iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

METHODS OF TAINT TRACKING IN COMPUTER ARCHITECTURE AND THEIR ANALYSIS

Victor Aworetan
University of Central Florida
Victor.Aworetan@Knights.ucf.edu

Jonathan Ponader
University of Central Florida
JPonader@Knights.ucf.edu

ABSTRACT
Vulnerabilities in software have a major ef ect on the security of the process. To mitigate many of these vulnerabilities a series of works focusing on the topic of taint tracking have been proposed. These works focus on the tracking of untrusted data as they interact with the applications. Then prevent the data from improperly influencing the data flow or program flow.
This paper discusses several of these approaches to taint tracking and looks at how they propose to manage the tracking of taint

1. INTRODUCTION

The tracking of taint, or information from untrusted sources, through an application, has become an important part of improving application security. As input from untrusted sources can lead to high-level attacks such as SQL injection, command injection and directory traversal. As well as low-level attacks such as buffer overflows, format string and pointer overwrite. These attacks attempt to change the execution of a program and/or leak sensitive information.
Taint tracking uses the idea of tracking the effects of untrusted inputs through the application's execution. This is done by propagating the "taint" as instructions interact with tainted data. The propagation of this taint can be done in several ways, through hardware, software, or instrumentation; or the combination of them. Hardware approaches usually consist of adding architectural features such as registers or extending architectural tables to hold the taint data. Software approaches usually consist of running the program inside of a wrapper that is used to track the taint. Instrumentations use the binary rewriting adding instructions to the application execution. This binary rewriting can either be done in the compiler or during execution to track the taint.
The way taint data is handled and propagated during the application's execution is the motivation for the different approaches employed in taint tracking. Each approach uses a unique way to store the taint metadata and propagate the taint through the application's execution.
The policy for propagating the taint can vary between applications as not all applications handle data in the same way. Usually, the policy for taint tracking is based on the handling of four situations. Copy, the process of moving tainted data from one location to another. Arithmetic, the use of tainted data in mathematical or logical operations. Address Calculations, the use of tainted in data as the address or offset in address calculations. Control, the use of tainted data in control statements, like if statements.
The rest of the paper is organized as follows: Section 2 discusses the five approaches analyzed in this paper, and gives a critique of the approaches. Section 3 gives an in-depth comparison of the approaches. Section 4 gives a conclusion to the current state of taint tracking.

2. DIFFERENT APPROACHES

Different approaches to taint tracking all use different methodologies to track and policies propagate the taint through the application. Some proposed approaches that follow for tracking the taint propose a methodology for tracking the taint but not the policy for propagating the taint. But do show a theoretical policy that works with the methodology and prevent the shown attacks. This is done as depending on an application the way the taint should propagate varies greatly, and acceptable behavior in one application might be an opening for an attack in another

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

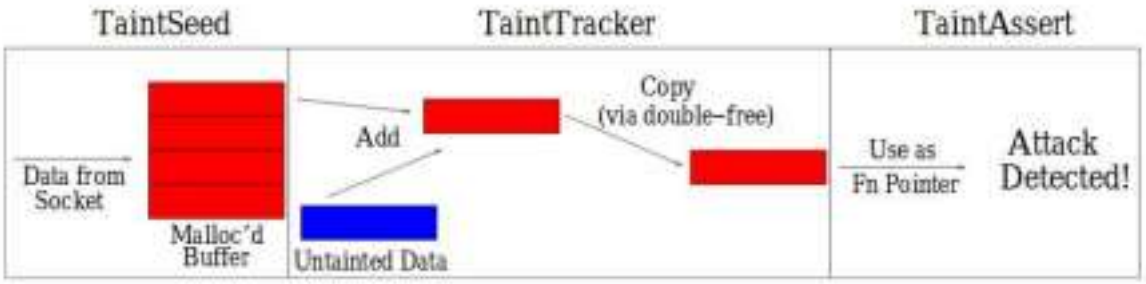

Figure 1: TaintCheck Detection of attack

2.1 Dynamic Taint Analysis

This approach is one of the basic forms of taint tracking as it uses instrumentation. Data that originate from untrusted sources, such as network sockets, are labeled as tainted - data arithmetically derived from these sources are also labeled tainted. The propagation of this tainted data is tracked during the program's execution and an attack is detected if the data is used in ways considered illegal. This approach allows the detection of low-level vulnerability attacks such as return address overwrite, buffer overflow and format string.

TaintCheck operates on a normally compiled binary program and as such can be used for a variety of programs.

Threat Model: The attacker can cause a value, such as a jump address, that is usually derived from a trusted source to be derived from his own input.

Taint Analysis is done by running the program in an emulation environment, Valgrind; whenever control reaches a new block, Valgrind first translates the block of x86 instructions into its own UCode, which is then passed to TaintCheck for the injection of its analysis code. TaintCheck then passes the rewritten code to Valgrind for translation back to the x86 instructions for execution.

TaintCheck uses four components;

TaintSeed - this marks any input from an untrusted source as tainted.

TaintTracker - this monitors each instruction that manipulates data to see whether the result is tainted or not

TaintAssert - this checks whether tainted data is used in ways defined as illegal by the active policy

Exploit Analyzer - provides information about how the attack occurred and what the exploit attempts to do

Each byte of memory, such as registers, heaps, etc has a four-byte shadow memory that stores a pointer to a Taint data structure if that location is tainted or a NULL pointer if it is not. This shadow memory is maintained using a page-table-like structure. Upon a memory being tainted, TaintSeed allocates a Taint data structure that records the system call that caused the taint, a snapshot of the current stack and a copy of the data written; the shadow memory is then set to a pointer to this structure - this is how TaintCheck maintains information about taint data propagation.

The TaintTracker adds instrumentation before each data movement - such as LOAD, STORE, or arithmetic operations - such as ADD, SUB. Upon a tainted result, TaintTracker sets the shadow memory of the result to point to the Taint structure of the tainted operand.

How each of these components work can be specified in a policy - a set of rules that determine which data should be marked as tainted and which operations are illegitimate.

TaintCheck is designed primarily to detect overwrite attacks such as buffer overflow, string format attack and function pointers attack - which it did successfully; although the occurrence of false negatives is dependent on the active policy.

In terms of performance, TaintCheck's implementation in the paper is not commendable as it needs lots of optimizations.

It is worthy of mention that with TaintCheck, automatic signature generation is possible - by observing parts/patterns of several attack payloads that are constant - and this allows for faster attack detection

Critique -

The instrumentation overheads caused by the use of Valgrind has a very severe impact on the demonstrated implementation of TaintCheck. Also, though using a page-table-like structure for the shadow memory mapping maximizes application memory usage (as it grows on demand), it leads to a high tracing overhead as it takes more instructions to allocate a memory mapping to an operand. Furthermore, there's no mention of how the shadow memory used for tracing is protected; it would have been better to see how this shadow

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

memory is truly secured. This work suffers significant setbacks where application performance is critical.

2.2 TaintTrace

TaintTrace is an instrumentation based approach for taint tracking, using dynamic instruction insertion. This allows it to work on all languages, as it works on the binary level, as well as work on a real used system as the overhead is significantly lower than previous taint tracking implementations.
TaintTrace can also work on a series of security policies. As the user is able to configure the security policies that should be tracked for the application. This includes marking which data origins should be considered untrusted and in which situations should the taint be propagated. In the analysis of this approach, the taint will be propagated in case of copy and arithmetic.
Threat Model: Vulnerabilities in the Red Hat Operating system, buffer overflow and overwrite attacks
The TaintTrace system consists of four components, a configuration file containing the taint propagation policies, shadow memory that tracks the taint, the program monitor that does the instrumentation, and a custom loader that loads the application binary and starts the monitor. These components work together to track the taint. The loader starts the process by loading all of the other components of the TaintTrace into memory. The monitor then reads the policy and initializes the shadow memory. From there the execution of the application binary begins, with the program monitor inserting instructions to track the taint as needed.
The shadow memory is used to keep track of the taint metadata, or which data has been tainted. This is done in a byte to byte manner, meaning that each byte of memory and general-purpose register, has a byte in shadow memory to keep track of its taint. Which is updated when an instruction updates their corresponding bytes in memory. This one to one mapping reduces the complexity of determining mapping locations, that caused overhead in previous solutions. This, however, causes a major memory overhead as for every byte of memory one needs a byte of shadow memory, effectively cutting one's memory in half.
On testing the effectiveness of the methodology with the above policy, TainTrace was able to detect and defend against format string attacks, buffer overflow, and critical variable attack. Preventing any critical information from being leaked or the control flow being hijacked.
Performance-wise Taintrace performed exceptionally better than previous implementations such as MemCheck. Which had an average case slowdown of 29.62x while taint trace had an average slowdown of 5.53x. Leading to a massive improvement in performance.
One major critique we have for this paper is the use of byte to byte mapping of the shadow memory, this overhead that leads to a practical halving of one's memory seems impractical for many applications. This is something that we believe should be greatly improved as the of the byte being mapped in shadow memory only 1 bit is actually being used. Our other concern with this work is the security of the shadow memory. They do address this security, by saying that if you attempt to access
shadow memory for a program under the monitor, the monitor would add a base offset to push one out of bounds. Which seems like weak security as it does not address what would happen if you are outside the monitor.

2.3 MemTracker

MemTracker is a hardware and software approach to taint tracking, as well as a way to find memory management errors. These errors can include pointer arithmetic errors, use of dangling pointers, reads from uninitialized memory, out of bounds memory access and memory leaks. To do this MemTracker adds hardware support for maintaining taint data and policy-based state updates. While allowing the software to continue to make taint tracking generic to be applied to all programs and handle edge cases.
To do this a programmable state transition table (PSTT) is added to hardware, with an array being stored in the applications virtual address space to keep track of the states. The PSTT allows custom taint tracking policies to be programmed in and handled by hardware to update the states. The array that holds the state data, is stored in the main memory. This allows one to take advantage of the current memory mechanism. To keep it secure, the array is separately mapped and can be protected from tampering using mprotect. The array consists of any power of 2 bits, that keep track of the state of the memory locations state.
Memtracker also extends the ISA to add event instructions. These event instructions act as flags to tell the hardware to check states of variables and update as needed.

When an event happens, whether it is an event instruction or other instructions that are events. The memory state is looked up, then used along with the instruction type to look up the proper action in the PSTT. For example, given the example PSTT in figure 2. If one was executing a Load instruction on a word with

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

the state Unallocated. A state lookup would be triggered. There they will load the word's state, and the instruction type and index the table to find the appropriate action. In this case, the state would be left the same, but an Exception would be triggered.

Event / State	UEVT0 (Alloc)	UEVT1 (Free)	LD	ST	Sub-word LD	Sub-word ST
0 (NonHeap)	0 E	0 E	0	0	0	0
1 (Unalloc)	2	1 E	1 E	1 E	1 E	1 E
2 (Uninit)	2 E	1	2 E	3	2 E	2 or 3
3 (Init)	3 E	1	3	3	3	3

Figure 2: Example PSTT, Given a Load Instruction with a State Uninitialized an Exception would be triggered

To add MeTracker into the execution pipeline two additional pipeline stages are added prior to the commit stages. The first of these stages is the pre-commit stage (PCMT). In this stage, the ROB is checked to let the instructions to exit this stage in in-order. This stage also fetches the MemTracker state. The next stage is the check stage (CHK), in this stage the event type and state are used to look up the corresponding entry in the PSTT. In this stage, the state is updated according to the PSTT. If the PSTT says throw an exception is thrown and handled like any other exception. If no exceptions the instruction moves on to the commit stage.
The caching of the MemTracker state has a major impact on the performance of MemTracker. The paper looked at 3 different approaches to the caching of the state data, split, shared, and interleaved caches. Upon evaluation, the split cache has the best worst-case performance and the best average-case performance. Therefore this caching method will be used in all evaluations of MemTracker.
As MemTracker is a method for tracking taint but not a policy in itself, the evaluation of memetracker is dependent on the policies used with memtracker. For testing, they used 4 policies. The first policy HeapData, checked for the use of unallocated and uninitialized data using 2 bits for state. The second poly HeapChunks, checks for buffer overflows from sequential access, using 1 bit for state. The third policy RetAddr detects when a return address is modified, using 2 bits for state. The final policy combined all of these policies using 4 bits as the state.
In their evaluation, they noticed that the overhead of Memetracker is dependent on the size of one's state. As the 1-bit state policy only had an overhead of 1.4%. While both 2-bit states policy had
an overhead of 1.9%. When all policies were combined into a 4-bit policy the overhead was only 2.7%. This overhead is low enough to allow memtracker to be used in real-world applications.
When the policies were tested to see if they could catch the errors they were designed to catch all errors. This shows that MemTracker is able to perform as expected. The policies are not the focus, but the functionality of the method of propagating the taint.
One major critique of the paper, is it focuses on the invalid access of memory. Not the cause of the invalid access of memory. Unlike other papers in this topic area, which focus on the tracking of taint from untrusted sources. MemTracker purely focuses on the invalid access they cause. Not how they got the invalid access.
Other issues is they only look at the overhead for on size PSTT, as they say the one experiment with is already larger than they need for the policies in which they experiment with. However in the situation that more states are needed, and a larger PSTT is needed how is the overhead affected. Is there a point where a PSTT of a certain size no longer works efficiently and some sort of PSTT caching is needed for efficient access time.

2.4 Raksha

Raksha was motivated by the uprising of high-level vulnerabilities, such as SQL injection and directory traversal, which previous works could not detect as those works were majorly focused on low-level vulnerabilities like buffer overflow attack and format string attacks. Also, the need to combine hardware speed and software flexibility was part of the motivation for Raksha.
Raksha is an architecture for software security based on dynamic information flow tracking (DIFT). The implementation of Raksha is deduced after the following careful observations; implementing DIFT in software gives room for much flexibility, but then, the need for binary instrumentation results in an undesirable amount of overheads. A hardware approach would maximize performance, however, the policy for security checks would be hardcoded to target memory corruption, leaving no room for flexibility in case we want to also check for vulnerabilities like SQL injection. Therefore, Raksha is implemented with a blend of software and hardware;

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

integrating the best of both worlds.
DIFT tags untrusted data as tainted and its propagation is tracked all through the application execution. DIFT associates a tag with every memory word, the result produced by a tainted data is also tagged and tainted data is used in an illegal way, an exception is raised.
Raksha uses hardware for transparent, fine-grain management of security tags for user code, OS code, and data across multiple processes. Software is used to provide the flexibility needed to detect a wide range of attacks. The security policies are software programmable and multiple policies - that either protect against different attacks or work mutually in protecting against and attack - can be run concurrently during execution.
Raksha supports multiple policies by associating each word with a 4-bit tag, where each bit supports an independent security policy with separate rules for propagation and checks. To avoid the overhead of an OS trap in case of a security exception, Raksha handles exceptions at the user level. The processor has an additional trusted mode, just like the kernel. A hardware register provides the address for a predefined security handler that is invoked upon a tag exception; when a tag exception is raised, the processor switches to the trusted mode within the user address space, instead of an OS trap. Raksha effectively protects the security handler's code and data using one of the four active security policies; if any part of the code or data is accessed outside the trusted mode, an exception is raised. Also, the predefined address for the exception handler can only be updated while in a trusted mode.
Hardware performs tag check and propagation for instructions executed outside the trusted mode. The rules for tag propagation and checks are specified by a set of tag propagation registers (TPR) and tag check registers (TCR); these rules are specified at the granularity of primitive operations classes; floating-point, move, integer arithmetic, comparisons and logical - instead of ISA instructions. This leads to tracking with high precision.
Raksha handles corner cases such as register resetting with XOR by allowing up to TPRs and TCRs to specify up to 4 rules for custom operations;
these then take precedence over the generic specified rule for its primitive operation.
Performance-wise, Raksha has a very good performance and was able to successfully protect against memory attacks such as buffer overflow and format string attack, as well as SQL injection, command injection, cross-site and directory traversal attacks.
Critique -
Raksha introduced a novel method for security exception handling which significantly reduces the exception handling overhead. However, making this the only feature on which the performance evaluation was based is not good enough; questions like "how does the architectural change (extra bits added to memory) affect performance?" is left unattended to.
The paper does not mention how to handle cases where just a byte of memory is accessed, since the tags are tracked only at word granularity rather than byte granularity.

2.5 Iodine

Iodine does taint tracing through static analysis instead of dynamic analysis as in TaintCheck and TaintTrace. However, it is a form of Optimistic Hybrid Analysis (OHA) and determines which operations are worth tracing and which can be executed without tracing. During recovery, Iodine eliminates the need for a rollback and instead uses a forward recovery approach.
It is observed that the rollback recovery is caused because of the dependence that exists between the monitor being elided and any potential future invariant violation that might affect the correctness of that elision; a roll-back free recovery system breaks this dependence. That is any monitor elided during program execution before an invariant failure has to be proven to be unnecessary to ensure the correctness of the dynamic analysis for the entire execution. Such elisions are referred to as safe elisions and Iodine ensures this by constraining its predicated static analysis such that it only elides a monitor if it is a safe elision; with this, upon an invariant failure, it is safe to switch to a conservatively optimized analysis and continue forward with the execution.
Iodine improves on OHA by targeting only expected dynamic executions to improve the precision and scalability of static analysis

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

Figure 3: Workflow of optimistic hybrid taint analysis

As seen above, a profiler observes representative executions to gather a set of likely invariants; these invariants are common case dynamic execution behaviors such as likely unreachable code, like unrealized called contexts. This is what the static predicated analysis is based on and finally, the program is instrumented (using DFSan) with the remaining DIFT monitors, and the checks are set as well. Upon an invariant failure, execution is transferred to s slow-path conservatively optimized analysis and continues forward.

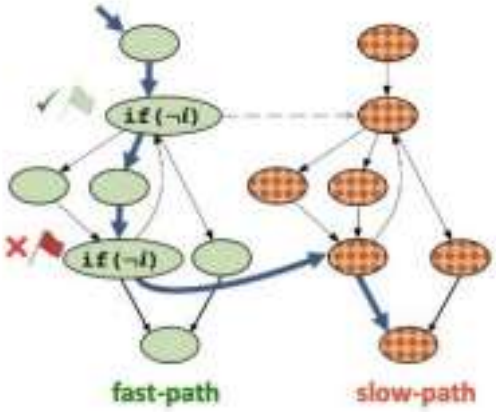

Figure 4: Forward recovery switching mechanism; execution switches from fast-path to slow-path upon invariant violation

Critique -
Undoubtedly, the ideas presented in Iodine are novel, but for truly optimized performance, a very rigorous profiling is needed to ensure the soundness of the invariants used for execution. Thus, in very robust systems, it's almost impossible to achieve fail-proof invariants. Thus, the execution encounters
more uncaptured behaviors, making Iodine execute more in the slow path.Also, the efficiency of Iodine is only laudable if, after profiling, only a small fraction of the program data needs to be monitored; in cases where almost all the profiled program's data need be tracked, then the efficiency of Iodine would be as the traditional conservatively optimized hybrid analysis.
Finally, its implementation only supports programs that are written in C.

3. Comparison of Approaches

The proposals we analyzed span a large time frame of furthering the progression of taint tracking techniques. Each technique has built on the previous methods in attempts to build a better method for tracking taint. The first method we looked at, Dynamic Taint Analysis, was one of the earliest papers looking at taint tracking using instrumentation. This method was slow and had some major memory overheads. They did mention some improvements that could be made in the paper, as this was an early prototype.

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

The next proposal, TaintTrace, built on Dynamic Taint Analysis by reducing overheads for both memory and runtime. This speedup drastically improved the usability of the taint tracking while still ensuring good security. This still had a large overhead for memory but less than previous implementations.
MemTracker was one of the first hardware software-based taint tracking mechanisms. This mechanism reduces the memory requirement of instrumentation but requires the addition of hardware to handle the logic of propagating the taint.
Raksha also came out around the time as another method for taint tracking. This method also used a hardware-software combination for propagating the taint. In Raksha the propagation of the taint is handled purely by hardware, with the opportunity for software to intervene for edge cases.
Iodine was the most recent proposal, being published in 2019. In this proposal, they look at an instrumentation approach for taint tracking. Reusing the byte to byte mapping previously used. In this implementation, they, however, pay close attention to the semantics of the code to remove unneeded instructions. The removal of these instructions is able to drastically improve the efficiency of the taint tracking.
All of these solutions have been improving the ability of taint tracking allowing it to be more and more usable in real-world applications. The combination of features from each of these works, in theory, should be able to come together and build an even better taint tracking mechanism.

Table 1: Comparison of Taint Tracking Proposal

Paper	Methodology Proposed	Policy Proposed	Taint Metadata Mapping	Limitations
Dynamic Taint Analysis	Instrumentation	User-defined	Byte to 4 Bytes	Overheads due to the way taint data is propagated
TaintTrace	Instrumentation	User-defined	Byte to Byte	Requires half of Memory to be Reserved
MemTracker	Hardware and Software	User-defined	Word to Bits	Sub-word accesses can invariably lead to false positives or negatives
Raksha	Hardware and Software	User-defined	Word to 4 Bits	Imprecision as tags are tracked at word granularity instead of bytes
Iodine	Instrumentation	User-defined	Byte to Byte	Language dependent

4. CONCLUSION

In this paper, we looked at a series of taint tracking proposals. All proposing a new methodology for tracking taint while it propagates through an application. This tracking of taint allows for the securing of modern applications from data leaks and loss of control, caused by input from untrusted sources. This can help secure many applications that have buffer overflows, format string, and pointer overwrite vulnerabilities.

There is still space for improvement in the taint tracking area of research, as all of the proposals studied still have overheads that can be reduced through further optimization. This optimization can come in better policies for taint tracking that ensures that no redundant or unnecessary operations are carried out. As well as ensuring that the methodologies that are used to propagate the taint and manage the taints metadata are efficient and do not have excessive storage overheads.

The current methodology even with the current overheads of both computation and memory, do however still allow for securing of many applications regardless of the original programming languages and the application functions. Thanks to the binary-based policies and the ability to modify the policies by the user

iJETRM
International Journal of Engineering Technology Research & Management
(IJETRM)
https://ijetrm.com/

to suit the application. This means that using the current taint tracking proposals one can secure many applications preventing attacks, with slowdowns that are reasonable for the security being gained.

REFERENCES

1) James Newsome and Dawn Song. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *In Proceedings of the 12th Network and Distributed System Security Symposium* (NDSS'05).
2) W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige, "Tainttrace: Efficient flow tracing with dynamic binary rewriting," in ISCC'06. *Proceedings. 11th IEEE Symposium on. IEEE, 2006, pp. 749-754.*
3) G. Venkataramani, B. Roemer, Y. Solihin, M. Prvulovic, "MemTracker: Efficient and programmable support for memory access monitoring and debugging", *Proc. of the 13th Int'l Symp. on High-Performance Computer Architecture, 2007.*
4) Michael Dalton , Hari Kannan , Christos Kozyrakis, Raksha: a flexible information flow architecture for software security, *Proceedings of the 34th annual international symposium on Computer architecture, June 09-13, 2007, San Diego, California, USA*
5) Banerjee, Subarno & Devecsery, David & Chen, Peter & Narayanasamy, Satish. (2019). Iodine: Fast Dynamic Taint Tracking Using Rollback-free Optimistic Hybrid Analysis. 490-504. 10.1109/SP.2019.00043.