Tainting is Not Pointless

Michael Dalton, Hari Kannan, Christos Kozyrakis
Computer Systems Laboratory
Stanford University
{mwdalton, hkannan, kozyraki}@stanford.edu

Pointer tainting is a form of Dynamic Information Flow Tracking used primarily to prevent software security attacks such as buffer overflows. Researchers have also applied pointer tainting to malware and virus analysis.

A recent paper by Slowinska and Bos has criticized pointer tainting as a security mechanism, arguing that it is has serious, inherent false positive and false negative defects. We present a rebuttal that addresses the confusion due to the two uses of pointer tainting in security literature. We clarify that many of the arguments against pointer tainting apply only to its use as a malware and virus analysis platform, but do not apply to the application of pointer tainting to memory corruption protection. Hence, we argue that pointer tainting remains a useful and promising technique for robust protection against memory corruption attacks.

Categories and Subject Descriptors: C.0 [General]: Hardware-Software Interfaces; D.4.6 [Operating Systems:] Security & Protection – Information Flow Controls

General Terms: Security, Design, Experimentation, Performance

Keywords: Software security, pointer tainting, dynamic information flow tracking, buffer overflow, memory corruption, malware detection, virus detection

1. INTRODUCTION

Dynamic Information Flow Tracking (DIFT) has been used to prevent a wide range of security attacks [5, 9, 24, 18]. DIFT tracks the flow of untrusted information within a program's runtime by extending memory and registers with one or more *tag bits*. A tag bit is typically set if the register or memory word contains untrusted information received over the network or some other source. Tags are propagated at runtime according to a set of tag propagation rules. For instance, if the tag bit for one or more of the input operands of an arithmetic operation is set, the tag bit for the output operand is set as well to indicate that it was derived from untrusted data. Tags are systematically checked to detect security attacks. For example, a code injection attack can be prevented by checking if the tag bit of an instruction is set before the instruction is executed. By properly selecting the rules for propagation and checking of tag bits, DIFT can be used to prevent a wide range of attacks including high-level threats such as as web authentication and authorization bypass [12], SQL injection [23], command injection, directory traversal, and cross-site scripting [21].

DIFT can also prevent memory corruption attacks such as buffer overflows. During a buffer overflow attack, untrusted input typically overwrites pointer values, resulting in arbitrary code execution. Several DIFT analyses have been proposed to stop memory corruption attacks [5, 10, 14, 18, 24] using policies that rely either on *bounds check recognition (BCR)* or on preventing *pointer injection (PI)*. BCR-based policies [5, 9, 18, 24] prevent memory corruption by forbidding untrusted data from being dereferenced without first applying a bounds check operation. Due to false positives and negatives encountered when applying BCR-based policies in the real world, researchers have recently proposed PI-based policies for DIFT analysis [14, 10]. PI uses two tag bits: a pointer tag bit to track the flow of legitimate application pointers, and a taint bit to track the flow of untrusted data. PI forbids code from dereferencing tainted values that do not have the pointer tag bit set as well.

DIFT has also been applied for malware analysis [29, 30]. Malware systems typically perform fine-grained taint tracking to track the flow of sensitive information, such as user keystrokes or passwords. Malware is identified by observing when a software module attempts to leak sensitive input to an untrusted source, such as a remote network server. This is a separate and distinct problem from memory corruption attacks. Policies for buffer overflow protection such as BCR or PI protect non-malicious, but buggy or vulnerable software from untrusted input. DIFT policies for malware analysis analyze malicious, untrusted code and benign, sensitive user input. The threat model used in malware analysis is completely different from the threat model required for robust buffer overflow protection.

A recent paper by Slowinska and Bos in the 2009 Eurosys conference [22] has criticized DIFT's potential as a security mechanism. They state that DIFT policies for buffer overflow prevention and malware analysis are rife with false positives and negatives defects. Research papers pointing out flaws in existing work are an important contribution to academia, as they often serve as motivation for future research efforts. The authors of [22] make a number of excellent points, but in their criticisms they conflate DIFT policies for memory corruption with the entirely separate use of DIFT for malware and virus analysis.

In this paper, we address the criticisms raised in [22] that relate to preventing memory corruption using DIFT. In Section 2, we rebut arguments that conflate DIFT policies for memory corruption with those for malware analysis. Section 3 discusses the criticisms that apply only to BCR-based techniques, and not to the recent research

.

based on pointer injection [14, 11]. While BCR-based policies have been shown to have a number of real-world false positives and negatives [8, 10], PI-based analyses do not share these flaws. Finally, Section 4 discusses the criticism that the PI-based policy from [11] is not easily applicable to the Intel x86 architecture and Windows systems. We present related work and potential techniques that address these concerns. Overall, we argue that DIFT has a bright future as a memory corruption prevention mechanism, providing robust, low-overhead protection from malicious attacks on unmodified binaries.

2. CONFLATING MEMORY CORRUPTION & MALWARE ANALYSIS

Summary of criticism: The pointer tainting criticisms raised in [22] often conflate different uses of dynamic information flow tracking: preventing memory corruption attacks on unmodified, vulnerable, non-malicious software applications and, more recently, providing a platform for malware analysis platform in order to detect untrusted, malicious programs. The threat models for these two use cases are entirely different, and thus conclusions for these two areas cannot be directly compared.

The authors of [22] justly and correctly criticize the use of pointer tainting for malware analysis. Any attacker can evade the DIFT policies used in [30, 29] by laundering sensitive information information via control flow. For example, a byte-by-byte copy might be accomplished using 255 *if* statements such as "if x = 0 then y = 0", effectively copying x to y using branch conditions. This approach evades the tag propagation policy, allowing malware to copy sensitive information from x to y without causing y's tag bit to be set. The malware can then leak y to remote network hosts, evading malware detection algorithms.

This presents a serious challenge for researchers because information flow due to branch conditions may be caused due to code on paths that are not executed [27]. For example, the not-taken path of a condition may set a variable x to a new value. If x is observed to have its old value, then the branch condition was false, which leaks information about the variables used in the branch condition. Preventing this form of information flow requires static analyses to examine the full control flow graph of the program being analyzed. On the common Intel x86 platform, static disassembly is undecidable [16] as the x86 is a variable-length ISA which has no requirements for instruction alignment. Even with perfect data and control flow tracking, malicious software can still employ covert timing channels to completely bypass all existing DIFT malware analyses [4].

One possible solution may be the approach taken in [17], where symbolic execution is used to determine the number of bits the attacker can determine when computing the result of a particular operation. This approach could be used to detect when an attacker is using branch condition or load/store address to propagate untrusted information. Another solution is to perform DIFT at a higher level of abstraction that fully encapsulates control flow, such as by tracking information flow at the object or process granularity [15, 31]. However, none of these solutions address covert timing channels, which can only be prevented by a DIFT solution at the hardware gate level [25]. Such a low-level approach is required to prevent information leaks due to inherent covert timing channels in modern hardware designs, such as cache or TLB misses. This solution requires significant hardware modifications and all protected applications must be modified or rewritten to support the new gate-level DIFT ISA instructions.

Rebuttal: None of these criticisms directed towards DIFT policies for malware detection apply directly to DIFT policies for memory corruption attacks. DIFT policies for memory corruption attacks have an entirely different threat model than malware prevention. When preventing buffer overflow attacks, the target application is benign, but may contain security vulnerabilities triggered by malicious input. In malware analysis, the potentially infected program is untrusted, but the input is sensitive, benign user keystrokes or files. The malware analysis must detect when the malicious code attempts to leak sensitive information, which can be done in many ways, including covert storage and timing channels.

DIFT policies for memory corruption prevention need only track common flows of information such as data movement. Since the underlying application is trusted, we can safely assume that it will not attempt to launder information via control flow or covert channels. No known memory corruption attacks in the real world rely on implicit information flow. Moreover, DIFT policies for memory corruption ensure that malicious input is never executed as code by forbidding the execution of tainted instructions. Thus, this use of DIFT is not mired in the false positive/false negative quagmire of tracking implicit information flow in untrusted code via branching and other control flow operations. The experimental results in [22] do not demonstrate any new false negatives or positives for state of the art memory corruption policies.

3. POLICIES FOR MEMORY CORRUPTION PREVENTION

Summary of criticism: We now focus on criticism correctly targeted towards DIFT policies for memory corruption. In Section 3.2, the authors of [22] state that memory corruption protection cannot be provided in the general case because of false positives due to bounds check recognition errors. The claim is partially evaluated in Section 5.1, which tests various policies for recognizing validation operations. The results show that, under standard bounds check recognition policies, false positives are extremely likely in real-world applications.

The difficulty of bounds check recognition with BCR-based policies is a well-known issue that has been discussed in prior DIFT work [8, 11, 14]. BCR-based policies untaint data when a bounds check occurs [5, 18, 24]. Most policies simply assume that all bounds check operations are comparisons. Unfortunately, this assumption leads to significant *false negatives* [9, 14]. Not all comparisons are bounds checks. For example, the glibc $strtok()$ function compares each input character against a class of allowed characters, and stores matches in an output buffer. If the DIFT policy interprets these comparisons as bounds checks, the output buffer is always untainted, even if the input to $strtok()$ is tainted. This can lead to false negatives such as failure to detect a malicious return address overwrite in the atphttpd stack overflow exploit [1].

However, the most critical flaw of BCR-based policies is an unacceptable number of *false positives* with commonly used software. Any scheme for input validation on binaries, including bounds check recognition, has an inherent false positive risk. While BCR-based policies untainted the variable that is bounds checked, none of the aliases for that variable in memory or other registers will be automatically validated. The use of aliases can lead to false positives. Moreover, even trivial programs can cause false positives because not all untrusted pointer dereferences need to be bounds checked [9]. Many common glibc functions, such as $tolower()$, $toupper()$, and various character classification functions ($isalpha()$, $isalnum()$, etc.) index an untrusted byte into a 256 entry table. This is completely safe, and requires no bounds check. However,

BCR policies fail to recognize this input validation case because the bounds of the table are not known in a stripped binary. Hence, false positives occur during common system operations such as compiling files with gcc and compressing data with gzip.

In practice, false positives in BCR-based policies occur only for data pointer protection. No false positive has been reported on x86 Linux systems so long as only control pointers are protected [7]. Unfortunately, control pointer protection alone has been shown to be insufficient as a security mechanism that prevents memory corruption attacks [6].

Rebuttal: While real-world false positives and negatives are a major issue with BCR-based policies, this not the case for the recently proposed PI-based policies that do not attempt to recognize bounds checks or any other validation operators. In fact, the papers on PI policies [11, 14] list as motivation code fragments that lead to the false positive and negative scenarios listed in Section 6.2 of [22].

Pointer-injection policies use two tag bits: a pointer tag bit to track the flow of legitimate application pointers and a taint bit to track the flow of untrusted information. The pointer bit is set for all instructions and data that refer to statically allocated memory at startup. Furthermore, the return values of any dynamic memory allocation system calls have the pointer bit set. Both pointer and taint bits are propagated at runtime during program execution. A security attack is reported if a tainted pointer is dereferenced and the pointer bit is clear. This occurs when untrusted input overwrites application memory containing pointers during a buffer overflow attack.

Pointer injection policies *never* clear taint bits due to possible validation instructions such as bounds checks. The only way to clear a tainted variable is to overwrite its storage location with trusted, untainted information. Hence, PI-based policies do not exhibit the false positive issues associated with input validation recognition in BCR-based policies.

PI-based policies are also robust against false negatives. Existing buffer overflow attacks rely on overwriting pointers rather than non-pointer data. PI identifies legitimate pointers in the application and tracks their flow at runtime. Attacker input has the taint bit set and the pointer bit clear. If during a buffer overflow, the attacker input overwrites a valid pointer, DIFT checks will prevent the application from dereferencing this pointer as its taint bit will be set but the pointer will be clear. The PI policy is sufficient to stop existing attacks such as all control or function pointer overwrites, data pointer overwrites(such as most heap exploits), all GOT overwrites, and all standard stack smashing attacks. PI protection can also be complemented with protections for non-pointer data. Existing work using information flow or tags to protect non-pointer data includes red-zone bounds checking for heap objects [10] and static analyses to determine non-pointer data regions that should never become tainted [3].

The robustness of PI-based policies has been demonstrated experimentally for both userspace and kernelspace code using the Raksha system [11]. DIFT propagation and checks were enabled for all programs and common workload scenarios, including booting Gentoo Linux, sending e-mail via Sendmail, and serving web pages via Apache. Even the Linux kernel, including optimized handwritten assembly code for memory copies and context switching, was protected with a PI-based policy without false positives. The policy was sufficient to correctly detect a wide range of security attacks, from traditional stack and heap userspace buffer overflows to kernelspace buffer overflows and even user/kernel pointer dereferences.

4. PRACTICALITY OF IMPLEMENTATION

Summary of criticism: The final set of criticisms in [22] discusses the practicality of implementing the PI-based policy described in [11]. In particular, Slowinska and Bos suggest that the policy successfully used by Raksha on the SPARC architecture and the Linux operating system cannot be easily ported to the x86 architecture and closed source operating systems such as Windows. This is because the PI-based policy in [11] requires reliable identification of pointers.

Pointers in applications can come from dynamic memory allocation system calls such as mmap and brk, or references to statically allocated memory embedded in the code and data of the executable. The policy in [11] relies on system call interposition to identify pointers from dynamic memory allocation system calls. This may cause compatibility problems if the system call ABI changes, or if system calls are undocumented as is often the case in proprietary OSes. Additionally, the method used for detecting static pointers in the code and data of executable requires scanning and disassembly of the object code of libraries and executables at startup. The implementation used in [11] was a SPARC V8 platform, and allowed for reliable disassembly due to the fixed-length instructions in the SPARC ISA. However, the x86 has variable-length instructions, and precise disassembly of x86 instructions is undecidable [16].

Rebuttal: While the exact PI-based policy in [11] has not been ported to the Intel x86 yet, we do not believe that this is due to any fundamental difficulty. In fact, the original paper introducing pointer-injection DIFT was implemented entirely on the Intel x86 using the popular Bochs full-system emulator [14]. This paper differs from the approach taken in [11] as it performs page table scans at runtime to determine dynamically if a pointer may refer to statically allocated memory. Since this scan can be prohibitive in terms of performance, the policy in [11] examines the code and data of an executable once at startup to identify pointers to statically allocated memory, and does not require page table walks at runtime.

To identify pointers to statically allocated memory (such as a global variable), the policy in [11] analyzes ELF executables at startup. Specifically, it scans the code section for a particular SPARC instruction (sethi) that is required by the ELF object file format when initializing a pointer value [19]. To port this technique to the x86, two problems must be solved. First, reliable static disassembly on the x86 is undecidable [16] because the x86 uses variable-length, unaligned instructions. Second, the ELF object file format does not restrict pointer initialization to a single instruction on the x86 [20].

However, [11] presents possible solutions to both problems. The x86 ELF object file format restricts references to statically allocated memory to the few x86 instructions that allow a 32-bit constant operand, such as `mov` [20]. To conservatively identify instructions used to initialize pointers, the DIFT system can scan the executable code at startup for valid pointer values at any offset (as x86 instructions are unaligned), and then scan backwards to see if any of the preceding bytes can form an instruction with a 32-bit constant operand. As x86 instruction size is limited to 17 bytes, this backwards scan is bounded to a constant length.

The other concern presented in [22] is that the techniques in [11] cannot be used on closed-source OSes such as Windows. System call interposition can be performed without valid source code by hooking the interrupt descriptor table [13], using dynamic binary translation tools in software [2, 28], or employing hardware support, such as the recent hardware support for virtual machines added by Intel and AMD [26]. A DIFT implementation on Windows may still perform any kernel-related instrumentation by creating a kernel module and interposing on system calls directly for

tasks such as initializing tags after a memory allocation system call. Furthermore, if kernel heap regions cannot be identified, any value that points into the virtual address range of the OS kernel can be conservatively treated as a valid pointer.

Overall, the work presented in [11] has promising solutions for the challenge of implementing PI-based DIFT policies for x86/Windows platforms. Such an implementation is quite possible, and should not be dismissed out of hand as is done in [22]. The Intel/Windows platform was once also considered to be non-virtualizable in real-world systems, until VMWare used dynamic binary translation techniques to accomplish what was formerly considered the impossible. Even if researchers only succeed in protecting Intel/Unix platforms, DIFT will have significantly improved the security of the majority of critical, high-impact servers.

5. CONCLUSIONS

We have argued that DIFT is a powerful and flexible platform for preventing software security vulnerabilities, from SQL injection [23] to buffer overflows [10]. We have demonstrated that many of the criticisms by Slowinska and Bos [22] apply only to the application of DIFT to malware analysis, which has an entirely different threat model than preventing buffer overflows. DIFT can be applied to many different attacks, and limitations or drawbacks for a particular form of attack cannot be directly applied to another attack with a different threat model. We have also shown that real-world false positive and negatives are observed only in older, bounds check recognition-based policies. The latest DIFT buffer overflow policies are based on pointer injection [10, 14] and have no observed real-world false positives, even when protecting the Linux kernel. Finally, we argue that the latest pointer injection policies can be ported to the Intel x86, and should not be dismissed out of hand as SPARC or Linux-specific.

Although we agree with Slowinska and Bos that DIFT is not perfectly suited for malware analysis, we strongly disagree that these limitations affect in any way the unprecedented success researchers have encountered when using DIFT to prevent input validation attacks. Never before has a single security technique been so flexible (preventing attacks from high-level cross-site scripting to low-level format strings and buffer overflows), so compatible (works on unmodified binaries without debugging information), so performant (no overhead with hardware support), or so secure (comprehensively prevents most attacks). *Tainting is not pointless.*

Acknowledgments

This work was supported by Stanford Graduate Fellowships funded by Sequoia Capital and Cisco Systems, an Intel PhD Fellowship, and NSF award CCF-0701607.

6. REFERENCES

[1] ATPHTTPD Buffer Overflow Exploit Code. `http://www.securiteam.com/exploits/6B00K003GY.html`, 2001.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX '05: Proceedings of the USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[3] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th conference on Operating Systems Design and Implementation*, 2006.

[4] L. Cavallaro, P. Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *GI SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, July 2008.

[5] S. Chen, J. Xu, et al. Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In *Proceedings of the Intl. Conference on Dependable Systems and Networks*, 2005.

[6] S. Chen, J. Xu, et al. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[7] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Intl. Symposium on Microarchitecture*, 2004.

[8] M. Dalton, H. Kannan, and C. Kozyrakis. Deconstructing Hardware Architectures for Security. In *the 5th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2006.

[9] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *Proceedings of the 34th Intl. Symposium on Computer Architecture*, 2007.

[10] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium*, pages 395–410, 2008.

[11] M. Dalton, H. Kannan, and C. Kozyrakis. Real-world buffer overflow protection for userspace and kernelspace. In *Proceedings of the 17th Annual USENIX Security Symposium*, pages 395–410, 2008.

[12] M. Dalton, N. Zeldovich, and C. Kozyrakis. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th Annual USENIX Security Symposium*, 2009.

[13] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[14] S. Katsunuma, H. Kuriyta, et al. Base Address Recognition with Data Flow Tracking for Injection Attack Detection. In *Proceedings of the 12th Pacific Rim Intl. Symposium on Dependable Computing*, 2006.

[15] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard os abstractions. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 321–334, New York, NY, USA, 2007. ACM.

[16] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *In ACM Conference on Computer and Communications Security (CCS)*, pages 290–299. ACM Press, 2003.

[17] J. Newsome, S. McCamant, and D. Song. Measuring channel capacity to distinguish undue influence. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 73–85, New York, NY, USA, 2009. ACM.

[18] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[19] Santa Cruz Operation. *System V Application Binary Interface, SPARC Architecture Processor Supplement*, 1996.

[20] Santa Cruz Operation. *System V Application Binary Interface, Intel386 Architecture Processor Supplement*, 1997.

[21] P. Saxena, D. Song, and Y. Nadji. Document structure integrity: A robust basis for cross-site scripting defense. *Network and Distributed Systems Security (NDSS) Symposium*, 2009.

[22] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the 4th ACM European Conference on Computer Systems*, pages 61–74, New York, NY, USA, 2009. ACM.

[23] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *Proceedings of the 33rd Symposium on Principles of Programming Languages*, 2006.

[24] G. E. Suh, J. W. Lee, et al. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[25] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 109–120, New York, NY, USA, 2009. ACM.

[26] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, 2005.

[27] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Intl. Symposium on Microarchitecture*, 2004.

[28] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[29] H. Yin, Z. Liang, and D. Song. HookFinder: Identifying and understanding malware hooking behaviors. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, February 2008.

[30] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[31] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.