

TRUVIN: Lightweight Detection of Data-Oriented Attacks Through Trusted Value Integrity

Munir Geden
Department of Computer Science
University of Oxford, UK
munir.geden@cs.ox.ac.uk

Kasper Rasmussen
Department of Computer Science
University of Oxford, UK
kasper.rasmussen@cs.ox.ac.uk

Abstract—Data-oriented attacks, where the adversary corrupts critical program data in memory, remain one of the most challenging security threats to address. Because the attacker does not touch any code or code pointers, data-oriented attacks are able to circumvent common defence strategies such as data execution prevention or control-flow protection. Data-flow integrity (DFI) techniques can address these attacks by detecting corruption of any program data. However, due to high performance penalties, these techniques are not widely adopted in practice. This paper presents TRUVIN, a lightweight scheme that mitigates data-oriented attacks by focusing on only those variables which are crucial to the integrity assurance. Instead of instrumenting every memory operation, TRUVIN selectively defends program data that originate from only trusted agents (e.g. the programmer), as they are considered critical to the runtime integrity. Our analysis is performed at compile-time and generates instrumentation only for the necessary operations. TRUVIN reduces the performance cost by a factor of 4.3 on average with 28% overhead compared to full instrumentation (121%), while retaining the security guarantees.

Index Terms—runtime integrity, data attacks, trust

1. Introduction

Despite more than three decades of effort, memory bugs are still unsolved and stay as the mother of all evils for computer security. Applications developed in unsafe low-level languages such as C or C++ inevitably host many of these bugs as their complexity and lines of code increase. Higher-level languages (e.g. Java) aim to solve the issue by assuring memory safety. However, they are not able to fully replace unsafe languages, because of their substantial performance penalties and inevitable dependencies on systems which are also developed in unsafe languages (e.g. JVM, OS). Even if there are attempts to make these languages safer, such as bounds checking, they are also subject to similar performance costs (ca. 300% overhead). Moreover, these attempts do not address all vulnerability types (e.g. format string), which prevents them from being widely adopted.

Memory bugs can form a basis for *control-* or *data-oriented* exploits. A control-oriented attack hijacks the program’s control-flow by taking over a code pointer such as a return or an indirect jump address. Control-flow integrity (CFI) [1] techniques, provided as a default feature by many compilers today, effectively mitigate those

attacks. Despite their reasonable overhead, CFI techniques do not address data-oriented (non-control data) attacks, where the adversary modifies the program data without touching any code pointers. Such attacks can enable the adversary to reach his goal in a different way by altering program variables deciding on the control flow.

Potential solutions to data attacks need to approximate memory safety by assuring the compliance of the program runtime with its static data (flow) features via software- or hardware-based schemes. However, pioneering software-based techniques that rely on program instrumentation either incur high overheads (e.g. DFI [2] with 104% overhead) or loosen the approximation accuracy to reduce the overhead with more coarse-grained checks (e.g. WIT [3] as a flow-insensitive solution). On the contrary, hardware-based solutions [4] eliminate the overhead problem by modifying CPU architectures. But such techniques are also impractical due to their substantial deployment costs.

For a lightweight and practical solution, we need a targeted approach that protects only program data critical to the runtime integrity without sacrificing fine-grained checks and asking for expensive hardware changes. However, this is a nontrivial task; because deciding on the criticality of a variable needs a semantic understanding of the program, if it is not annotated by the programmer [5]. Oversimplifying the problem as the protection of all condition variables fall short since there may be other variables used directly by sensitive functions without affecting branch decisions. Also, some condition variables might be legitimately defined by the user input, where the integrity checks would be unnecessary. Despite some targeted proposals that work for kernel-space [6] or that require programmer annotations [5], there is not yet a generic solution for user space; we can thus automatically identify critical program data without having to understand the program semantic.

Given these shortcomings, this paper introduces TRUVIN, a lightweight software-based scheme that protects against data-oriented attacks via targeted instrumentation. TRUVIN describes program variables with values originating from trusted agents (e.g. the programmer) as critical, in order to avoid redundant instrumentation of non-critical ones which are already under the control of untrusted agents as potential attackers. As TRUVIN does not ask to understand the internal program semantic, it can effectively address sophisticated data-oriented attacks with far less overhead.

Hence, this paper makes the following contributions:

- 1) We have introduced a distinction between critical and non-critical program variables/data based on the trustworthiness of their value origins.
- 2) We have designed a novel flow-sensitive trust propagation tool, analysing the program code to identify variable values in need of protection.
- 3) We propose a security quantifier, *Loop Protection Ratio (LPR)*, assessing the reduced loop attack surface as a metric of hardening Turing-complete attacks (i.e. data-oriented programming (DOP)).
- 4) We present a new scheme TRUVIN that detects data-oriented attacks via automated transformation of programs with far less overhead.

2. Background

Memory bugs can trigger different types of attacks, depending on the targeted memory region and features. Even if the integrity of the code and code pointers is assured, the attackers can still leverage program data.

2.1. Data-Oriented Attacks

Data-oriented attacks stay as the most challenging threat slipping under the radar of common mitigation techniques such as data execution prevention (DEP) and control-flow protection (CFI). Because the attacker does not touch any control data such as return addresses, these attacks are also called *non-control data* attacks, whereas Carlini et al. [7] generalise the ones producing infeasible path traces as *control-flow bending* attacks. Chen et al. [8] have first drawn attention to the potential of data attacks by demonstrating concrete examples on real-world applications. Later, Hu et al. [9] have coined the term *data-oriented programming (DOP)*. DOP promises for powerful Turing-complete attacks in case of a fitting vulnerability. The attacker can execute arbitrary payloads by corrupting only data objects while preserving the control-flow integrity. To perform a DOP attack, the adversary requires a memory bug that can compromise a loop (the dispatcher) with necessary branches and instructions (gadgets). Ispoglou et al. [10] have taken this a step forward by automating the discovery of such a vulnerability via *block-oriented programming compiler (BOPC)*.

2.2. Data-Flow Integrity

For the effective mitigation of data-oriented attacks in general, fine-grained data flow integrity (DFI) guarantees are required. Miguel et al. [2] have proposed a pioneering DFI scheme that instruments all memory operations of the program to detect data corruptions when the data is used. The proposed technique relies on compile-time *reaching definitions* analysis. This is a flow-sensitive analysis that for each memory read (e.g. variable use) determines which write instructions (e.g. variables definitions) are allowed to define the target address. At runtime, the scheme maintains a runtime definitions table (RDT). The RDT keeps the record of the most recent write instructions on any memory address. Then, for each read access, the scheme checks this table to decide whether the actual instruction that has previously defined the address value

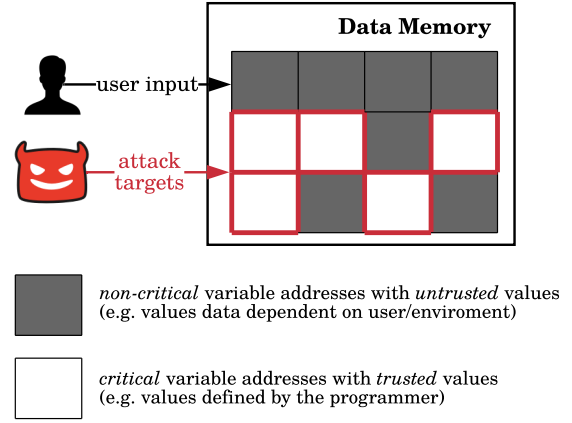


Figure 1: Separation of variables based on defining agents.

is legitimate (i.e. reaching definitions) or not. Despite some optimisations, the naive implementation instruments all memory operations regardless of their importance to the attacker with a 104% overhead reported.

3. Problem Setting

Proposed DFI scheme [2] could not be widely adopted mainly due to its high performance costs. In contrast, hardware-assisted solutions [4] ask for expensive hardware changes despite their low overheads. A targeted software-based technique can thus be practical by reducing the instrumentation overhead. However, determining the program data in need of protection is a challenging task, which TRUVIN aims to address.

3.1. Motivation

For DFI scheme [2] relying on *reaching definitions* analysis, performance overhead is mainly defined by two factors. The first major one is instrumented memory operations, most of which correspond to variable *definitions* or *uses*, and can be roughly approximated from the number of program variables. The second minor factor is the cardinality of *reaching definitions* sets that determines the search cost for each memory read. In order to minimise the cost arising from both factors, TRUVIN adopts a novel targeted approach that addresses data-oriented attacks with far less overhead.

Since the overhead is proportional to the number of memory operations, TRUVIN selectively checks the integrity of variable values which are expected to be the primary attack objectives while avoiding redundant instrumentation on others. In order to achieve this, at a broad level, our scheme classifies variables into two groups as *critical* and *non-critical* based on the trustworthiness of agents contributing to their values (see Figure 1). We formulate critical variables as addresses hosting *trusted* values that originate from reliable agents such as the programmer, for the given program point in a flow-sensitive setting. On the other hand, non-critical variables are described as addresses containing *untrusted* values that are directly or indirectly (i.e. data dependencies) defined by potentially malicious agents such as users or the environment.

Because non-critical targets can be manipulated even by legitimate users, the adversary cannot benefit much from corrupting their values. For a data-oriented attack, the adversary needs to overwrite some critical variable value in a way that the legitimate program semantic (i.e. the programmer) does not anticipate. Accordingly, untrusted agents (e.g. users) as potential attackers must not overwrite the critical values directly or through data dependencies and must not have an impact on those beyond the legitimate semantic (i.e. control dependencies only). Based on these insights into variables, in order to minimise the instrumentation overhead, TRUVIN does not check the integrity of untrusted values hosted by non-critical variables, which are expected to be only preliminary or intermediary objectives, but not the ultimate attack targets.

3.2. System Model

The system accepts the program’s intermediate representation (IR), given by the compiler front-end, as the input. Then, the first analysis part identifies its variable instructions operating with trusted values. The second part transforms the IR by selectively instrumenting the identified instructions. Lastly, modified IR is translated into the executable, which is capable of detecting data corruptions.

We consider that the system has the code integrity, DEP, fine-grained control-flow (CFI) protection on both forward and backward edges. The system secures the runtime information stored for verification (i.e. shadow memory) through randomisation or other means such as isolation provided by trusted components (e.g. OS kernel, TEE). We do not make any assumptions about how this is achieved. Although the program code can contain memory bugs, the programmer’s intentions are correct, which means the program is free of logical or semantic flaws (i.e. a legitimate user defines a variable that results in undesired consequences). We assume that the program’s IR enables us to identify its variables precisely via their corresponding instructions (i.e. address operands, alias sets). Orthogonal research problems such as inevitable limitations of static approximations or other precision issues due to imperfections of points-to analyses are not within the scope of this paper.

3.3. Adversary Model

The adversary’s goal is to modify any critical variable, i.e., a variable that holds a trusted value, typically given by the programmer at compile-time. Trusted values do not directly depend on any input that is potentially malicious, but they are instead mostly controlled by the program semantic to reflect changes in internal state. In practice, an attacker could, for example, modify such a variable value by overflowing a non-critical variable addresses in the same stack frame. But we do not make any assumptions about how corruption is achieved. The adversary has full control over the value of all non-critical variables of the program. However, he is not able to interfere with the instrumentation process, meaning that he cannot modify the instrumented binary and shadow memory allocated to host the instrumentation data. The attacker will also fail if any control data is disturbed, given that the system assumes a perfect CFI in place. The program does not provide a pure

information leakage scenario, where only confidentiality is compromised without harming the data integrity. Physical or hardware attacks are also not covered.

This adversary model captures data-oriented attacks extensively, including those corrupting only a specific variable and more sophisticated DOP attacks. It does not only help us to focus on lightweight detection of those attacks but also lets other solutions (e.g., isolation) be adaptable to our scheme.

4. Distinguishing Critical Variables Hosting Trusted Values

Our targeted approach relies on the fact that program data is control or data dependent on each other, the value origins of which have to be either trusted or untrusted, while control dependencies act as the legitimate interface to reflect trusted values from untrusted input. By distinguishing critical variables based on the value trustworthiness, TRUVIN implicitly isolates critical group from non-critical ones while placing fine-grained checks among the critical ones. In order to determine critical variables, we first perform a static analysis of the program IR. This novel analysis propagates the trust originating from reliable sources such as the programmer, and identifies variables hosting trusted values—which can be only control dependent on untrusted values—as critical.

4.1. Trust Sources and Propagation

Analysing the trustworthiness of variable values is twofold: trust sources and trust propagation. We formulate *trust sources* as origins that hold integrity guarantees and assign their values via trustworthy processes such as variable (re)definitions by the programmer. Because of the immutability of the code (where programmer values are stored) and basic assignment operation (undoubtedly free of bugs), programmer-defined values introduce trust to the system. The analysis module also allows defining other trust sources. For instance, the analyst can specify external trust sources such as dynamic functions or system calls, which are not part of the program code. Also, the assignments from specific database or configuration files—considered to have integrity properties—and functions placed for sanitisation can be stated by the analyst.

Apart from those sources, the analysis propagates trust throughout the program to discover other emerging trusted values or the ones of which the trust vanishes. *Propagation* rules given below are defined for different operation types:

- *Assignment*: If a trusted value is copied to another variable, the result is also trusted (e.g. $t_2 = t_1$).
- *Unary*: If the operand of a unary operation is trusted, the result is also trusted (e.g. $t_2 = -t_1$).
- *Binary*: If both operands of a binary operation are trusted, the result is also trusted (e.g. $t_3 = t_1 + t_2$).
- *Compare*: If at least one of the operands is trusted, the result is also trusted (e.g. $t_3 = (u_1 < t_2)$). (i.e. this rule enables the programmer to use control dependencies as the interface for value checking or sanitisation.)
- *Address*: If all the operands of a pointer arithmetic operation are trusted, the result is also trusted (e.g. $t_3 = \&v_1 + t_2$).

- *Call*: If the function is a trust source, the function output is also trusted (e.g. $t_1 = f()$). If the function is a sanitising one, specified output is trusted (e.g. $t_2 = f(u_1)$). If the function is a trust propagating one, based on the fulfilling argument of the propagation rule, the specified output is trusted (e.g. a function returning a trusted value in case the first argument is trusted: $t_3 = f(t_1, u_2)$).

4.2. Static Trust Analysis

To distinguish critical variables hosting trusted values according to the rules above, we adopt a static flow-sensitive approach that uses iterative data-flow analysis framework [11]. This framework first requires data-flow equations for the control flow graph (CFG) nodes to represent their entry (in) and exit (out) states. Then, it solves them at compile-time by repeatedly performing the abstract interpretation of program statements until the whole system converges (stabilises). Abstract interpretation is often defined by three parameters. The first one is *transfer function* that simulates the execution of instruction(s) of each CFG node n . The second parameter is the *direction* of the analysis (i.e. forward or backward). And the last one is *join operator*, which can be either union or intersection, stating how to combine the property flowing from predecessor or successor nodes.

Since TRUVIN requires fine-grained information about the memory operations to be instrumented, for our analysis, each CFG node n corresponds to a single instruction instead of using basic blocks. We formulate our transfer function as follows:

$$f(n) = gen[n] \cup (in[n] - kill[n]) \quad (1)$$

$gen[n]$ is set of trusted values introduced in node n ,
 $kill[n]$ is set of values whose trust vanished in node n ,
 $in[n]$ is the set of trusted values at the entry of node n ,

Because the trust originates from earlier statements (e.g. trust sources), it is a property flowing *forward*, which gives the direction of our analysis. We choose *intersection* as the join operator because of the exclusive definition of trust and not having an interest in protecting variables that can be controlled by the user through at least one legitimate path. As a *forward must* analysis, our data-flow equations therefore become:

$$in[n] = \bigcap_{p \in pred[n]} out[p] \quad (2)$$

$$out[n] = f(n) \quad (3)$$

$out[n]$ is the set of trusted values at the exit of node n

For the outcomes of Equations 2 and 3 to be converged, we have used function-wise worklist algorithm to solve the equations at function level. For a program-wide analysis, this algorithm is applied to each function in the reverse topological order of the call graph (i.e. bottom-up traversal). A caller function can thus use the analysis outcome of the callee for

scalable interprocedural trust propagation. Functions with arguments run this algorithm multiple times to figure out standalone propagation of each argument to the function output (assuming the argument of interest holds a trusted value). Based on discovered patterns on the function output such as the return values and arguments called by reference, propagation rules are generated during the bottom-up traversal of the call-graph. These rules describe which function arguments can propagate its trust or whether the function can act as a trust source even if none of the argument values is trusted. Created rules enable the abstract interpretation of call statements for an interprocedural analysis. Upon completion of the bottom-up traversal of the call graph, the program-wide analysis provides stable entry (*in*) and exit (*out*) states for each program instruction. These sets inform us about the trustworthiness of variable values on the given instruction/node. This information already encloses memory instructions that must operate with trusted values, which corresponds to the critical variable instructions.

Despite its interprocedural approach, the analysis is deliberately designed as context-insensitive. Because the instrumentation code of a function should not differentiate for different calling contexts, we have not considered a context-sensitive trust propagation that would not benefit the transformation phase.

5. Detection of Data-Oriented Attacks

After instructions operating with trusted values are identified as critical, the program is transformed in a way that the attacks modifying those can be detected.

5.1. Value-Based Integrity Checks

The scheme proposed by Miguel et al. [2] is not suitable for targeted instrumentation of specific variable operations. Because the scheme compares the static *reaching definitions* sets with the actual instructions that define variables at runtime, it has to record all memory writes first regardless of their legitimacy. Otherwise, if the adversary leverages an instruction not recorded on the runtime definitions table (RDT), the attack cannot be detected.

For this reason, TRUVIN adopts a different approach to make targeted instrumentation possible. Instead of monitoring every write instruction, we check the value integrity of variables by allocating a shadow cell to each critical variable identified. Shadow cells are designed to store actual variable values defined by legitimate instructions only. When a critical variable is legitimately defined, the instrumentation updates its corresponding shadow cell with the actual value written. When the same variable is used, the instrumentation checks whether the actual value read matches with the shadow value lastly recorded in the corresponding cell. If the values do not match, TRUVIN concludes for an attack, because the variable must be overwritten by an illegitimate instruction, which is not statically computed. This approach substitutes instruction identifier checks by creating implicit *def-use* instruction pairs (reaching definitions) through variable values. Only one hypothetical scenario that would stay undetected is overwriting a variable with the same existing value of a legitimate definition. However, the adversary cannot

```

1 void login(){
2   int authenticated=0; /*trusted value*/
3   int role_id; /*untrusted value*/
4   int login_attempt=0; /*trusted value*/
5   char pwd[STR_SIZE]; /*untrusted values*/
6   char user[STR_SIZE]; /*untrusted values*/
7   read(user, "Please enter username:"); /*vulnerable*/
8   if (is_user_locked(user))
9     exit(ERROR_USER_LOCKED);
10  role_id=get_role_id(user); /*trusted value*/
11  while (authenticated==0 && login_attempt<=MAX){
12    read(pwd, "Please enter password:"); /*vulnerable*/
13    if (check_credentials(user, pwd))
14      authenticated=1; /*trusted value*/
15    login_attempt++; /*trusted value*/
16  }
17  if (authenticated==0 && login_attempt>MAX){
18    lock_user(user);
19    exit(ERROR_USER_LOCKED);
20  }
21  if (authenticated){
22    if (role_id<=SYSTEM_ADMIN)
23      generate_privileged_session(user);
24    else
25      generate_unprivileged_session(user);
26  }
27  return;
28 }

```

Figure 2: Vulnerable program code.

benefit from this scenario, as he is obviously in need of a different value to perform the attack.

Thanks to the value comparisons permitting targeted instrumentation, TRUVIN can thus avoid the overhead of checking non-critical variables with untrusted values. Additionally, the value-based integrity replaces the cost of searching on instruction sets (i.e. reaching definitions) by a single value compare. The only exception is pointers that may define the same variable of interest. In such cases, separate shadow cells are allocated to the pointers. For a critical variable definition via legitimate pointer dereference, the instrumentation records the actual value written to the allocated pointer cell. Then, if there is a native use of a critical variable, the value of which might be already defined by the pointer, the instrumentation checks both the native variable and the pointer cells for a matching shadow value. The cost (i.e. the number of shadow cells checked/updated) for such a variable use is thus no more than the number of pointers that may define the variable of interest at the given program point. In contrast, searching on reaching definitions has to consider both pointer-based and native definitions reaching from different control flow paths.

Since the proposed technique relies on shadow values to detect attacks, one concern can be space requirements of composite variables such as arrays or strings. Even though benchmark experiments have never identified those structures as critical entirely, in such a case, we suggest using checksums to digest consecutive elements of composite variables (e.g. URL) similar to the previous work [12].

5.2. Scope

TRUVIN recognises an overwrite of a critical variable (trusted) value by a non-critical variable (untrusted) value instruction as an attack. Still, it deliberately ignores cases such that a non-critical instruction corrupts untrusted values that should be defined by other non-critical instructions (e.g. later modification of the user/environment

```

line 4: int login_attempt=0
store i32 0, i32* %4, align 4
.....
line 10: role_id=get_role_id(user)
%14 = call i32 @get_role_id(i8* %13)
call void @vi_def_32(i32 %14, i16 2)
store i32 %14, i32* %3, align 4
.....
line 11: while( ..login_attempt<=MAX)
%19 = load i32, i32* %4, align 4
call void @vi_use_32(i32 %19, i16 3)
%20 = icmp sle i32 %19, 5
.....
line 15: login_attempt++
%31 = load i32, i32* %4, align 4
call void @vi_use_32(i32 %31, i16 3)
%32 = add nsw i32 %31, 1
call void @vi_def_32(i32 %32, i16 3)
store i32 %32, i32* %4, align 4
.....
line 17: if(..login_attempt>MAX)
%37 = load i32, i32* %4, align 4
call void @vi_use_32(i32 %37, i16 3)
%38 = icmp sgt i32 %37, 5
br i1 %38, label %39, label %41
.....
line 22: if (role_id<=SYSTEM_ADMIN)
%45 = load i32, i32* %3, align 4
call void @vi_use_32(i32 %45, i16 2)
%46 = icmp slt i32 %45, 2
br i1 %46, label %47, label %49

```

Figure 3: IR instrumentation on the slices of *role_id* and *login_attempt* variables.

input with the attacker payload). While this design choice forms a basis to the desired performance gain, it can lead to states where some untrusted values do not satisfy path/semantic constraints after the corruption. However, the utilisation of such states for an attack is unlikely due to the following: First, for branch decisions dependent on the value/range checks of untrusted data, the control (comparison) outcome would be already transferred to a variable as a trusted value, before the corruption. Second, leveraging such state for a control-flow bending attack requires the program to contain semantically redundant controls (e.g. duplicate check within a nested loop or control structure).

Considering the lack of an established benchmark to evaluate effectiveness against data-oriented attacks, we evaluate TRUVIN’s security promises based on hardening of DOP attacks and the discussion with some real-world exploits. In contrast to the BOPC [10] automating DOP attacks, we propose *Loop Protection Ratio* (LPR) as a metric of the reduced loop attack surface. Since powerful Turing-complete DOP attacks require the adversary to compromise at least a loop structure (i.e. gadget dispatcher) for arbitrary execution, the LPR metric, as a fraction of loop headers with instrumented condition variables, aims to assess the hardening of these attacks under our scheme.

6. Implementation of the Scheme

We have implemented a proof-of-concept of the design explained in Sections 4 and 5 to evaluate its performance promises primarily. Two LLVM passes¹ are implemented to analyse and transform the programs’ intermediate representations (IR). These passes selectively inject runtime checks. Thus, the attacks targeting critical program variables can be detected.

1. <https://github.com/msgeden/truvin>

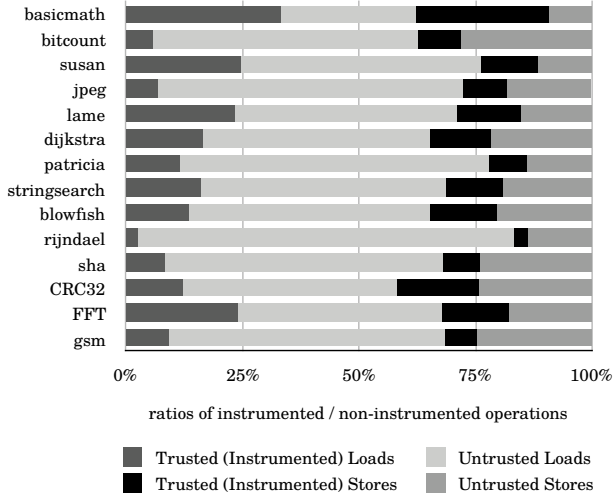


Figure 4: Ratios of instrumented memory instructions.

6.1. A Concrete Example

To illustrate how the passes work, we use a vulnerable C program which represents a typical login system, as seen in Figure 2. This program contains five variables out of which two `pwd` and `user` arrays host the login credentials. Other `authenticated`, `role_id` and `login_attempt` are used by control and loop structures. The program loads untrusted user credentials to the memory through a vulnerable `read` function. As the attacker can exploit this function to bypass the credential check (i.e. `authenticated`), he can also reset `login_attempt` to perform brute-force attacks for password discovery. Besides, he can modify `role_id` to perform a privilege escalation attack.

6.2. LLVM Passes

LLVM uses a language- and target-independent intermediate representation (IR), which is a high-level strongly-typed assembly language with RISC-like instructions, many of which are in three-address code. It uses partial static single assignment (SSA) with an infinite virtual register set and assumes two kinds of variables which are *top-level* and *address-taken* variables.

Trust Propagation. Our first LLVM pass analyses how the trust propagates throughout the program, as explained in Section 4. Although it allows the analyst to define additional trust sources (e.g. database reads), we have used programmer-defined values as the main trust source during benchmark experiments. This pass propagates trust through all top-level and address-taken variables. However, instrumentation targets only address-taken variable operations. Figure 3 illustrates the trust propagation on the IR slice of `role_id` and `login_attempt` variables in Figure 2. Operands highlighted in different colours represent *pre-* and *post-*instruction states of trust. Differently from other variables, `role_id` starts hosting a trusted value upon return of `get_role_id(user)` function. Since this function returns only programmer-defined constants (due to the limited number of roles), it is discovered

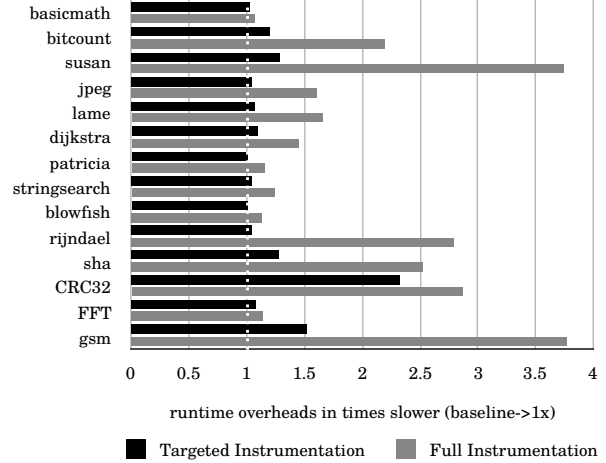


Figure 5: Runtime overheads.

as a trust source by the analysis pass as it should. This pass provides the information of trusted values available on the entry and the exit of each IR instruction within a field- and flow-sensitive setting. It thus determines which *def-use* pairs (*store-load*) operate with trusted values and must be instrumented by the following transform pass.

Value Integrity Checks. The second pass transforms the program IR by injecting function calls to check the value integrity of identified variables. As explained in Section 5.1, these calls either store shadow values for variable definitions or load existing shadow values to check whether they match with the actual ones during variable uses. Figure 3 also demonstrates *injections* placed for the load/store operations of `role_id` and `login_attempt`. These functions wrap necessary instrumentation which are inlined during compilation. Both function types accept two kinds of parameters. The first argument holds the exact copy/digest of the variables (e.g. value operand of *store* and the result operand of *load*). The second one is the variable identifier(s) that help to locate the corresponding shadow cell(s).

7. Evaluation

This section first evaluates the performance gains regarding runtime and space overheads. Next, it analyses the security guarantees provided.

7.1. Performance

For performance evaluation, we have used MiBench [13] as the most popular open-source uniprocessor benchmark suite. To contrast the performance of targeted instrumentation with a naive approach, we have compared TRUVIN against fully instrumented benchmark programs. Full instrumentation placing injections for all memory operations is considered as a performance approximation of a naive DFI scheme without making any security promises.

Regarding the instrumentation avoided, Figure 4 presents the ratios of memory instructions operating with trusted and untrusted values. Although the metrics vary

Program	Trusted	Untrusted	LPR
basicmath	17	1	94.4%
bitcount	6	0	100%
susan	47	0	100%
jpeg	398	61	86.7%
lame	348	18	95.1%
dijkstra	7	0	100%
patricia	9	1	90%
stringsearch	19	2	90.4%
blowfish	36	0	100%
rijndael	10	0	100%
sha	10	0	100%
CRC32	3	0	100%
FFT	12	0	100%
gsm	62	2	96.8%
TOTAL	984	85	92.1%

TABLE 1: Loop headers with instrumented (trusted) and non-instrumented condition variable operations. *LPR*: Reduced loop attack surface for DOP attack elimination.

depending on the program and the operation, only 22% of the memory operations require instrumentation for the whole benchmark. This corresponds to 12849 out of 56694 (*load*: 7508/40864 and *store*: 5950/15830), which implies that only a small portion requires instrumentation.

Although performance gain arises from operations left non-instrumented, the number of executions of instrumented ones determine the actual overheads. As for the benchmark suite, TRUVIN has produced only 28% runtime overhead. In contrast, the full instrumentation has incurred 121% overhead, which is close to the naive DFI scheme [2] reporting 104% overhead for a different set of benchmark programs. Figure 5 presents the runtime overheads of each transformed benchmark program to compare TRUVIN against fully instrumented programs. Performance gains vary from 1.4x to 44x depending on the benchmark program. Regardless of the variance, the selective instrumentation provides 4.3x performance gain for the whole suite. Considering space overheads, shadow memory that duplicates only trusted values requires 8x less memory than the allocation made for full instrumentation. Moreover, selectively instrumented binary sizes with inlined calls are only 40%, whereas fully instrumented binaries are 134% greater than non-instrumented binaries.

7.2. Security Analysis

For a successful attack, the adversary must overwrite a trusted value of a critical variable by exploiting an illegitimate instruction that belongs to either a non-critical variable or another critical variable. The first scenario, an illegitimate data-flow from untrusted agents to the trusted domain, will be caught due to outdated shadow cell of the variable of interest. The second scenario, an illegitimate data-flow among the critical addresses, will also be revealed due to unmatched shadow values since the instrumentation of the exploited instruction would update only its corresponding shadow cell.

DOP Attacks. Table 1 demonstrates the number of loop headers (i.e. program statements that decide for the loop iteration based on condition variable checks) of which condition variable values are trusted. The ratio of variables with trusted values to all variables corresponds to the LPR metric explained in Section 5.2. Because the

variable instructions operating with trusted values are instrumented, DOP attacks in need of a gadget dispatcher utilising these loops/variables will be caught. For the whole suite, TRUVIN promises to reduce the loop attack surface by 92.1% as it defends all loop headers for 8 out of 14 benchmark programs (100% LPR). As the remaining loop variables could genuinely be part of/dependent on user inputs, some of them might be defined via external functions that need to be stated as the trust source by the analyst. However, in either case, it provides strong evidence that TRUVIN successfully identifies variables critical to the program flow and catches attacks targeting them without any programmer intervention.

Real-world Scenarios. Although there is not an established benchmark to measure the effectiveness of TRUVIN, we discuss it considering some real-world exploits introduced in related papers [8], [9], [14], [15]. The first one is a vulnerability (CVE-2001-0144) found in many *SSH* implementations, which overwrites *authenticated* flag to bypass the authentication process. Similarly, a *Chrome* bug (CVE-2014-1705) in the renderer process can be exploited to alter critical *m_universalAccess* flag to bypass the SOP (same-origin policy) enforcement. TRUVIN can mitigate both scenarios that overwrite decision-making variables since they are all defined by the programmer. Two examples, *wu-ftpd* and *sudo* suffer from format string vulnerabilities (CVE-2000-0573, CVE-2012-0808). Those bugs can result in privilege escalation by altering variables that host the user ID of calling processes. Both cases will be caught by TRUVIN when *getuid()* system call is defined as an external trust source. Apart from these, *Null httpd* (CVE-2002-1496) bug makes adjusting the CGI-BIN path possible for remote execution. Another vulnerability (CVE-2011-4862) in *telnet* daemons can lead to corruption of a configuration string *loginprg* that specifies the executable path responsible for user authentication. TRUVIN’s design allows detecting corruption of those strings, as long as they could be represented as digests on the shadow memory.

Our scheme not only eliminates Turing-complete capabilities of data-oriented attacks (LPR), but also it can mitigate well-known real-world exploits as specific attack executions.

8. Related Work

Regarding targeted solutions to data-oriented attacks, KENALI [6] protects the Linux kernel against particular privilege escalation attacks. It selectively defends branch variables used by functions returning specific access error codes. KENALI places write integrity tests (WIT) [3] to prevent illegitimate overwrite of the identified data. Compared to the DFI [2], WIT [3] is a more coarse-grained technique that instruments all memory writes to restrict pointer dereferences, so that they can define only objects in its own points-to sets given by a flow-insensitive analysis. As a user-space solution, Datashield [5] allows the programmer to annotate custom data types (struct) as critical to selectively isolate the data held by these types. Apart from the impracticality of the annotations, Datashield associates the data sensitivity (ideally the property of the value) with the custom data types. This type-based

approach overlooks critical data stored on generic types (e.g. integer). A recent paper [16], published while our work was under submission, proposes checking mainly condition variables to extend the integrity attestation of embedded system operations. However, this study does not exclude condition variables that might be legitimately defined/dependent by/on the user. Furthermore, it requires programmer annotations for variables that are not part of branch decisions and used by sensitive functions. Contrarily, TRUVIN decides on data criticality (i.e., the need for integrity assurance) based on the trustworthiness of their value agents. As our approach spots the underlying cause of being critical, it also allows for a more fine-grained and automated selection of critical data.

As a contrasting approach, taint-tracking studies can be considered as the negation of our trust analysis in some ways. Differently, those studies inspect how user/suspicious input propagates throughout the program and whether it reaches any sensitive function or address. Popular dynamic techniques that are used to detect exploits at runtime either heavily instrument the program [17], [18] with huge runtime overheads (6x - 30x slowdown) or rely on some expensive hardware-extensions [19]. Although static taint analyses do not incur overheads, they are mostly used to reveal semantic flaws (i.e. a path from a taint source to a specified sink). This is mainly due to the difficulty of detecting corruptible addresses beyond foreseeable program semantic that is subject to approximation limitations. TRUVIN, as a hybrid technique benefiting from the strengths of both static and dynamic approach, first identifies trusted (untaintable) variable values in need of protection at compile time. Then, it selectively instruments their corresponding legitimate operations to detect integrity issues on them, which are not foreseeable by the program semantic/analysis.

9. Conclusion

This paper presents TRUVIN, a novel software-based lightweight scheme that detects data-oriented attacks without having to instrument every memory operation. TRUVIN avoids unnecessary checks on preliminary or intermediary targets/stages of an attack hosting untrusted values already. This means that we only need to instrument a fraction of the variables, thereby saving both CPU time and memory, while retaining the security guarantees.

For this to work, we first distinguish critical variables based on the trustworthiness of agents contributing to their values, (e.g. the programmer as a trusted or user as an untrusted agent), by deploying a flow-sensitive analysis that propagates the trust throughout the program flow at compile-time. Based on the information extracted, our scheme transforms the (potentially) vulnerable program by selectively injecting runtime checks on instructions that must operate with trusted values. This ensures that if a trusted value of a critical variable is corrupted by the attacker, it will be detected, but without overhead of checking every address. Experiments show that our targeted approach with 28% overhead yields 4.3x gain on average compared to the full instrumentation (121%).

TRUVIN is a lightweight solution that achieves generalisable and targeted defence against data-oriented

attacks without asking for a deep understanding, or time-consuming annotation, of the program code.

References

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005, pp. 340–353.
- [2] C. Miguel, M. Costa, and T. Harris, "Securing Software by Enforcing Data-flow Integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160.
- [3] P. Akrividis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, 2008, pp. 263–277.
- [4] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-Flow Isolation," in *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*, 2016, pp. 1–17.
- [5] S. A. Carr and M. Payer, "DataShield: Configurable data confidentiality and integrity," *ASIA CCS 2017 - Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security*, pp. 193–204, 2017.
- [6] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, "Enforcing Kernel Security Invariants with Data Flow Integrity," in *Proceedings 2016 Network and Distributed System Security Symposium*, no. February, 2016, pp. 21–24.
- [7] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *USENIX Security Symposium*, 2015, pp. 161–176.
- [8] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security Symposium*, vol. 5, 2005.
- [9] 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 *Proceedings - 2016 IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 969–986.
- [10] 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*. ACM, 2018, pp. 1868–1882.
- [11] G. A. Kildall, "A unified approach to global program optimization," in *1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1973, pp. 194–206.
- [12] M. Geden and K. Rasmussen, "Hardware-assisted Remote Runtime Attestation for Critical Embedded Systems," in *17th Annual Conference on Privacy, Security and Trust (PST)*. IEEE, 2019.
- [13] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," *2001 IEEE International Workshop on Workload Characterization, WWC 2001*, pp. 3–14, 2001.
- [14] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic Generation of Data-Oriented Exploits," in *USENIX Security Symposium*, 2015, pp. 177–192.
- [15] Y. Jia, Z. L. Chua, H. Hu, S. Chen, P. Saxena, and Z. Liang, "The 'Web/local' boundary is fuzzy: A security study of Chrome's process-based Sandboxing," *ACM*, 2016, pp. 791–804.
- [16] Z. Sun, B. Feng, L. Lu, and S. Jha, "OAT: Attesting operation integrity of embedded devices," in *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1433–1449.
- [17] J. Newsome and D. Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software," in *NDSS 05 Networks and Distributed Systems Security*, 2005.
- [18] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "TaintTrace: Efficient flow tracing with dynamic binary rewriting," in *Computers and Communications, 2006. ISCC'06. Proceedings. 11th IEEE Symposium on*, 2006, pp. 749–754.
- [19] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ACM Sigplan Notices*, vol. 39, no. 11, 2004, pp. 85–96.