–762.

[3] S. Chen, J. Xu, and E. C. Sezer, "Non-control-data attacks are realistic threats," in *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, 2005.

[4] N. Stojanovski, M. Gusev, D. Gligoroski, and S. J. Knapskog, "Bypassing data execution prevention on microsoftwindows XP SP2," in *Proceedings of the The Second International Conference on Availability, Reliability and Security, ARES 2007, The International Dependability Conference - Bridging Theory and Practice, April 10-13 2007, Vienna, Austria*, 2007, pp. 1222–1226.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, 2003.

[6] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, 2005, pp. 340–353.

[7] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 177–192.

[8] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 969–986.

[9] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, 2018, pp. 1868–1882.

[10] T. Nyman, G. Dessouky, S. Zeitouni, A. Lehikoinen, A. Paverd, N. Asokan, and A. Sadeghi, "Hardscope: Thwarting DOP with hardware-assisted run-time scope enforcement," *CoRR*, vol. abs/1705.10295, 2017.

[11] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, 2006, pp. 147–160.

[12] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, 2009, pp. 245–258.

[13] ——, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, 2010, pp. 31–40.

[14] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*, 2008, pp. 263–277.

[15] G. Dessouky, T. Abera, A. Ibrahim, and A. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, 2018, p. 106.

[16] Z. Sun, B. Feng, L. Lu, and S. Jha, "OEI: operation execution integrity for embedded devices," *CoRR*, vol. abs/1802.03462, 2018.

[17] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: hardware-assisted data-flow isolation," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 1–17.

[18] L. Cheng, H. Liljestrand, T. Nyman, Y. T. Lee, D. Yao, T. Jaeger, and N. Asokan, "Exploitation techniques and defenses for data-oriented attacks," *CoRR*, vol. abs/1902.08359, 2019.

[19] "lowrisc," https://www.lowrisc.org/.

[20] chipsalliance, "Rocket chip," https://github.com/freechipsproject/rocket-chip.

[21] RISC-V, "Risc-v tests," https://github.com/riscv/riscv-tests.

[22] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida, "Delta pointers: buffer overflow checks without the checks," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, 2018, pp. 22:1–22:14.

[23] "llvm," http://llvm.org/.

[24] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against ROP attacks," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[25] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic, "Watchdoglite: Hardware-accelerated compiler-based pointer checking," in *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*, 2014, p. 175.