Verified and Optimized Inlined Reference Monitors

PhD Candidacy Proposal

Bin Zeng
Department of Computer Science and Engineering
Lehigh University
zeb209@lehigh.edu

May 5, 2013

Abstract

Current software stacks are built on top of unsafe languages such as C and C++. Software attacks sabotage program executions by inducing control flow transfers to shellcode or manipulating data pointers to read/write sensitive information. By embedding security checks into subject programs during compilation, many attacks can be foiled effectively. In this proposal, we investigate current software attacks and review the existing approaches and their strengths and weaknesses. Finally, we propose research directions and report the current status of our work.

Contents

1 Introduction

In recent years, software attacks increased steadily even when a large amount of resources have been invested into vulnerability mitigations and defenses. Software attacks continue to plague the software industry and users. As a recent example, Twitter, Facebook, Apple and Microsoft have been hacked during the last year and suffered enormous economical loss and intangible reputation damage [32].

One of the major culprits is the ubiquitous use of unsafe languages such as C and C++. Although many type-safe languages such as Java, C#, Python and Ruby et al. gained popularity and started to replace weakly-typed languages in some areas, many software systems are still written in C and C++. A full transition to safe languages is unlikely to happen any time soon. Even for safe languages, their trustworthiness is questionable because their compilers, virtual machines and interpreters are mostly written in C and C++.

Through embedding security checks into programs written in unsafe languages, many vulnerabilities can be mitigated. As a simple example, Control Flow Integrity (CFI) [3] inserts dynamic checks before computed control flow transfers to ensure that execution paths follow predetermined control flow graphs, constructed by source-level static analysis, binary analysis or even program profiling. Many attacks including traditional buffer overflows and arc-injection attacks can be effectively foiled by CFI. As a simple example, traditional buffer overflow attacks overwrite return addresses, pushed onto the stack by call instructions, to point to injected shellcode. CFI ensures that return instructions can only return to the instructions after the calls to the function, thus preventing buffer overflows. As another instance, return-to-libc attacks cannot happen with CFI enforced because the attacker-induced control flow transfers to libc functions violate the predetermined control flow graph.

While some security mechanisms are widely deployed such as stack guard, others see little adoption because they are either ineffective, unrobust, inefficient, impractical or a combination of them. In general, security mechanisms are evaluated according to four key metrics:

- **Effectiveness**. How effectively the security mechanism defends against a wide vector of security attacks. No panacea defense exists for all attacks. A security mechanism targets certain attacks. For example, CFI can defend against arc-injection attacks but can not prevent memory corruption attacks. Furthermore, the effectiveness of CFI varies with the precision of the enforced control flow graph.

- **Robustness**. How well the security mechanism resists being bypassed or circumvented when encountered with well-resourced and patient adversaries. As a simple example, Address Space Layout Randomization (ASLR) [10, 11, 23] is effective against many attacks that require the exact memory locations of objects. However, it can be circumvented or even broken through brutal-force attacks if the programs are poorly-randomized and the entropy is low [4, 17, 35].

- **Efficiency**. How much cost the security mechanism incurs. Some security mechanisms are lightweight while others incur heavy runtime overhead. Security mechanisms with heavy overhead can be used in security-critical systems such as the backend of banking

systems while they see little adoption in performance-critical systems. Mere ten percent of performance improvement can save thousands of server racks in a typical data center.

- **Practicality**. How easy it is to deploy the security mechanism in current production systems. Many security mechanisms are incompatible with legacy code or require even major hardware modifications. Some security mechanisms are intrusive while others are composable and can be added as an additional layer to the current system or even embedded into the programs.

1.1 Thesis Statement

Inlined Reference Monitors (IRMs) embed security checks into programs to enforce certain security polices [18, 20, 21]. The embedded checks can validate many properties of subject programs. In general, IRMs can be enforced through the combination of static verification and dynamic checks. For those sensitive operations whose properties can be statically proved to be safe, checks are unnecessary. For those that depend on dynamic information, checks are required. IRMs are powerful and can enforce many security policies while incurring relatively low overhead compared to other mitigations.

Unfortunately, none of the existing security mechanisms including IRMs satisfy all of the aforementioned requirements. For instance, Pittsfield sandboxes only memory writes but not reads, thus ineffective against data confidentiality attacks [24]. Also, although Pittsfield incurs relatively low overhead, it might still be problemetic in performance-critical systems. As another example, NaCl x86-32 uses hardware segmentation to isolate mobile binary code [41], thus incurring very little runtime overhead. However, hardware segmentation is unavailable on 64-bit x86-64. Instead, NaCl x86-64 inserts checks into subject programs to ensure memory accesses fall into an allowed region. However, memory read sandboxing is disabled by default due to high runtime overhead. In addition, Pittsfield, NaCl x86-32 and NaCl x86-64 binaries are all locked down to a specific ISA and thus incompatible with other ISAs such as the prevalent ARM processors.

As another example, current low-level IRMs are enforced through binary rewriting [1], assembly instrumentation, or transformations on equivalent representations. These low-level representations are tightly coupled with a specific Instruction Set Architecture (ISA) and thus incur poor retargetability. The IRM enforcement for x86-32 cannot be easily ported to ARM or even x86-64. In addition, low-level representations are hostile to program analyses and optimizations, which are necessary for efficient IRM enforcement. The wealth of structured information gets lost or diluted during the lengthy compilation process before it reaches low-level representations.

Enforcing IRMs at a machine-indepedent high-level representation brings many benefits. First, the IRMs are retargetable. The check instrumentation and optimizations can be reused by all the targets supported by a compiler. Only the lowering component needs to be customized for each target. Second, the high-level representations bring the wealth of structured information, amenable to program analyses and optimizations. For example, LLVM IR is based on Unlimited Register Machine (URM) model in Static Single Assignment (SSA) form, which renders it extremely easy to find the definition of a variable. LLVM IR contains explicit control flow information, loop information, dominator tree information,

type information, and even some source-level information, which facilitate program analyses and optimizations. Third, high-level representations hold many data structures that are amenable to program transformations. As a simple example, LLVM IR contains only 57 instructions while the target for x86 has over 3500.

In the dissertation research, we strive to design and implement a framework for IRMs towards the aforementioned metrics. The framework will be built on the industrial-strength compiler framework Low-level Virtual Machine (LLVM). The checks are inserted on the LLVM IR level and lowered into various machine instruction sequences for different targets in the backend. Since LLVM IR is machine agnostic, all the check instrumentation and optimizations can be reused by all the supported targets. Only the lowering component needs to be added for a new target. In addition, the wealth of structured information, as noted above, can be used for optimizations to reduce the overhead of IRMs. In order to remove all the compiler passes and check instrumentation and optimizations out of the Trusted Computing Base (TCB), a verifier is needed to validate the final secured programs, ensuring trustworthiness.

Programs written in unsafe languages such as C and C++ are vulnerable. These programs can be secured through IRMs whose efficiency, trustworthiness and practicality can be enhanced through high-level enforcement, aggressive optimizations and verification.

Expected Contributions. The major contributions of my dissertation research will be threefold:

1. A retargetable and reusable framework for various IRMs.
2. A set of optimizations in the framework to decrease the overhead of IRMs.
3. A verification mechanism to verify the optimizations and ensure the trustworthiness of the framework.

The proposal is organized as follows. Section 2 introduces pressing threats. Section 3 describes existing defenses and analyzes their strengths and weaknesses. Section 4 describes our research plan and Section 4.5 presents the future work and the last section concludes the proposal.

2 Threats

Many attacks have been discovered and ingenious defenses have been proposed in recent years. In this section, we investigate the current attacks. A thorough understanding of current threats facilitates the design and implementation of effective defenses. This section introduces various types of software attacks such as buffer overflows, return-to-libc attacks, Return-Oriented Programming (ROP), and null-pointer dereferences.

Software attacks have come a long way since stack smashing. Novel attacks and exploits have been invented such as arc-injections, null-pointer dereferences, format bugs, use-after-free, and uninitialized reads et al. Oftentimes, maliciously crafted input comes through a remote communication channel and triggers vulnerabilities inside buggy programs and takes over the program execution. Injecting shellcode has been a dream goal for attackers.

Although buffer overflows, also referred to as buffer overruns, were first maliciously exploited back in 1988 by Morris Worm, they still account for the largest share of CERT advisories [30]. Traditional buffer overflows try to overwrite return addresses stored on the stack to point to injected shellcode. Buffer overflows have metamorphosed into other forms recently as many defenses have been deployed such as Data Execution Protection (DEP) [25], Address Space Layout Randomization (ASLR) [40] et al. Shellcode injection becomes difficult when DEP is deployed. As a result, the return address is changed to point to existing C library code, as opposed to injected shellcode, resulting in return-to-libc or arc-injection attacks. If the return address is modified to direct control flow to existing application code snippet itself, it is called (ROP) [14, 34]. ROP chains together *gadgets* discovered in existing code section to achieve various functionality intended by attackers. Researchers have found that these *gadgets* can form a Turing-complete language [13, 14]. Since ROP can bypass many existing vulnerability mitigations such as DEP, it gained enormous attention in recent attacks and presents a pressing challenge for security researchers.

Another variant of buffer overflow is called pointer subterfuge. Instead of return addresses, pointer subterfuge change the program's control flow by overwriting function pointers pervasive in large programs, especially those written in C++ due to virtual method table (vtable). Thus, pointer subterfuge can bypass security mitigations such as stack guard.

Heap smashing allows exploitation of buffer overflows in dynamically allocated memory, as opposed to on the stack. Instead of return addresses, heap smashing can modify arbitrary memory through overwriting metadata maintained by memory allocators.

Many programs use value NULL to flag that a pointer is uninitialized or undefined. It is used often as a loop termination condition. Usually, dereferencing a null pointer causes a segmentation fault on Unix and Unix-like machines. Up until recently, researchers did not consider it to be a security vulnerability. Dowd et al. discovered an ingenious way to exploit null-pointer dereferences [16].

Many other attacks exist. We only enumerated the common and pressing attacks.

3 Related Work

This section introduces closely related security mechanisms including IRMs, CFI, Software-based Fault Isolation (SFI) et al. and analyzes their strengths and defficiencies.

IRMs inserts check to guard sensitive operations. Positioning checks before sensitive operations has a long history. Researchers have been inserting various checks into programs for long. For example, at source level, programmers conduct various checks inside subject source programs to inspect whether certain conditions hold. As a common practice to improve software reliability, developers insert assertion statements into source programs to check the state of program executions. As another concrete instance, authentication programs verify whether a user has legal identity. At the intermediate representation level, researchers have been doing bounds checking to ensure memory safety [5, 9, 15]. At the machine code level, machine instruction sequences are inserted to enforce certain security policies such as CFI [2, 3, 39, 42] and SFI [24, 33, 38, 41, 42].

Clearly, this is a promising and well-studied research area. Subsequently, we will discuss the closely related work.

Software-based Fault Isolation. SFI, also known as sandboxing, isolates untrusted or faulty modules from the rest of the system to ensure coarse-grained memory safety [24, 33, 36, 38, 41]. A carefully designed interface is the only pathway untrusted modules can interact with the rest of the system. The original SFI was proposed for fault isolation and later on it was adopted for security [38]. Large softwares enrich their functionality through extensions [24, 37]. Those extensions may come from untrusted sources or contain vulnerabilities that render the whole system weak, thus need to be isolated so that their failures do not destabilize the whole system. Their communications with the rest of the system is restricted through a carefully designed interface. SFI can be implemented either with the assistance of hardware or through a software-based approach. NaCl x86-32 isolates untrusted modules through hardware segmentation in x86-32 which is deprecated in x86-64 and unavailable on other popular Instruction Set Architectures (ISAs) like ARM. Type-safe languages such as Java [22], Typed Assembly Language (TAL) [26, 27], and Proof-Carrying Code (PCC) [6, 28, 29] enforce memory safety through a language-based approach. However, either they are unavailable for mainstream languages such as C and C++ or they impose considerable burdens on programming productivity and runtime performance. Alternatively, SFI can be implemented through the combination of static verification and inlined checks. For direct memory accesses, the memory addresses can be statically verified. For computed memory visits, inlined checks are inserted to make sure that the memory accesses do not read or write disallowed memory region. To the best of our knowledge, traditional SFI implementations are done through binary rewriting or assembly instrumention [24, 33, 38, 41, 42]. They are tightly tied to a specific target machine and difficult to be ported to other ISAs.

Control Flow Integrity. One subtle requirement by SFI and other IRMs is that the inserted security checks cannot be bypassed by adversaries, thus requiring some form of control flow restriction. Conventional IRMs have to restrict control flows through code pointer alignment by padding with *nop*s, which incurs additional overhead. CFI ensures that runtime control flows follow a predetermined control flow graph even if the whole data memory is under the control of attackers [2, 3, 42]. CFI guarantees a strong control flow restriction than required by IRMs, thus can serve as the cornerstone for SFI and other IRMs. It can be implemented by the combination of static verification and inlined dynamic checks.

One way to enforce CFI is to inserts ID's at the targets of computed control transfers and checks whether the ID at the target address matches the one encoded within the check positioned before computed control flow transfers [3]. Wang et al. enforces CFI through defunctionalization [39]. Direct control flow transfers are statically verified. For computed control flow transfers, they encode all the potential targets in a write-protected table and uses an index to retrieve the target address. Before each computed control transfer, the index is checked to make sure that it falls into the table before used to fetch the target address.

XFI. On top of CFI, XFI employs a protected shadow stack to store return addresses [19]. XFI and CFI enhance each other with XFI promoting control flow precision from Deterministic Finite Automata (DFA) to Pushdown Automata (PDA) and CFI providing the guarantees for control flow restrictions [3]. However, XFI is platform specific and and its performance might be problematic for production systems.

Bounds Checking. Bounds checking examines whether a variable is within bounds before it is used [5]. It can detect memory errors in unsafe languages such as C and C++ [7].

However, bounds checking incurs heavy overhead which prevents them from being adopted even after optimizations are applied. The incompatibility with legacy libraries is another obstacle for its application [5].

In this section, we discussed current protection mechanisms including IRMs, CFI and SFI et al. IRMs is a general security mechanism by inserting checks directly into subject programs to enforce certain security policies. The security policies can be modeled as a security automata. CFI, SFI and XFI are special examples of IRMs. IRMs are implemented through binary rewriting, assembly transformation, or other low-level code instrumentation. Components can hardly be shared among different IRMs. Even porting the same IRM for a different target requires substantial work.

4 Research Plan

The key goal of my dissertation is to build a framework for efficient and trustworthy IRMs enforcement that meets all the aforementioned requirements for security mechanisms. Compilers serve as the heart for program analyses and transformations. The abundance of high-level information, succinct intermediate representations, convenient data structures all facilitate program analyses and optimizations. Many optimizations are done during compilation. The framework will be built on top of the industrial-strength compiler LLVM. Security checks are inserted into subject programs at different stages of the compilation pipeline. For efficiency, unnecessary security checks are optimized away during compilation. In order to minimize the Trusted Computing Base (TCB), the transformed programs after compilation needs to be verified to ensure trustworthiness.

We have identified a vector of problems with existing IRMs and proposed new techniques to solve them. The first problem with the traditional SFI implementation is that only memory writes are sandboxed for information integrity because sandboxing memory reads incurs heavy runtime overhead. In other words, data integrity is protected but confidentiality is sacrificed for efficiency. Data confidentiality is extremely important for users. For example, bank accounts, personal information, credit card numbers, trade secrets, government documents are valuable and should be protected from unauthorized parties. In some ways, confidentiality is more important than integrity. We have proposed, implemented and evaluated an innovative data sandboxing mechanism by combining CFI and data sandboxing. To reduce the overhead incurred by sandboxing memory reads, we have introduced optimizations using static analysis to cut down the overhead. The second problem with traditional SFI implementations is that they are tightly tied to a specific ISA, resulting in poor retargetability and reusability. Conventional SFIs are implemented through binary rewriting, assembly instrumentation or transformations on equivalent representations. It is very difficult and time-consuming to port one implementation to another ISA. We propose to enforce SFI at a machine-independent level. Next, we elaborate on the problems and our solutions.

4.1 Enforcing Data Confidentiality through Combining CFI and Data Sandboxing

Conventional SFI implementations such as Pittsfield insert security checks before memory writes for data integrity, but not reads due to the heavy overhead incurred by sandboxing memory reads, as there are many more memory reads than writes in a typical program. Untrusted code can read arbitrary memory regions, thus violating confidentiality. Futhermore, in order for the inserted checks not to be circumvented by attackers, traditional SFI implementations divide programs into chunks and aligned them at memory locations by padding with *nop*. Control transfers are sandboxed so that they can only branch to the starting addresses of chunks. In other words, SFI implementations require a certain level of control flow restriction. Although *nop* instructions do not cause side effects, they incur considerable overhead because they take decoding resources inside processors. Modern processors can decode at most four instructions simultaneously. *nop* instructions consume the limited decoding resources.

Solution. In order to protect sensitive data from untrusted code, we have proposed an innovative data sandboxing mechanism by combining CFI and full data sandboxing for both memory reads and writes. As noted above, inlined reference monitors require a certain level of control flow restriction so that inserted security checks cannot be bypassed by adversaries. CFI provisions a stronger control flow guaranttee than required by inline reference monitors. By combining control flow integrity with data sandboxing, the control flow requirement can be discharged. In addition, with CFI enforced, many optimizations become possible because CFI ensures the strong control flow guaranttee even when attackers control the whole data memory region. In order to decrease the runtime cost caused by sandboxing both memory reads and writes, we have proposed, implemented and evaluated three optimizations to reduce the runtime overhead. These optimizations strive to reduce the number of inserted security checks, thus reducing the runtime overhead. The first optimization is redundant check elimination. Some checks are unnecessary because the subject pointer has been checked already and not modified before the second use. The second optimization is sequential memory access optimization. Many memory visits calculate memory locations by adding a small constant to a base pointer. The base pointer is shared among multiple memory visits. Thus, if the constant offset is smaller than the guard region size and the base pointer has been checked already, the subsequent checks can be removed without harming the trustworthiness. The last optimization, called loop-based check hoisting, handles the situation when checks reside in loops. Loops account for the majority of program execution time. Checks inside loops can cause considerable overhead. This optimization hoists loop checks to its preheader thus reducing the number of times a check is executed. To further ensure the trustworthiness of the optimizations, we have proposed and implemented a range analysis based verifier to validate the final product, thus ensuring trustworthiness by eliminating all the optimizations and transformations from the TCB.

We have fully implemented the solutions in industrial strength compiler framework LLVM and evaluated the performance on SPECint2000. The runtime overhead of full data sandboxing by combining CFI and static analysis is only 27.15%, significantly lower than previous work.

Publication. Our paper based on the work was published on the ACM CCS 2011

conference.

4.2 Enabling Retargetability for Data Sandboxing

Conventional IRMs enforcements are implemented through binary rewriting, assembly instrumentation or transformations on equivalent low-level representations. These low-level representations are machine-dependent and tightly tied to a specific ISA. Thus, one implementation for a specific ISA cannot be easily ported to another architecture, thus incurring poor retargetability and reusability. It is none-trivial and time-consuming to implement an IRM for a different ISA. As a simple example, the NaCl x86-32 uses hardware segmentation to isolate untrusted modules. However, hardware segmentation is unavailable on 64-bit x86-64. Thus, sandboxing has to resort to software-based approach via inserting security checks directly into subject programs. This approach cannot be easily adapted to ARM processors. ARM processors have different instructions and thus require a reimplementation from ground up. It is demonstrated by that NaCl's support for ARM processors came years after the support for x86-32.

Solutions. In order to enable retargetability and reusability for inlined reference monitors, we propose to build a framework to enforce inlined reference monitors at a machine-independent level, i.e. LLVM IR. LLVM IR is the intermediate representation used by LLVM compiler framework. To enforce the IRMs at LLVM IR, all the security check instrumentation and optimization modules can be shared by various target machines supported by the LLVM compiler. Only the lowering part needs to be customized for each target machine, which is only a small fraction of the whole code base. Thus, the IRMs enforcement is retargetable. In addition, LLVM IR has many characteristics amenable to program analysis and optimizations. The whole LLVM intermediate language is designed for program analyses and optimizations. As a simple example, in order to find the definition of a variable at machine-code level, complicated and error-prone dataflow analysis has to be employed while it is extremely easy to do so in LLVM IR which is in SSA form. Furthermore, LLVM IR contains much less instructions than a typical target machine (LLVM IR has only 57 instructions while the X86 target contains more than 3500 instructions). LLVM IR contains high-level structured information such as type information, loop information and even dominator tree information that can be used for program analyses and optimizations. However, enforcing low-level security mechanisms at a high-level IR can weaken the trustworthiness of low-level security mechanisms, as the compiler passes after check instrumentation pass can violate the assumptions assumed by low-level security mechanisms. Most compilers assume a machine model looser than a typical attack model. Thus, compiler transformations can violate the attack model when doing optimizations. As a simple example, optimizing compilers assume that a memory load from a stack slot will return the value last stored to that slot. This is true when the program is not under attacks. However, given the abundance of memory corruption bugs, attackers can easily overwrite stack slots through buffer overflows. Thus, the value of untrusted memory location can be changed arbitrarily by attackers. As another example, return address is pushed onto the stack when a function gets called. The compiler assumes that the return address will stay the same during the execution of the called function. This is not true given the wealth of buffer overflow vulnerabilities. Thus, in order to ensure the trustworthiness of enforcing IRMs at LLVM IR, we designed a path-sensitive range analysis

based verifier to certify the final output. Range analysis based verifier is powerful so that various optimizations can be verified.

Publication. Based on the work, we have submitted a paper to the prestigious USENIX Security 2013. The paper has been accepted.

4.3 Improve the efficiency of Dynamic Taint Tracking

Taint tracking, also referred to as taint analyis, is a powerful security mitigation against memory corruption attacks. It is a special form of information flow analysis and can detect whether untrusted users have manipulated control flow, thus preventing buffer overlfow attacks. Dynamic taint tracking dynamically tracks down the information flow from untrusted users, i.e. taint source, to sensitive operations, i.e. taint sink. The taint sources can be files such as pdf, mp3 or html files, network packets, keyboard, mouse and touchscreen input, webcam et al. The taint sinks can be sensitive operations such as control flow transfers, and system calls. During program execution, the dynamic taint tracker propagates taint to keep track of tainted information according to a set of rules, called taint propagation rules. If an operation uses the value of some tainted object, say X, to compute the value of another object, say Y, then object Y becomes tainted. Taint propagation is transitive.

Dynamic taint tracking can be used for attack detection and prevention. Current software attacks usually arrive as user input from a remote communication channel, and once resident in a program memory, triggers pre-existing software flaws and overwrites control data to hijack program execution. Taint tracking can track down the information flow from untrusted users to sensitive operations including control flow transfers. It can detect whether user-controlled data reach sensitive operations such as control flow transfers.

Unfortunately, current dynamic taint tracking incurs prohibitively heavy overhead. Current implementations are so slow that it sees little adoption in production systems. Even the fastest dynamic taint tracker doubles the program execution time. For example, Argos, a dynamic taint tracker based on QEMU, incurs 15x-26x slowdown depending on the benchmark program [31]. As another example, Minemu, the world's fastest dynamic taint tracker, incurs a slowdown of 1.5x-3x [12]. Their performance is unsatisfactory and prohibits them from being adopted in production systems.

The major reason for the slowdown is that current dynamic taint trackers are all built on top of emulators using dynamic translation which itself is expensive. For instance, Argos incurs 15x-26x slowdown, most of which is caused by the emulator QEMU [8]. Minemu incurs 2.4x slowdown on SPECint 2006 on average, of which 1.4x is caused by dynamic translator [12], albeit designed specifically for taint tracking. In addition, emulators dynamically translate binary code which usually contain no symbol table, no debug information, and little structured information that are necessary for aggressive program analyses and optimizations needed to reduce the runtime overhead. In a typical compiler, the whole framework is designed to assist program analyses and optimizations. Many optimizations can be applied to dynamic taint tracking.

Solutions. Dynamic taint tracking implemented as IRMs provides the opportunity to greatly lower the heavy overhead. First, IRMs need no dynamic translation. Taint propagation operations are inlined within the subject programs, thus obviating dynamic translation. To separate program memory from taint tags, we can use software-based fault isolation to

ensure that subject programs do not tamper with taint tags. In addition, control flow integrity can be adopted to make sure that embedded taint propagation operations and SFI checks cannot be bypassed by adversaries. Our SFI and CFI implementation incurs low overhead compared to dynamic translation [42]. Second, IRMs implemented in a compiler enables aggressive optimizations that can further reduce the overhead. In a typical compiler, the abundance of structured information can be utilized to eliminate unnecessary taint propagation operations. As a simple example, an operation takes two objects, say A and B, and computes the result C. If it is statically proved that A and B cannot be tainted by user input, then there is no need to propagate taint to object C. On the other hand, if A or B is tainted, then C is tainted. We can taint C together with A or B in one operation. As another example, if an object in a loop is tainted, there is no need to taint it for every iteration. In other words, we can hoist it up or sink it down.

Also, the current implementations of dynamic taint tracking all target x86. The efficiency and practicality on ARM architecture, the ISA for the vast majority of mobile devices, are still unknown. We plan to retarget this framework to ARM architecture and evaluates its effectiveness and efficiency on ARM. The implementation is planned to be finished and evaluated by the end of August, 2013 and a submission to a related top conference will be made in September 2013.

4.4 A Journal Paper on Optimized and Verified IRMs

After all the aforementioned work is completed, a journal paper will be written and submitted. The journal paper will combine all the work we have done centering on verified and optimized IRMs and include additional work. The paper will talk about the high-level design, the implementations, optimizations, and verifications. It will be submitted to a top journal like ACM Transactions on Information and System Security (TISSEC).

Traditional IRM implementations either target a single target machine, an individual source language, or a specific security policy. No general framework for various IRMs targeting different ISAs exists yet. As an instance, CFI, a special IRM to ensure control flow safety, only enforces control flow safety for a specific target machine such as x86. It is non-trivial to port it to another ISA such as ARM. As another example, Pittsfield isolates untrusted code from the other modules through embedding sandboxing instructions into subject programs. However, Pittsfield enforces only one policy for a specific target, 32-bit x86. No general framework for all sorts of IRMs exist. This paper will introduce a general, retargetable framework, built on top of the industrial-strength compiler LLVM for various IRMs for different target machines. The framework will include all the IRMs we have enforced including CFI, and data sandboxing on LLVM IR. To further demonstrate the generality of the framework, we will implement another security policy such as fine-grained memory access control. Traditional sandboxing techniques allocate a big memory region for untrusted components. Stack, heap, global data, and text segments are all confined in the allowed memory region. However, sandboxing techniques do not ensure the memory safety for untrusted modules. To demonstrate the generality, we plan to implement a fine-grained memory access control policy. The policy ensures that memory writes only write to the designated memory buffers, thus ensuring memory safety for subject programs. The checks will be inlined on the LLVM IR level and control flow integrity will be ensured.

The journal paper will be submitted in November 2013.

4.5 Graduation Timeline

The graduation is planned in May 2014. Here is the timeline. By the end of August 2013, the solutions for improving the efficiency of taint tracking will be implemented. In September 2013, a submission will be made to a top conference. In September 2013, general exam will be finished, which is a necessary step towards graduation. In April 2014, dissertation will be defended. In May 2014, PhD degree will be awarded.

5 Conclusions

In this proposal, we studied prevalent software attacks and the existing vulnerability mitigations. Although buffer overflows have been around for more than 25 years and many ingenious mitigations have been invented, they still pose serious threats to software industry and users. Recently, buffer overflows have morphed into other forms and present new challenges to security research. We investigated the current vulnerability mitigations and studied their shortfalls. Few mitigations meet the aforementioned requirements for security defenses. For example, IRMs are promising defense techniques because they are powerful enough to enforce many policies, but they are not retargetable. Their runtime overhead is still too high for performance-critical systems and their trustworthiness is questionable without verifiers. We have proposed solutions to tackle the problems. First, traditional sandboxing implementations target only memory writes but not reads due to runtime performance considerations. We have combined CFI with data sandboxing and proposed optimizations to cut down the overhead to a new low level. Also, current IRMs are enforced through binary rewriting or other low-level transformations. We have proposed a framework to enforce IRMs at an intermediate representation level, i.e. LLVM IR, which enables language agnosticism, retargetability, and aggressive optimizations. Furthermore, current taint tracking is enforced either by source-level transformation or through binary rewriting. They are either language-dependent or machine-dependent, and incur heavy overhead. We proposed a new approach to do taint tracking on LLVM IR, thus enabling language-independency, retargetability and optimizations.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A theory of secure control flow. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 111–124. Springer, 2005.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity: principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

[4] Martín Abadi and Gordon D. Plotkin. On protection by layout randomization. *ACM Trans. Inf. Syst. Secur.*, 15(2):8:1–8:29, July 2012.

[5] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.

[6] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, LICS '01, pages 247–, Washington, DC, USA, 2001. IEEE Computer Society.

[7] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] Emery D. Berger and Benjamin G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM.

[10] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: an efficient approach to combat a board range of memory error exploits. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[11] Hristo Bojinov, Dan Boneh, Rich Cannings, and Iliyan Malchev. Address space randomization for mobile devices. In *Proceedings of the fourth ACM conference on Wireless network security*, WiSec '11, pages 127–138, New York, NY, USA, 2011. ACM.

[12] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: the world's fastest taint tracker. In *Proceedings of the 14th international conference on Recent Advances in Intrusion Detection*, RAID'11, pages 1–20, Berlin, Heidelberg, 2011. Springer-Verlag.

[13] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.

[14] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, pages 559–572, New York, NY, USA, 2010. ACM.

[15] Dinakar Dhurjati and Vikram S. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*, pages 162–171, 2006.

[16] Mark Dowd. Application-specific attacks: Leveraging the actionscript virtual machine, 2008.

[17] Tyler Durden. Bypassing pax aslr protection, 2002.

[18] Ulfar Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[19] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[20] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, SP '00, pages 246–, Washington, DC, USA, 2000. IEEE Computer Society.

[21] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms*, NSPW '99, pages 87–95, New York, NY, USA, 2000. ACM.

[22] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[23] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, ACSAC '06, pages 339–348, Washington, DC, USA, 2006. IEEE Computer Society.

[24] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[25] microsoft. A detailed description of the data execution prevention (dep) feature in windows xp service pack 2, windows xp tablet pc edition 2005, and windows server 2003, September 2006.

[26] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. Talx86: A realistic typed assembly language. In *In Second Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, May 1999.

[28] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 106–119, New York, NY, USA, 1997. ACM.

[29] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, OSDI '96, pages 229–243, New York, NY, USA, 1996. ACM.

[30] J. Pincus and B. Baker. Beyond stack smashing: recent advances in exploiting buffer overruns. *Security Privacy, IEEE*, 2(4):20–27, July-Aug.

[31] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, April 2006.

[32] Mattew J. Schwartz. Microsoft hacked: Joins apple, facebook, twitter. http://www.informationweek.com/security/attacks/microsoft-hacked-joins-apple-facebook-tw/240149323, 2013.

[33] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, pages 1–1, Berkeley, CA, USA, 2010. USENIX Association.

[34] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[35] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[36] Christopher Small. A tool for constructing safe extensible c++ systems. In *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*, COOTS'97, pages 13–13, Berkeley, CA, USA, 1997. USENIX Association.

[37] Christopher Small and Margo Seltzer. A comparison of os extension technologies. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 4–4, Berkeley, CA, USA, 1996. USENIX Association.

[38] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[39] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 380–395, Washington, DC, USA, 2010. IEEE Computer Society.

[40] Ollie Whitehouse. *An Analysis of Address Space Layout Randomization on Windows Vistaâ Ďć*, volume Symantec. Microsoft Press, February 2007.

[41] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Orm, Shiki Okasaka, Neha Narula, Nicholas Fullagar, and Google Inc. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy*, 2009.

[42] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security*. ACM, October 2011.