Mauro Conti, Stephen Crane, Tommaso Frassetto, Andrei Homescu, Georg Koppen, Per Larsen, Christopher Liebchen, Mike Perry, and Ahmad-Reza Sadeghi

Selfrando: Securing the Tor Browser against De-anonymization Exploits

Abstract: Tor is a well-known anonymous communication system used by millions of users, including journalists and civil rights activists all over the world. The Tor Browser gives non-technical users an easy way to access the Tor Network. However, many government organizations are actively trying to compromise Tor not only in regions with repressive regimes but also in the free world, as the recent FBI incidents clearly demonstrate. Exploiting software vulnerabilities in general, and browser vulnerabilities in particular, constitutes a clear and present threat to the Tor software. The Tor Browser shares a large part of its attack surface with the Firefox browser. Therefore, Firefox vulnerabilities (even patched ones) are highly valuable to attackers trying to monitor users of the Tor Browser.

In this paper, we present selfrando—an enhanced and practical load-time randomization technique for the Tor Browser that defends against exploits, such as the one FBI allegedly used against Tor users. Our solution significantly improves security over standard address space layout randomization (ASLR) techniques currently used by Firefox and other mainstream browsers. Moreover, we collaborated closely with the Tor Project to ensure that selfrando is fully compatible with AddressSanitizer (ASan), a compiler feature to detect memory corruption. ASan is used in a hardened version of Tor Browser for test purposes. The Tor Project decided to include our solution in the hardened releases of the Tor Browser, which is currently undergoing field testing.

Keywords: De-anonymization exploits, code-randomization, privacy-oriented software, Tor Browser.

DOI 10.1515/popets-2016-0050
Received 2016-02-29; revised 2016-06-02; accepted 2016-06-02.

Mauro Conti: Università degli Studi di Padova, E-mail: conti@math.unipd.it
Stephen Crane: Immunant, Inc., E-mail: sjc@immunant.com
Tommaso Frassetto: CASED/Technische Universität Darmstadt, Germany, E-mail: tommaso.frassetto@trust.cased.de
Andrei Homescu: Immunant, Inc., E-mail: ah@immunant.com
Georg Koppen: The Tor Project, E-mail: gk@torproject.org
Per Larsen: Immunant, Inc., E-mail: perl@immunant.com

1 Introduction

The Tor Project provides a suite of free software and a worldwide network designed to facilitate anonymous information exchange and to prevent surveillance and fingerprinting of these interactions. The Tor network is open to anyone and widely used by civil rights activists, whistleblowers, journalists, citizens of oppressive regimes, etc. Many sensitive websites, including the late Silk Road black market, are only accessible over Tor. Consequently, the Tor Network is continually facing de-anonymization attacks by law enforcement, intelligence agencies, and foreign nation states. A de-anonymization attack aims to disclose information, such as the identity or the location, of an anonymous user. While many de-anonymization attacks rely on weaknesses in the network protocol, they often require that adversaries control a large number of Tor nodes [26] or only work in a lab environment [39].

An alternative and practical way to de-anonymize Tor users is to exploit security vulnerabilities in the software used to access the Tor network. The most common way to access Tor is via the Tor Browser (TB) [73], which includes a pre-configured Tor client. Since TB is based on Mozilla's Firefox browser, they share a large part of their attack surfaces. In 2013, the Federal Bureau of Investigation (FBI) exploited a known software vulnerability in Firefox [71] to de-anonymize Tor users that had not updated to the most recent version of TB [27, 57, 74]. Due to the success of this operation, exploit brokers [79] (and, presumably, governments and criminals) are currently soliciting exploits for the TB. In early 2016, it was confirmed that the FBI continues to monitor the Tor network, this time using a de-

Christopher Liebchen: CASED/Technische Universität Darmstadt, Germany, E-mail: christopher.liebchen@trust.cased.de
Mike Perry: The Tor Project, E-mail: mikeperry@torproject.org
Ahmad-Reza Sadeghi: CASED/Technische Universität Darmstadt, Germany, E-mail: ahmad.sadeghi@trust.cased.de

anonymization attack devised by Carnegie Mellon University researchers [19].

The Open Technology Fund commissioned a study on current and future hardening efforts to reduce the attack surface of the TB [58]. One of the recommendations was to use compiler techniques to detect memory corruption (buffer overflow, use-after-free, uninitialized variables, etc.) such as the AddressSanitizer (ASan) feature [61]. Another key recommendation was to use address space layout randomization (ASLR) to prevent exploitation of memory corruption vulnerabilities. While ASan imposes a high runtime overhead [61], ASLR is very efficient. However, ASLR was recommended because it is widely supported by compilers and operating systems, not for its security properties. In fact, the shortcomings of ASLR are well documented in the academic literature [8, 16, 33, 62, 64, 68]. ASLR can be made significantly stronger by randomizing not just the base address of modules but also the code inside each module. Address space layout permutation (ASLP) [44], for instance, randomizes the location of each function individually, thwarting many of the techniques used to bypass ASLR. Until now, however, the ASLR improvements suggested in the literature have suffered from one or more drawbacks that have prevented their use in practice. Some techniques rely on binary rewriting, which does not scale to complex programs such as web browsers [22, 38]; others randomize the code using a customized compiler [35], or require each user to download their own unique binary [42].

Goals and Contributions The goal of this paper is to demonstrate a load-time randomization technique—named selfrando—that improves security over ASLR while preserving the features that enabled ASLR's widespread adoption. While technically challenging, our use of load-time function layout permutation ensures that the attack surface changes from one run to another. Load-time randomization also ensures compatibility with code signing and distribution mechanisms that use caching to efficiently serve millions of users. Finally, we worked in close collaboration with the TB developers to ensure that selfrando was fully compatible with ASan so that users can use both techniques simultaneously. ASan is used in a hardened version of TB to detect and diagnose memory corruption errors.

Summing up, our main contributions are:

- **Practical Randomization Framework** Unlike other solutions that have only been tested on benchmarks, selfrando can be applied to the TB without any changes to the source code. To the best of our knowledge, selfrando is the first approach that avoids risky binary rewriting or the need to use a custom compiler, and instead works with existing build tools. Moreover, it is fully compatible with ASan, which required additional implementation effort since the randomization interferes with ASan.
- **Increased Entropy and Leakage Resilience** selfrando reduces the impact of information leakage vulnerabilities and increases entropy relative to ASLR, making selfrando more effective against guessing attacks. Our use of load-time randomization mitigates threats from attackers observing binaries during download or after the executable files have been stored on disk.
- **Hardening the Tor Browser** We demonstrate the practicality of selfrando by applying it to the entire TB without requiring any code changes. Our detailed and careful evaluation shows that the startup and performance overheads of selfrando are negligible.

2 Background

2.1 Exploiting Memory Corruption

Unlike modern programming languages, C and C++ rely on manual memory management, trading reliability for flexibility and performance. Hence, memory management errors often create vulnerabilities that can be exploited to hijack control flow and perform other malicious operations that were never intended by the program authors.

Traditionally, attackers used a buffer overflow to directly inject malicious code into a program and execute it [6]. However, the introduction of the W⊕X policy that requires memory pages to either be writable or executable, but not both, made most code-injection attacks [49] obsolete. As W⊕X became commonplace, attackers changed their tactics from code injection to code reuse. These attacks reuse existing, legitimate code for malicious purposes and have therefore proven far harder to stop than code injection. Return-into-`libc` (RILC) attacks, for example, arrange the stack contents so the attacker can call dangerous functions inside the C library with attacker-controlled arguments [52]. Researchers later demonstrated a more general class of code reuse attacks called return-oriented programming (ROP) [63]. The insight behind ROP is that attackers can build a malicious virtual machine out of short instruction sequences—called *gadgets* in ROP parlance—

ending with a return (or some other indirect branch). These gadgets are all located inside application code, so the attacker has no need to inject them into the program. Over the last decade researchers have discovered many variants of code-reuse attacks [10, 11, 15, 59, 76], most of which are not stopped by ASLR, W⊕X, or other widely deployed exploit mitigations.

2.2 Preventing Code-Reuse Exploits

To successfully mount a code-reuse attack, several requirements must be met. First, the application must contain a memory corruption vulnerability that allows control flow to be hijacked. Techniques such as control-flow integrity and stack canaries make control-flow hijacking harder but do not prevent it outright [14, 29, 36, 48]. Another key requirement is knowledge of the absolute addresses of the gadgets to reuse. In principle, ASLR [54] prevents adversaries from knowing the absolute locations of ROP gadgets. However, since ASLR only randomizes the base address of a library, adversaries still know the relative positions of all functions inside a library. Using this knowledge together with a leaked code pointer, attackers can compute the absolute addresses of all functions in the same library. Academics have documented numerous ways to leak code or pointers to code [24, 60, 62, 66]. Permuting the functions inside a library removes attackers' knowledge of the relative function layout inside each library, and additionally improves entropy by allowing an exponentially higher number of code layouts in comparison to ASLR [44].

Numerous other fine-grained diversity techniques have been suggested in the literature. In this paper, we focus on function permutation since it is practical and efficient, as shown by existing diversity surveys [17, 46].

Unfortunately, previous fine-grained diversity approaches have been unable to replace ASLR because they have one or more of the following drawbacks:

1. they introduce unacceptable performance overheads [69],
2. they rely on unsafe binary rewriting techniques that do not scale to complex, real-world applications,
3. they randomize code at compile time which is incompatible with current distribution mechanisms that are optimized to deliver a single binary.

In contrast, the technique we present in this paper, selfrando, avoids all of these drawbacks and scales to real-world applications including Firefox and TB.

2.3 Trust in Privacy-preserving Software

As we have previously mentioned, any tactic that allows de-anonymization of Tor network users is likely to be attempted by law enforcement, intelligence agencies, and other resourceful adversaries. The ability to surreptitiously insert backdoors into the TB would be a particularly powerful attack. In order to reduce the likelihood that backdooring attempts would go unnoticed, the Tor developers ensure that builds are reproducible. Even though the TB source code can be downloaded by anyone, differences in build tools, libraries, file system layout and even system time make it hard to simply build the TB from sources and compare it to the official binaries to ensure the absence of backdoors. Therefore, the TB is built using Gitian, a special tool which provides a reproducible build environment [55, 72]. This allows third parties to independently compile and verify the binaries distributed by the Tor Project and detect any signs of external compromise.

Gitian consists of a virtual machine and a number of build scripts to automate the process. The virtual machine insulates the build from the outside environment. At the time selfrando was developed, the TB builds for Linux used a virtual machine based on Ubuntu 10.04. Hence, many build tools were unavailable or outdated. To cope with this shortcoming, we either compiled a recent version of the tool in the virtual machine itself or we adapted the build process to support the older version. The Tor developers recently (May 2016) switched to a virtual machine based on Debian 7. During the switch no modifications were necessary to our code.

3 Selfrando

3.1 Design Objectives

Our main objective is to substantially raise the costs for attackers to exploit memory-corruption vulnerabilities. For practicality reasons, we choose to support complex C/C++ programs (e.g., a browser) without modifying their source code. Further, we retain full compatibility with current build systems, i.e., we should avoid any modification to compilers, linkers, and other operating system components. To be applicable to privacy-preserving open-source tools, we must not rely on any third-party proprietary software. Finally, our solution should not substantially increase the size of the program in memory or on disk.

3.2 Threat Model

We make standard assumptions from underlying real-world adversary settings: we assume that a remote attacker triggers a memory corruption vulnerability to hijack control flow and achieve remote code execution. Due to the widespread deployment of stack protections (e.g., StackGuard [18] and SafeStack [70]) and the fact that most exploits against browsers rely on use-after-free errors [75], we assume that the adversary exploits a heap-based memory corruption vulnerability. This means that the adversary can use code pointers stored on the heap to disclose the location of code before mounting a code-reuse attack. Further, we assume that a W⊕X policy is in place to prevent code injection, which is true for all modern systems. In this work we do not consider attacks that target the browser's JIT engine.

Note that our threat model does not cover some theoretical attacks such as JIT-ROP [66] and COOP [59] that have only been demonstrated in an academic setting. As mentioned above, our main objective is high practicality while significantly improving security provided by ASLR against memory corruption attacks; defenses that can stop JIT-ROP and COOP are less efficient and rely on special hardware support, a custom compiler, and a patched OS kernel [12, 20, 21].

3.3 Selfrando Design

Existing exploit mitigations such as W⊕X and ASLR already make de-anonymization exploits costly to develop. Thus, exploits which bypass these mitigations often target high-profile applications with many users. Although the Tor user base isn't large, the TB shares a large amount of code with Firefox which has hundreds of millions of users and contains more than 20 million lines of code. The similarities between the TB and Firefox make it comparatively easy to re-purpose mainstream Firefox exploits to de-anonymize Tor users. We can use our improved randomization mechanism to protect the TB and at the same time strongly raise the bar for the adversary to port exploits from Firefox to TB.

The easiest way to perform fine-grained code randomization is by customizing the compiler to take a seed value and generate a randomized binary [32, 42]. Unfortunately, compiling and distributing a unique binary for each is impractical for introducing diversity among a population of programs [30, 78]. With more implementation effort, we can delay randomization until load-time, which has several benefits. Most importantly, software vendors only need to compile and test a single binary. A single binary also means that users can continue to use hashes to verify the authenticity of the downloaded binary. Finally, modern content delivery networks rely extensively on caching binaries on servers; this optimization is no longer possible with unique binaries.

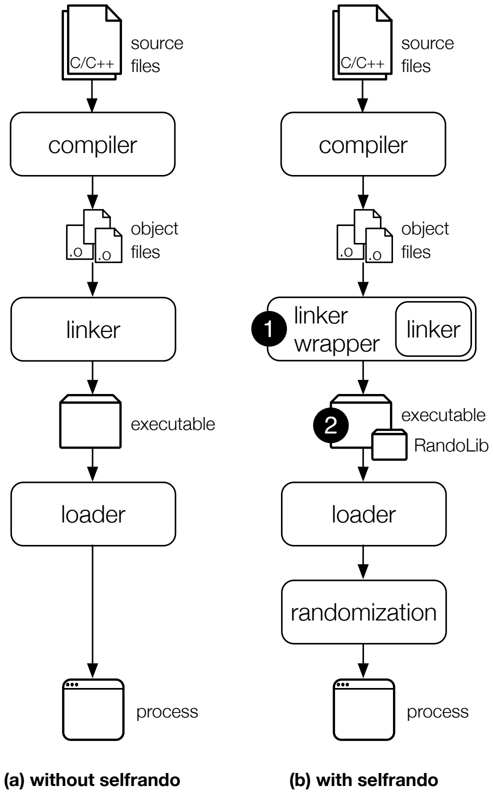

Fig. 1. Building and running applications without (a) and with selfrando (b) enabled.

In the context of privacy-preserving software such as TB, compile-time randomization raises additional challenges. Randomized builds would complicate the deterministic build process,[1] which is important to increase trust in the distributed binary (see Section 2.3). Moreover, compile-time randomization would (a) increase the feasibility of a de-anonymization attack due to individual, observable characteristics of a particular build, and (b) allow an attacker to build knowledge of the mem-

[1] A randomized build can be implemented in a deterministic environment by passing a random seed as an input to the deterministic process. The builds would then be distributed along with their seed. A user could then check the integrity of her build by running the deterministic process again with the same seed. However, that check would not prove the integrity of builds with other seeds.

ory layout across application restarts, since the layout would be fixed.

For these reasons, we decided to develop a framework which makes the program binary randomize itself at load time. We chose function permutation (ASLP) as the randomization granularity, since it dramatically increases[2] the entropy of the code layout while imposing the same low overheads as ASLR [44]. Since discovering function boundaries at load-time by analyzing the program binary is unreliable and does not scale to large programs, we pre-compute these boundaries statically and store the necessary information in each binary. We call this *Translation and Protection* (TRaP) information.

Rather than modifying the compiler or linker, we developed a small tool which wraps the system linker, extracts all function boundaries from the object files used to build the binary, then appends the necessary TRaP information to the binary itself. Our linker wrapper works with the standard compiler toolchains on Linux and Windows and only requires a few changes to the build scripts to use with the TB.

Figure 1a represents the usual workflow from the C/C++ source code to a running program. Figure 1b represents the modified workflow with selfrando. A *linker wrapper* intercepts calls to the linker and calls selfrando to gather information on the executable file ❶. Then, it embeds TRaP information and a load-time randomization library, RandoLib, into the binary file ❷. When the loader loads the application, it will invoke RandoLib instead of the entry point of the application. RandoLib will randomize the order of the functions in memory and then transfer control to the original program entry point.

4 Implementation

One of our main goals is to demonstrate the practicality of selfrando by integrating it into the TB. To test selfrando before it is released to Tor users at large, the Tor project decided to first include our defense in a series of experimental, hardened builds for Linux.[3] The hardened builds of Tor include additional features such as AddressSanitizer (ASan), a compiler feature which can detect memory corruption. ASan and selfrando are complementary in nature. The former detects bugs that can create security issues, however, ASan is not a defense mechanism like selfrando and should not be relied upon to stop exploits [51].

To build a program with selfrando, the build scripts must be updated to use our linker wrapper rather than directly invoking the system linker. The wrapper accepts the same arguments as the system linker, so modifying the build scripts is a straightforward task for a skilled software developer. This enables us to intercept any invocation of the linker and modify its arguments. In the following we will explain the major implementation aspects with the help of Figure 2. Notably, we will explain in detail how selfrando (1) extracts the metadata needed to create self-randomizing binaries, (2) embeds the extracted information and the load-time component into the generated binary, and (3) permutes all functions during load time without breaking the application.

Finally, we describe two practical challenges that we solved to make selfrando compatible with the hardened build of TB. Specifically, we needed selfrando to (4) support stack unwinding which is needed for stack traces and exception handling and (5) be compatible with ASan.

4.1 Extracting TRaP Information

When a module is loaded, selfrando permutes the order of all its functions. To do so, selfrando requires accurate information about function boundaries. If this information is not accurate, shuffling the function layout may inadvertently introduce errors that prevent correct execution of the application. After a function is moved, all references and pointers to this function, e.g., the target address of a call, become invalid because they still reference the old address. Hence, selfrando needs to update all references to the moved function, and therefore requires, for each function, a complete list of all locations that reference that function.

Such information is present in the intermediate object files ❶. Since this metadata is usually not required during execution, the linker strips it from the final binary. Our linker wrapper therefore intercepts the linking process to extract function boundaries and references and embeds this information for use at load time.

However, object files do not explicitly mark all function references. Specifically, we found that in some cases the compiler optimizes the code by inserting direct jumps between two functions. Such references are not

[2] We compare the entropy of function permutation and ASLR in Section 5.

[3] Selfrando is also compatible with Android and closed-source platforms such as Microsoft Windows.

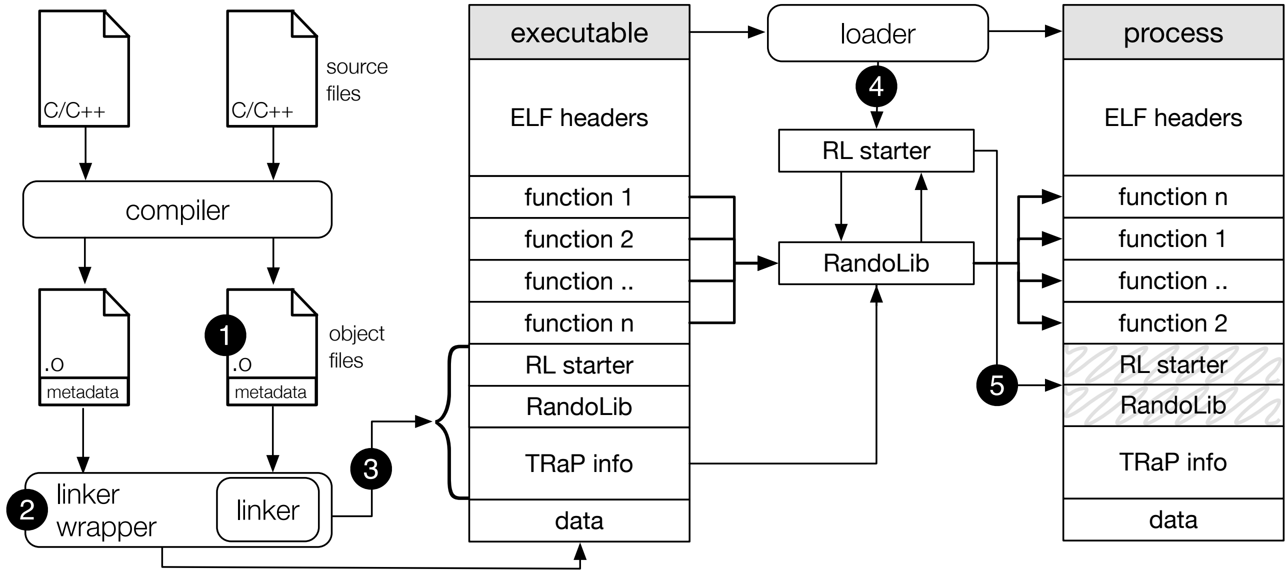

Fig. 2. Workflow of selfrando.

marked with an explicit relocation because they are already resolved by the compiler. Fortunately, we can disable this behavior with a compiler option causing the compiler to place each function in a separate section. Since the compiler marks all references between sections, we can then see all function references. While enabling this option slightly increases build-time (0.07%), it also enables a linker optimization which increases locality [31].

Pre-compiled language runtime object files are another obstacle. One example is `crtbegin.o` for GCC which contains functions to initialize the runtime environment for applications that were programmed in C. In our current implementation, we treat such object files as one single block because they contain only a few functions. This has a negligible impact on the overall randomization entropy. Nevertheless, we are currently investigating how we can generate selfrando-compatible versions of the pre-compiled object files.

After selfrando extracts the necessary metadata from each generated object, it adds an additional linker argument that instructs the linker to generate a *map file*, which is a text file that contains the memory layout of the final binary ❷. Using the metadata and the map file, selfrando can compute the final location of each function in the executable file and all references to these functions.

Next, we explain how we embed the TRaP information in the binary to make it available to the run-time component—RandoLib.

4.2 Embedding TRaP information

We include the TRaP info, which is used by RandoLib, in the executable to make selfrando self-contained. This avoids having to manage additional files, which could add logistical burden.

However, from a technical point of view, embedding the data in a space-efficient and binary format-compatible way without modifying the linker is challenging. The main reasons are that (1) some of the metadata is only available *after* linking is complete, and (2) we cannot pre-allocate space for the data since the exact amount of space needed is unknown until linking is done. In particular, the start address of each function in the linked binary is determined by the order and final addresses of the object files in the binary, and therefore unknown until all objects are linked.

To add additional data to the final binary, we have to resort to a trick that involves changing the linker input so that it adds an empty segment header in the beginning of the binary. Note that a linked ELF binary is divided into segments. The linker creates a segment header which contains segment metadata, e.g., size and memory permissions, for each segment. The loader uses this metadata to load each segment of the binary into memory. Due to the structure of the binary format, adding an empty segment header in the beginning of the binary enables selfrando to append an arbitrary amount of data. When the linker is finished, we append the TRaP info and RandoLib to the end of the binary and set the values of the empty segment header accordingly ❸. Finally, we change the start address of the binary—its *entry point*—to RandoLib. Hence, after the loader loads the binary into memory, it will transfer control to RandoLib, which then performs the function permutation.

4.3 Load-time Function Permutation

RandoLib performs function permutation using the embedded TRaP info, and consists of two parts: a small helper stub and the main randomization module. The purpose of the helper stub (*RL Starter* in Figure 2) is to make all selfrando data inaccessible after RandoLib finishes. The operating system loader ❹ calls this stub, invoking RandoLib as the first step of program execution.

The function permutation algorithm proceeds in several steps. First, RandoLib generates a random order for the functions using the Fisher-Yates shuffling algorithm. Second, RandoLib uses the embedded metadata to fix all references that became invalid during the randomization. Finally, after RandoLib returns, the helper stub makes selfrando's data inaccessible ❺, and jumps to the original entry point of the binary.

While this approach might seem straightforward, we faced several technical challenges. For example, we have to consider that C++ code and certain assembly instructions require a certain alignment for every function. The Itanium C++ method pointer specification assumes that all functions are at least 2-byte aligned [43]. Further, we found that some assembly instructions are sensitive to alignment, e.g., `movdqa` which is commonly used in the implementation of cryptographic functions. We account for the alignment of C++ functions by increasing the size of the code segment by one byte per function. This allows RandoLib to maintain the least significant bit alignment of functions during copying. During our evaluation, we found that this alignment increases the file size on average by 0.3%.

Our implementation does not fully support alignment-sensitive assembly instructions, as they are not used by the TB. We can currently run programs that use such instructions by preserving the four least significant bits of function addresses during randomization. Moreover, we are working on a static analysis tool that can identify functions that contain these instructions, and mark them in the TRaP info so RandoLib can take their alignment constrains into account.

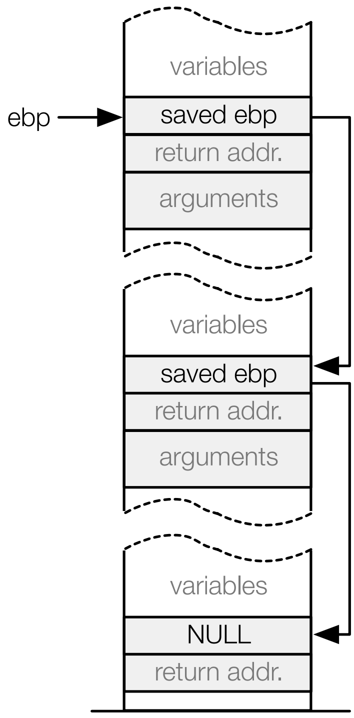

Fig. 3. Stack layout with the frame pointers.

4.4 Stack Unwinding

During program execution, the program stack is divided into *stack frames*. Each stack frame corresponds to a function call and consists of local variables, the return address, and arguments which were passed to the callee. *Stack unwinding* is the process of iterating through all active stack frames, starting from the most recent. It is mainly used for stack traces and exception handling, as both require access to previous stack frames. Exception handling uses stack unwinding to find the exception handler for a given exception after the program has thrown an instance of that exception.

Traditionally, stack unwinding is supported by chaining stack frames as a singly-linked list, where each stack frame includes a pointer to the previous stack frame. The head of the linked list is stored in a dedicated register called the *base pointer* (BP) (`ebp` on x86). When a new stack frame is added, the called function saves the BP register of the caller on the stack, then overwrites the BP register to point to the current stack frame, as shown in Figure 3.

Modern compilers omit the frame pointer for optimized code to reduce memory usage on the stack and free another register for general purpose computations. To still support stack unwinding, compilers generate additional metadata which can be used to identify individual stack frames. Function permutation invalidates function references inside the stack unwinding metadata, so RandoLib updates them.

4.5 AddressSanitizer

The TB developers use AddressSanitizer (ASan) [61] to detect memory corruption bugs in their hardened releases. To allow selfrando to be deployed on TB, selfrando needs to work correctly with ASan.

In general, selfrando does not interfere with the normal operation of ASan. When ASan detects a memory corruption, it generates a stack trace, which is supported by selfrando (cf. Section 4.4). To help troubleshoot memory corruption bugs, ASan annotates the stack trace with symbolic information. Specifically, it uses a *symbolizer* to obtain the function name and the source code location of every address in the stack trace. After selfrando randomizes the order of functions, the symbolizer can no longer correctly map the stack addresses to function names. We restore the symbolizer's ability to annotate stack traces by emitting a map file that stores the original and actual address of each randomized function. We modify the symbolizer of ASan to use the emitted mapping to map the addresses of the stack trace to the original address.

While storing the randomization map on disk is a potential security risk, exfiltrating this map would require that the attacker can read the randomization map file. The ability to read arbitrary files gives the attacker other, more significant advantages. For example, an attacker could use this advantage to disclose the full memory layout of the program by reading the special `/proc/self/mem` file.

5 Experimental Evaluation

We thoroughly evaluated selfrando from a security, performance and compatibility standpoint.

5.1 Security Analysis

We first evaluate the security of our solution and ASLR in terms of randomization entropy. This shows how well each defense resists brute force attacks. We then use a real-world exploit to compare our solution to ASLR in cases where attackers exploit information leakage which can be more effective than brute force guessing.

Randomization Entropy

For any randomization scheme the amount of entropy provided is critical, because a low randomization entropy enables an attacker to guess the randomization secret with high probability [64]. We compare selfrando to ASLR—the standard code randomization technique that is available on all modern systems.

We determined the real-world entropy of ASLR by running a simple position-independent program multiple times and analyzing the addresses, on a Debian 8.4 machine using GCC 6.1.0 and Clang 3.5.0. ASLR provides up to 9 bits of entropy on 32 bit systems and up to 29 bits of entropy on 64 bit systems. While the ASLR offset on 32 bit systems is guessable in a reasonable amount of time, such attacks become infeasible on 64 bit systems because the address space is that much larger. However, an attacker can bypass ASLR by leaking the offset that the code is loaded at in memory through a pointer into application memory. Once this offset is known the attacker can infer any address within the application, because it is used to shift the address of the whole application.

Selfrando, on the other hand, applies more fine-grained function permutation. This means the randomization entropy does not depend on the size of the address space, as it is the case for ASLR, but on the number of functions in the randomized binary. The total entropy generated by selfrando on a library containing m functions depends on the factorial of m:

$$E_t = \log_2(m!)$$

On the other hand, the attacker does not usually need to disclose the whole layout; the addresses of a few functions are enough. Assuming the attacker already bypassed ASLR, the attacker needs to disclose the least significant bits of each pointer. The entropy of a pointer to a randomized function depends on the size of the executable section s:

$$E_p = \log_2(s) - 1$$

We need to subtract 1 because the least significant bit of the addresses is preserved during the randomization. Assuming that the attacker needs gadgets in n different functions, the total number of bits the attacker needs to disclose is the minimum of E_t and n times E_p:

$$E = \min(E_t, n \times E_p)$$

In practice, E_t is much greater than E_p due to the factorial, so we can assume $E = n \times E_p$.

Technique	Entropy
ASLR (32 bit)	9 bits
ASLR (64 bit)	29 bits
Selfrando (10 KB code)	$13 \times n$ bits
Selfrando (163 KB code)	$17 \times n$ bits
Selfrando (6.5 MB code)	$22 \times n$ bits
Selfrando (92 MB code)	$26 \times n$ bits

Table 1. Randomization entropy of ASLR and selfrando for different address space sizes and function counts. For selfrando, we report the number of bits the attacker needs to guess for each function address the attacker needs.

Using TB as our model organism, we use the number of functions to calculate the minimum and maximum entropy for a binary protected by selfrando. The smallest library (`libplds4.so`) has 44 functions in 10 KB of code, while the biggest (`libxul.so`) has 242 873 functions in 92 MB. The median is 494 functions in 163 KB, while the average is 16 814 functions in 6.5 MB. Table 1 shows that for each function address, the attacker needs to guess between 13 and 26 bits. If we assume that the attacker needs the address of at least three functions, selfrando is significantly more effective than ASLR. For the smallest library, the attacker needs to guess at least 39 bits, while for the biggest, the attacker needs at least 78 bits.

Additionally, selfrando provides higher leakage resilience compared to ASLR because the attacker no longer knows the relative function layout inside each binary.

Real-world Exploits against the Tor Browser

One of our main objectives is to enhance the resilience of TB against code-reuse attacks. Previously conducted attacks, e.g., by the FBI [57], fail because these attacks do not consider selfrando (see Appendix A for an overview of the exploit the FBI used). Therefore, we analyze the attack surface of TB after selfrando was applied in a realistic attack scenario. We base our analysis on four observations we made while studying real-world exploits.

First, nearly all modern attacks exploit heap-based vulnerabilities, despite the existence of stack vulnerabilities [50]. However, whether a vulnerability can be exploited to launch a code-reuse attack depends on different factors, like how reliably the vulnerability can be triggered and the present mitigation techniques. Today, most stack-based vulnerabilities are not exploitable because they are mitigated by modern stack defenses [18, 70].

Second, information disclosure attacks are often limited to leaking heap memory because they access memory relative to the address of the vulnerable memory object. A buffer overread, for example, can be exploited to disclose consecutive memory which might contain interesting pointers, whereas a use-after-free vulnerability can be exploited to disclose interesting pointers of the freed object. In both cases the attacker is not able to (repeatedly) disclose absolute, and therefore, arbitrary, addresses. For these reasons we assume that in a practical scenario the attacker cannot leak information that is not located on the heap, e.g., stack or code pages. To overcome this limitation attackers use a technique, called *heap feng shui* [67], to place an object that contains valuable pointers near to the vulnerable object.

Third, most real-world attacks are based on ROP. While other types of code-reuse attacks exist [15, 52, 59], ROP remains the most versatile technique. To execute a ROP payload, the attacker needs to either inject his payload directly on the stack, or use a *stack-pivot* gadget to overwrite the stack pointer with an address that points to the ROP payload on the heap. As mentioned previously, the attacker usually has no access to the stack. Hence, the first gadget in the ROP chain is normally a stack-pivot.

Fourth, ROP is merely used to bypass W⊕X policies and enable code injection, i.e., a small ROP payload is used to (1) mark the data memory containing the shellcode as executable and (2) branch to the shellcode. The shellcode will then perform the actual task of de-anonymizing the user or installing surveillance software. To mark a data page as executable, only a single system call is needed. Hence, the attacker requires only gadgets that load the arguments for the system call into the registers, then issue a system call and return to the shellcode.

Based on these four observations, we examined the main TB library with selfrando enabled (`libxul.so` having a size of 92MB) to find out whether an attacker is able to disclose the address of a stack-pivot and a system call gadget based on addresses that can be found on the heap. We focus on stack-pivot and system call gadgets because they are less common, and therefore, harder to disclose compared to gadgets that only load a value into a register. In total, we found ten stack-pivot and 76 system call gadgets of which only 4 and 29 respectively are available through virtual functions whose addresses are exposed on the heap through indirection tables called *virtual tables*.

We manually analyzed each function and concluded that no pointer to these functions is ever written on

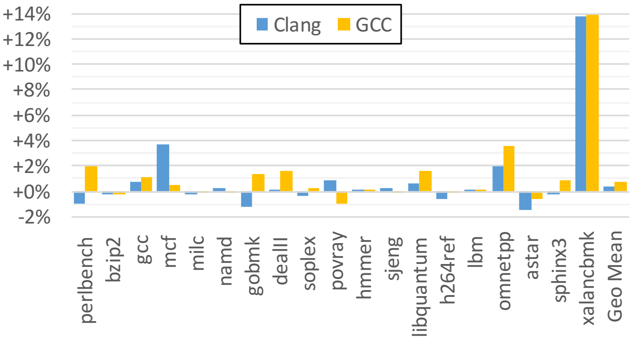

Fig. 4. Run time overhead on the benchmarks in the SPEC CPU2006 suite (full selfrando).

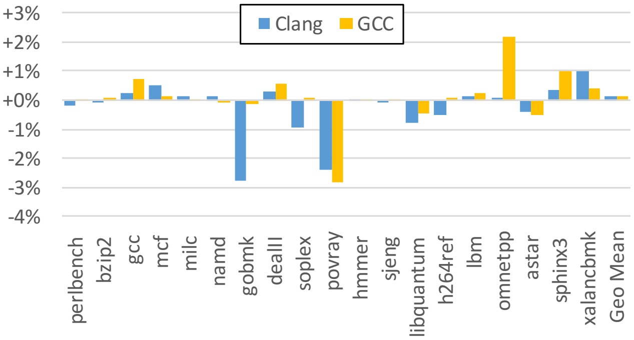

Fig. 5. Run time overhead on the benchmarks in the SPEC CPU2006 suite (identity transformation).

the heap. The reason is that these function pointers are only accessed through an indirection layer, i.e., memory objects on the heap contain a pointer to a virtual table which is located in the code or data section of the application and contains a number of pointers to virtual functions. Since the attackers can only disclose the virtual table pointer, but not the virtual table itself, as it is not on the heap, they cannot disclose gadget addresses. Note that, when only ASLR is applied, the address of the virtual table is randomized with the same offset as the ROP gadgets. Therefore, such an attack can bypass ASLR but not selfrando.

We therefore conclude that selfrando can thwart most real-world exploits. Attackers can only succeed in rare cases where they can disclose the complete heap and data section.

5.2 Performance Overhead

We performed multiple tests to measure selfrando's run-time overhead. Since selfrando works at load-time, we also measured the additional startup time.

All tests were performed on a system with an Intel Core i7-2600 CPU clocked at 3.40 GHz, with 12 GB of RAM and a 7200 RPM hard disk. We used version 5.0.3 of the Tor Browser on Ubuntu 14.04.3.

5.2.1 Load-time Overhead

We measured the load time of TB by inserting a `return` statement in the `main` function, after the dynamic libraries are loaded but before the program actually does anything. We invoked the modified program and measured the load time using the standard tool `time`. As a baseline, we used the source code of TB 5.0.3, unmodified except for the `main` function. For both versions, the reported time is the average of 10 runs. We cleaned the disk cache before each run, so the binary was loaded from the disk every time.

The average load time for the normal version was 2.046 s, while the selfrando version took 2.400 s on average. The average overhead is 354 ms. We believe this is an acceptable overhead considering the improved protection against de-anonymization attacks.

5.2.2 Run-time Overhead

To test the run-time overhead of selfrando, we ran the SPEC CPU2006 benchmark suite as well as a number of modern JavaScript benchmarks.

We executed all the C and C++ benchmarks in SPEC CPU2006 with the two standard Linux compilers (GCC and Clang) with selfrando enabled. Moreover, we ran the benchmarks with a version of selfrando that always chooses the original order for the randomization (*identity transformation*). This version runs all the load-time code but it does not actually modify the code segment. It allows us to distinguish between load-time overhead and run-time overhead. We ran each benchmark three times with the `ref` workload. The reported figures are the median values.

Figure 4 shows the performance overhead on each benchmark. The geometric mean of the positive overheads is 0.71% for GCC and 0.37% for Clang. The overhead of each benchmark except for `xalancbmk` is below 4%. We found `xalancbmk` to be an outlier, with an overhead of about 14%. We investigated this issue using the Linux performance analysis tool, `perf`, comparing the full selfrando and the identity transformation runs. We discovered a 69% increase in L1 instruction cache misses and a 521% increase in instruction

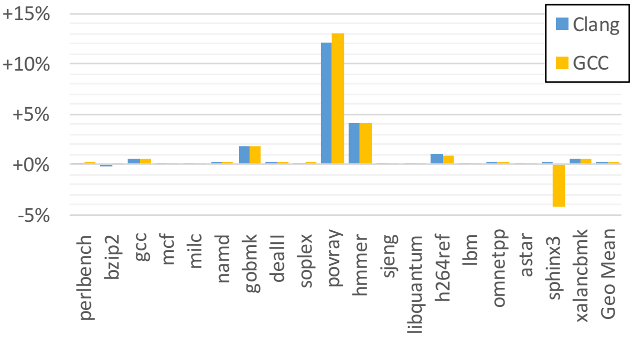

Fig. 6. Memory overhead of the benchmarks from the SPEC CPU2006 suite (full selfrando).

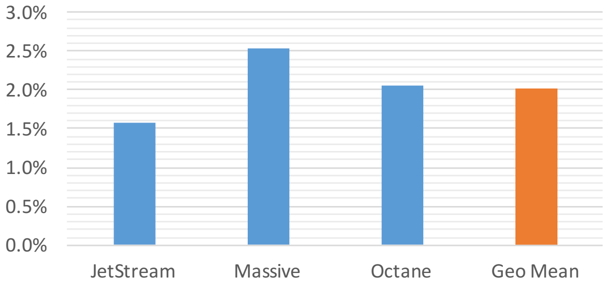

Fig. 7. JavaScript performance overhead of selfrando w.r.t. a version with all our modifications but without the actual randomization.

TLB (Translation Lookaside Buffer) misses. We believe that the xalancbmk benchmark is sensitive to the function layout and that some frequently executed functions must be co-located to ensure optimal performance. We didn't observe a high sensitivity to the function layout for any of the other benchmarks. A possible extension to selfrando to cope with location-sensitive programs is to automatically use performance profiling to identify groups of functions that should be moved as a single bundle similar to the work of Homescu et al. [41]. If these bundles are small enough, this extension would not significantly reduce the security of a large application (xalancbmck contains 13478 functions). Figure 5 shows the run time overhead with the identity transformation.

In some cases, selfrando actually improves performance. In particular, we observed that with the identity transformation the performance of gobmk and povray improves up to 2.5%. We suspect this is caused by the compiler flag that places each function in its own section, which enables further linker optimizations [31]. This flag is not enabled by default, but selfrando requires it (see Section 4.1).

Figure 6 shows the overhead on the memory usage of each benchmark. To measure the memory usage, we used the *maximum resident set size* reported by the time utility. The geometric mean of the positive overheads is 0.18% for GCC and 0.20% for Clang. We also measured the absolute overheads: the geometric mean of the positive values is 299 kB for GCC and 295 kB for Clang.

The memory overhead of all benchmarks except for povray and hmmer is below 2%. These benchmarks have higher relative overheads due to their small memory footprints, about 5 MB for povray and about 9 MB for hmmer. Their absolute overheads are about 600 kB and 400 kB respectively.

Finally, we evaluated selfrando with modern JavaScript benchmarks that focus on realistic web workloads: JetStream 1.1., Massive and Octane 2.0 [1–3]. As a baseline, we used a version of TB with the same modifications we need for selfrando (see Section 5.3), but without the randomization. Since selfrando does not protect JIT-compiled code, we disabled the JIT compiler by setting the *Tor Security Slider* to *Medium-High*. Figure 7 reports the results. Each benchmark produces a score (higher scores are better) and we report the relative *decrease* on the score. The geometric mean of the overheads is 2.02%, while the worst overhead is 2.5%.

Our measurements confirm that selfrando can be integrated in real-world applications with low overhead.

5.3 Compatibility

Selfrando was optimized to protect the TB which is built with GCC. However, we built several other Linux programs such as GNU Bash 4.3, GNU less 458, Nginx 1.8.0, Socat 1.7.3.0 and Thttpd 2.26. We tested each of them using application-specific workloads, such as serving files and running shell scripts, and we did not encounter any problem.

To demonstrate compatibility with other compilers we decided to build Chromium [9]. We chose Chromium because this project has a large and complex code base, and uses Clang [47] as default compiler. Like with TB, we had to resort to the libc heap allocator, as Chromium's default heap allocator relies heavily on Thread-Local Storage (TLS) and, hence, is not fully compatible with selfrando. However, after changing the heap allocator we successfully built and ran Chromium.

Both browsers implement cryptography using low-level code that embeds data in the code segment. This produces unexpected results when the data is moved along with the functions and the alignment is not preserved. For Firefox, we disabled the low-level implementation and we used the high-level one. For Chromium, there was no easy way to disable the alignment-sensitive code and we had to preserve the four least significant bits of the addresses during the randomization (see Section 4.3).

To ensure selfrando did not break any functionality we tested both browsers with popular websites[4] and we did not encounter any problems.

5.4 Including selfrando in the Tor Browser

The Tor Project is experimenting with a number of different tools to produce *hardened builds* of TB [56]. We worked closely with their developers in order to make it easy to integrate selfrando in TB. Selfrando was added to the *nightly hardened* builds released and May 13, 2016 or later [45]. They plan to release a hardened version that includes selfrando and to evaluate the inclusion of selfrando in the normal version.

6 Discussion

Privacy Implications

Load-time code randomization effectively creates a unique code layout for each TB session. Theoretically, an adversary with the ability to read memory can exploit this to create a unique fingerprint to identify the user across different websites.

However, we argue that modern Web technologies (like JavaScript) by themselves can be exploited to leak information to identify users across different websites. Moreover, even without selfrando, an attacker that can read the memory or leak some pointers can fingerprint a browsing session in a number of different ways. ASLR creates code diversity because the binary and the libraries are loaded at different addresses. ASLR also affects the allocation of dynamic data structures such as the heap, stack and data within the heap. The allocation of these data structures is highly dependent on the usage of the browser, and hence, it is very likely that the disclosure of heap addresses is already enough to identify users. Additionally, a potential fingerprint of the randomized code is only valid for one browsing session; after a browser restart, the code layout is randomized differently. Finally, selfrando is compatible with XoM [7, 12, 20, 34] which prevents reading memory that contains code in the first place.

Hence, our randomization scheme does not increase the risk of fingerprinting.

System libraries

While software protected by selfrando works smoothly with unprotected libraries (and protected libraries work smoothly with unprotected programs), the security guarantees provided by selfrando are obviously limited to software that was re-built with selfrando. The TB includes most needed libraries, and hence, is not affected by this.

Future Work

Our current implementation focuses on applying selfrando to the TB. We are currently working on improving operating system specific features, such as the support for thread-local storage (TLS). TLS is heavily used in Firefox's default heap allocator *jemalloc*, however, it is possible to build the TB using the default heap allocator provided by libc instead, which does not rely on TLS. In fact, the TB developers expressed their desire to use a different allocator as well [56].

7 Related work

Run-time defenses usually rely on either memory randomization or integrity checks to prevent vulnerability exploitation.

7.1 Randomization-based defenses

We refer to the SoK paper by Larsen et al. [46] for a thorough analysis of the proposed software diversity tools and limit our discussion to recent works which are relevant to our purposes.

XIFER by Davi et al. [22] is a load-time fine-grained randomization tool that does not require access to the source code or offline analyses. However, its processing speed (< 0.7 MB/s) makes it unsuitable for complex applications that need to be loaded quickly.

Giuffrida et al. [35] proposed a compiler-based periodic re-randomization strategy for microkernels; this

[4] To get a representative set, we selected the Alexa Top 100 sites (http://www.alexa.com/topsites) of November 2015.

strategy would require end users to compile the TB locally on their system which is impractical for users with low end systems and would significantly increase the download size of the TB. Homescu et al. [42] built a compile-time randomization approach that scales to large applications such as the TB but requires that each user download a unique copy of the browser. The approaches by Giuffrida et al. and Homescu et al. both require a heavily customized compiler and do not work with the standard build tools for Linux and Windows.

Instruction Location Randomization (ILR) by Hiser et al. [40] rewrites binaries in a new randomized encoding that is interpreted by a virtual machine with a performance overhead of about 15%. Unlike our approach, ILR is incompatible with just-in-time compiled code.

Binary stirring by Wartell et al. [77] processes binaries at install time by disassembling them and adding a load-time component; it also needs a run-time component due to imperfect disassembly. It is not suitable for our purposes since it relies on a commercial disassembler that cannot be bundled with free software. Additionally, performing additional processing at installation time invalidates the code signature of a signed program.

Marlin by Gupta et al. [38] also randomizes binaries at load time. Unlike binary stirring, Marlin does not contain a runtime-component to detect and compensate for disassembly errors. While the omission of a runtime component lowers overheads in time and space, Marlin is limited to simple ELF binaries that disassemble without errors.

A recent patch submitted to OpenBSD [25] randomizes the layout of the C library during system boot. In particular, the patch permutes the linking order of each translation unit. This shuffles symbols (e.g. functions) relative to symbols defined in other files but does not change the order of symbols defined in the same translation unit. The OpenBSD approach therefore adds less entropy than selfrando which shuffles each function independently no matter what translation unit defines it. Moreover, selfrando generates a different layout for each application each time it launches, preventing the attacker from leveraging a vulnerability in one application to disclose the layout of the library in a different application on the same system.

7.2 Leakage-resilient diversity approaches

Unfortunately, security tools based solely on randomization are vulnerable to attacks aimed at disclosing the pointers to code pages. Snow et al. [66] showed that, if the attackers can read arbitrary memory pages through a vulnerability, they can recursively scan the memory, find other code pages, disassemble them and craft an ad-hoc ROP attack (JIT-ROP). Bittau et al. [8] showed that it is possible to perform a similar attack even without a complete memory read vulnerability, just by observing whether the program crashes for a particular input (this particular attack would not work if the program randomizes itself for each run).

Thus, even fine-grained randomization does not provide complete leakage resilience on its own. This has motivated numerous papers that combine memory randomization techniques with integrity checks (such as execute-only memory) to provide comprehensive protection.

Execute-only memory on x86 processors is difficult to achieve because read permissions are implicitly granted to executable pages. To do so, XnR by Backes et al. [7] marks all pages *not present* and inspects every page reference inside the operating system page-fault handler. HideM by Gionta et al. [34] uses a particular TLB implementation available in certain processors. Readactor by Crane et al. [20] uses a lightweight hypervisor in order to enable the extended page tables feature in modern x86 processors and enforce execute-only memory in hardware. LR^2 by Braden et al. [12] uses a software-only approach based on load masking.

Many of these tools include randomization to provide comprehensive attack resilience; most implementations randomize the code at compile time. These tools could be made more practical by using selfrando to simplify distribution without sacrificing security.

7.3 Integrity-based defenses

Control-flow integrity (CFI) [4, 5] prevents control flow hijacking by only allowing jumps and calls at run-time that are present in the source. Implementing CFI with acceptable performance overhead on commodity hardware is hard; thus, many CFI implementations trade coarse-grained CFI enforcement for better performance.

Most CFI implementations do not rely on randomization, so an attacker can exploit a coarse-grained CFI policy by carefully constructing a malicious payload offline and then using it [13, 23, 36, 37]

Finally, Code-Pointer Integrity (CPI) aims to prevent pointer hijacking by storing code pointers, pointers to code pointers etc. in a safe region; all accesses to the safe region are instrumented to ensure the integrity of the pointers. Performance overhead is relatively small because CPI only needs to instrument a subset of memory operations. The critical issue is the protection of the safe region; on 64-bit Intel processors, segmentation is not available, thus CPI is forced to use information hiding. Unfortunately, the most efficient implementations of this defense can also be bypassed [28].

8 Conclusions

The most widely used and privacy-sensitive programs have large code bases which makes it virtually impossible to ensure that they contain no vulnerabilities. Many exploit mitigations have been proposed to prevent attacks, however no existing tool has the performance and deployability properties that are needed for complex but user-friendly software such as the Tor Browser.

We have introduced selfrando, a fast and practical load-time randomization tool. It has negligible run-time overhead, a perfectly acceptable load-time overhead, and it requires no changes to protect the Tor Browser.

Moreover, selfrando can be combined with integrity techniques such as execute-only memory to further secure the Tor Browser and virtually any other C/C++ application.

Acknowledgments

This work was supported in part by the German Science Foundation (project S2, CRC 1119 CROSSING), the European Union's Seventh Framework Programme (609611, PRACTICE), and the German Federal Ministry of Education and Research within CRISP.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts FA8750-15-C-0124, FA8750-15-C-0085, and FA8750-10-C-0237 and by the National Science Foundation under award number IIP-1520552.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, the National Science Foundation, or any other agency of the U.S. Government.

References

[1] Jetstream 1.1. http://browserbench.org/JetStream/.

[2] Massive: the asm.js benchmark. https://kripken.github.io/Massive/.

[3] Octane 2.0. http://chromium.github.io/octane/.

[4] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, 2005.

[5] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information System Security*, 13, 2009.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 2000.

[7] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[8] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh. Hacking blind. In *35th IEEE Symposium on Security and Privacy*, 2014.

[9] Black Duck Software, Inc. Chromium project on Open Hub. https://www.openhub.net/p/chrome, 2014.

[10] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. In *6th ACM Symposium on Information, Computer and Communications Security*, 2011.

[11] E. Bosman and H. Bos. Framing signals—a return to portable shellcode. In *35th IEEE Symposium on Security and Privacy*, 2014.

[12] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-resilient layout randomization for mobile devices. In *23rd Annual Network and Distributed System Security Symposium*, 2016.

[13] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium*, 2014.

[14] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, 2015.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A. Sadeghi, H. Shacham, and M. Winandy. Return-oriented programming without returns. In *ACM SIGSAC Conference on Computer and Communications Security*, 2010.

[16] X. Chen. ASLR bypass apocalypse in recent zero-day exploits. http://www.fireeye.com/blog/technical/cyber-exploits/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html, 2013.

[17] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12, 1993.

[18] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *8th USENIX Security Symposium*, 1998.

[19] J. Cox. Confirmed: Carnegie Mellon University attacked Tor, was subpoenaed by Feds. http://motherboard.vice.com/read/carnegie-mellon-university-attacked-tor-was-subpoenaed-by-feds, 2016.

[20] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical code randomization resilient to memory disclosure. In *36th IEEE Symposium on Security and Privacy*, 2015.

[21] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. D. Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[22] L. Davi, A. Dmitrienko, S. Nürnberger, and A. Sadeghi. Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM. In *8th ACM Symposium on Information, Computer and Communications Security*, 2013.

[23] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium*, 2014.

[24] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code randomization resilient to (Just-In-Time) return-oriented programming. In *22nd Annual Network and Distributed System Security Symposium*, 2015.

[25] T. de Raadt. openbsd-tech — Anti-ROP mechanism in libc. https://marc.info/?l=openbsd-tech&m=146159002802803&w=2, 2016.

[26] R. Dingledine. Tor security advisory: "relay early" traffic confirmation attack. https://blog.torproject.org/blog/tor-security-advisory-relay-early-traffic-confirmation-attack/.

[27] R. Dingledine. Tor security advisory: Old tor browser bundles vulnerable. https://lists.torproject.org/pipermail/tor-announce/2013-August/000089.html, 2013.

[28] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *36th IEEE Symposium on Security and Privacy*, 2015.

[29] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[30] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *6th Workshop on Hot Topics in Operating Systems*, 1997.

[31] F. S. Foundation. Gcc manual — § 3.10, options that control optimization. https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc/Optimize-Options.html#index-ffunction-sections-1103, 2015.

[32] M. Franz. E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, 2010.

[33] G. Fresi Roglia, L. Martignoni, R. Paleari, and D. Bruschi. Surgically returning to randomized lib(c). In *25th Annual Computer Security Applications Conference*, 2009.

[34] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *5th ACM Conference on Data and Application Security and Privacy*, 2015.

[35] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced operating system security through efficient and fine-grained address space randomization. In *21st USENIX Security Symposium*, 2012.

[36] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of control: Overcoming control-flow integrity. In *35th IEEE Symposium on Security and Privacy*, 2014.

[37] E. Göktas, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium*, 2014.

[38] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino. Marlin: A fine grained randomization approach to defend against ROP attacks. In *Network and System Security*. 2013.

[39] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *ACM Workshop on Cloud Computing Security*, 2009.

[40] J. Hiser, A. Nguyen, M. Co, M. Hall, and J. Davidson. ILR: Where'd my gadgets go. In *33rd IEEE Symposium on Security and Privacy*, 2012.

[41] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automatic software diversity. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[42] A. Homescu, T. Jackson, S. Crane, S. Brunthaler, P. Larsen, and M. Franz. Large-scale automated software diversity—program evolution redux. *Dependable and Secure Computing, IEEE Transactions on*, 2015.

[43] Itanium informal industry coalition. Itanium C++ ABI: Member pointers. https://mentorembedded.github.io/cxx-abi/abi.html#member-pointers, 1999-2015.

[44] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference*, 2006.

[45] G. Koppen. Include selfrando patches into our hardened builds. https://trac.torproject.org/projects/tor/ticket/17406, 2015.

[46] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz. SoK: Automated software diversity. In *35th IEEE Symposium on Security and Privacy*, 2014.

[47] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2004.

[48] C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[49] Microsoft. Data execution prevention (DEP). http://support.microsoft.com/kb/875352/EN-US/, 2006.

[50] Microsoft. Exploitation Trends. *Microsoft Security Intelligence Report*, 16, 2013.

[51] S. Nagy. Address sanitizer local root. http://seclists.org/oss-sec/2016/q1/363, 2016.

[52] Nergal. The advanced return-into-lib(c) exploits: PaX case study. *Phrack Magazine*, 11, 2001.

[53] G. Owenson. Analysis of the FBI Tor malware. http://blog.owenson.me/analysis-of-the-fbi-tor-malware/, 2013.

[54] PaX Team. *Homepage of The PaX Team*, 2001. http://pax.grsecurity.net.

[55] M. Perry. Deterministic builds part one: Cyberwar and global compromise. https://blog.torproject.org/blog/deterministic-builds-part-one-cyberwar-and-global-compromise, 2013.

[56] M. Perry. iSEC partners conducts Tor Browser hardening study. https://blog.torproject.org/blog/isec-partners-conducts-tor-browser-hardening-study, 2014.

[57] K. Poulsen. FBI admits it controlled Tor servers behind mass malware attack. https://www.wired.com/2013/09/freedom-hosting-fbi/, 2013.

[58] T. Ritter and A. Grant. iSEC Partners Final Report — Tor Project Tor Browser Bundle. https://github.com/iSECPartners/publications/tree/master/reports/Tor%20Browser%20Bundle, 2014.

[59] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *36th IEEE Symposium on Security and Privacy*, 2015.

[60] J. Seibert, H. Okhravi, and E. Söderström. Information leaks without memory disclosures: Remote side channel attacks on diversified code. In *ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[61] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, 2012.

[62] F. J. Serna. The info leak era on software exploitation. In *Blackhat USA*, 2012.

[63] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *ACM SIGSAC Conference on Computer and Communications Security*, 2007.

[64] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *ACM SIGSAC Conference on Computer and Communications Security*, 2004.

[65] sinn3r. Here's that FBI Firefox exploit for you (cve-2013-1690). https://community.rapid7.com/community/metasploit/blog/2013/08/07/heres-that-fbi-firefox-exploit-for-you-cve-2013-1690, 2013.

[66] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *34th IEEE Symposium on Security and Privacy*, 2013.

[67] A. Sotirov. Heap Feng Shui in JavaScript. In *Blackhat Europe*, 2007.

[68] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the memory secrecy assumption. In *2nd European Workshop on System Security*, 2009.

[69] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In *34th IEEE Symposium on Security and Privacy*, 2013.

[70] The Clang Team. Clang 3.8 documentation SafeStack. http://clang.llvm.org/docs/SafeStack.html, 2015.

[71] The Firefox Developers. Mozilla foundation security advisory 2013-53: Execution of unmapped memory through onreadystatechange event. https://www.mozilla.org/en-US/security/advisories/mfsa2013-53/, 2013.

[72] The Gitian developers. Gitian: a secure software distribution method. https://gitian.org/.

[73] The Tor Project. The tor browser. https://www.torproject.org/projects/torbrowser.html.

[74] The Washington Post. Meet the woman in charge of the FBI's most controversial high-tech tools. http://wapo.st/1m7UMBQ, 2015.

[75] C. Tice. Improving function pointer security for virtual method dispathes. https://gcc.gnu.org/wiki/cauldron2012?action=AttachFile&do=get&target=cmtice.pdf, 2012.

[76] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. W. Freeh, and P. Ning. On the expressiveness of return-into-libc attacks. In *14th International Symposium on Research in Attacks, Intrusions and Defenses*, 2011.

[77] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *ACM SIGSAC Conference on Computer and Communications Security*, 2012.

[78] D. Williams, W. Hu, J. W. Davidson, J. D. Hiser, J. C. Knight, and A. Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security Privacy*, 2009.

[79] Zerodium. Our exploit acquisition platform. https://www.zerodium.com/program.html, 2015.

A Overview of the exploit used by the FBI in 2013

In 2013, the FBI compromised a number of servers used by Tor hidden services and used them to serve an exploit to de-anonymize users of the Tor network [57]. When the user visited one of the booby-trapped pages with the Tor Browser, the exploit abused an use-after-free vulnerability of Firefox in order to enable arbitrary code execution [65]. The main payload of the exploit collected the MAC address and the host name from the victim machine and sent the data to an attacker-controlled web server, bypassing Tor [53]. That message also included a unique ID provided by the booby-trapped page in order to correlate a specific user to a specific visit. The attacker then knew the public IP address, MAC address and host name of every user that visited the booby-trapped page.