SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

July 2015, Vol. 58 072102:1–072102:15
doi: 10.1007/s11432-015-5285-y

# WebC: toward a portable framework for deploying legacy code in web browsers

YIN Jie[1]*, TAN Gang[2], BAI XiaoLong[1] & HU ShiMin[1]

[1]*Department of Computer Science and Technology, Tsinghua University, Beijing* 100049*, China;*
[2]*Department of Computer Science and Engineering, Lehigh University, Bethlehem PA* 18015*, USA*

Received April 29, 2014; accepted November 26, 2014 ; published online April 7, 2015

**Abstract** For security, most web applications are developed in some type-safe language, such as JavaScript or Java. However, there is a huge amount of legacy codes developed in unsafe languages, which provide rich functionality and are more efficient than their type-safe counterparts. To allow browsers to incorporate type-safe components in a secure way, previous approaches use the software-based fault isolation (SFI) to isolate untrusted legacy code. The SFI approach performs machine-code transformation for security, but the downside is the loss of architecture independence. We propose WebC, a system that allows legacy code transmitted over the web via the Low Level Virtual Machine (LLVM) bitcode format. The untrusted bitcode is transformed by WebC into code in the WebC security language, which enforces both memory isolation and control-flow integrity. Compared with previous approaches, WebC is more portable, provides stronger security, and allows more flexible memory management. Experimental results show that the average runtime overhead of WebC is modest.

**Keywords** WebC, software fault isolation, symbol link, shadow memory area, portable

**Citation** Yin J, Tan G, Bai X L, et al. WebC: toward a portable framework for deploying legacy code in web browsers. Sci China Inf Sci, 2015, 58: 072102(15), doi: 10.1007/s11432-015-5285-y

## 1 Introduction

Modern web applications provide rich functionality. For security, browsers prefer web applications to be developed in some type-safe language, such as JavaScript or Java. In this way, capabilities of applications are confined by the language model. On the other hand, certain web applications may need to incorporate components developed in an unsafe language such as C or C++. As one example, computer games running inside a browser often demand higher performance than what JavaScript can provide. Recent comparison between languages showed that JavaScript programs are 3x–20x slower than corresponding C/C++ programs even using Google's V8, a highly optimized JavaScript engine. Another motivation for a web application to incorporate C/C++ components is to reuse legacy code.

To accommodate the need to incorporate C/C++ components, web browsers often provide some interface for third-party developers to implement new plug-ins that can extend a browser's functionality. For example, Firefox provides the Netscape plugin application programming interface (NPAPI) [1]. A plug-in is implemented in a native language such as C/C++ and extends the browser with powerful features, but

* Corresponding author (email: yin-j10@mails.tsinghua.edu.cn)

 

the downside is that a malicious or a buggy plug-in can possibly damage the web browser or even the host system. A plug-in that resides in the same address space as the browser can modify the browser's critical data structures and invoke operating system (OS) system calls directly.

Our system, called WebC, aims to allow browsers to incorporate legacy C/C++ code with some security guarantee. It has the same goal as Native Client (NaCl [2]) and Xax [3]. NaCl uses software-based fault isolation (SFI [4,5]) to isolate untrusted native code inside the Chrome browser. One limitation, however, is that a NaCl module must be tied with a specific architecture. As a result, NaCl has to provide separate implementations for x86-32 [2], x86-64 [6], and ARM (Advanced RISC Machine) [6]. Portable Native Client (PNaCl[1)]) uses a portable intermediate language for storing the target program. For PNaCl, it is the intermediate language of the program that is portable across different architectures instead of the implementation of the security constraint on the native code. Hence, for PNaCl to support a new architecture, NaCl has to design and implement a new architecture-dependent native code constraint. Architecture dependence threatens web portability, the reason why Mozilla and other browser vendors openly objected to the integration of NaCl into their browsers[2)]. For systems such as XFI [7], although there is no architecture-specific constraint, the implementation of the rewriter and verifier is architecture specific. For example, on x86 architecture, the rewriter does not handle MultiMedia eXtension (MMX), Streaming SIMD Extensions (SSE) instructions which are commonly used in general purpose programming.

In WebC, untrusted legacy code downloaded to a browser is in an intermediate representation (IR): LLVM's bitcode. The bitcode format is mostly machine independent and much more portable than native code[3)]. Furthermore, the code producer can easily produce bitcode by compiling legacy C/C++ components through LLVM's toolchain. The bitcode language, however, is not safe. For instance, bitcode can perform pointer arithmetic and use the resulting pointer to read/write arbitrary memory location. Therefore, WebC has to constrain the bitcode behavior and ensures that it does not damage the browser's security.

To constrain the behavior of the legacy code, WebC performs the bitcode-level transformation on each downloaded program. Since the implementation of WebC targets at the bitcode which is more portable, WebC is implementation portable across different architectures. IR-level transformation not only provides better portability, but also allows the compiler back-end to perform aggressive optimization for better performance. Specifically, WebC converts input bitcode into code in the WebC security language, a restricted form of bitcode. Key aspects of the security language are described as follows.

1. The data region may be discontiguous and is partitioned into a set of memory chunks. All data (except for return addresses) are in those memory chunks. In SFI systems such as NaCl, the data region is a single contiguous memory region of a predetermined size. By contrast, the data region of a WebC application is discontiguous: although each memory chunk is a contiguous region, memory chunks may not be adjacent. As a result, WebC's memory management is more flexible than SFI systems. For instance, when the web application requires more memory than predetermined, those SFI systems can not dynamically adjust the limit of memory size. However, there is no such predetermined memory size constraint in WebC. To protect memory access to a discontiguous data region, we propose a novel instrumentation technique using a shadow memory area (described in Section 3).

2. A function symbol-link pool is built for indirect calls. The pool includes entries for possible indirect call targets. Before an indirect call, runtime checks are inserted to ensure the call target falls into one of the entries in the pool. In a similar way, a label symbol-link pool is built for an indirect branch and the branch is instrumented so that it targets only entries in the pool. Using symbol-link pools, the WebC runtime can quickly check the security of indirect calls and jumps.

3. A return instruction is another kind of "indirect branches" as it retrieves a return address from the

1) Donovan A, Muth R, Chen B, et al. PNaCl: portable native client executables. http://src.chromium.org/viewvc/native_client/data/site/pnacl.pdf, 2010.

2) Metz C. Mozilla: our browser will not run native code. http://www.theregister.co.uk/2010/06/24/jay_sullivan_on_fire, 2010.

3) Regrettably, the bitcode is not totally machine independent and depends on endianness, integer sizes, and other machine aspects; please see future work for more discussion.

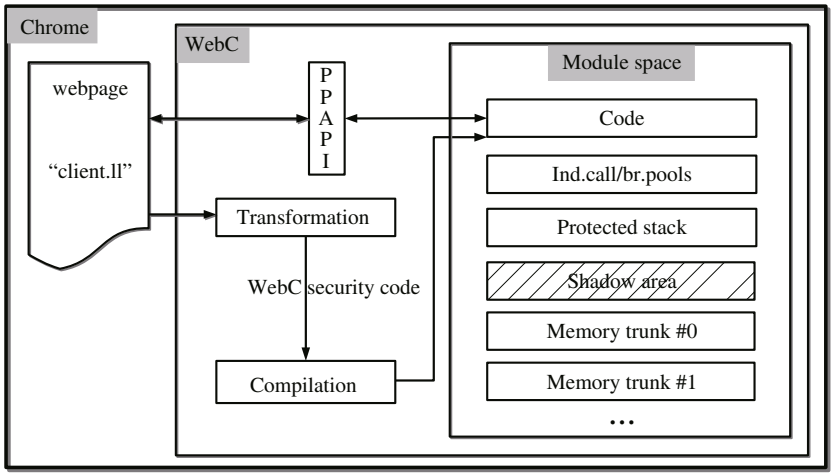


**Figure 1** WebC overview.

stack and transfers the control to the address. However, instrumenting all return instructions is costly. To avoid that, WebC's transformation separates return addresses from the rest of the stack data. There are two stacks after the transformation: a protected stack and a data stack. The protected stack resides outside of the data region and is protected, allowing WebC to not instrument return instructions. The data stack is used to pass function arguments and resides in the data region.

The security language enforces the following security policy.

**Memory isolation.** Memory reads and writes stay in a discontiguous data region. This in general does not prevent reads/writes outside of an array and is a coarse-grained memory isolation policy.

**Control-flow integrity.** All function calls, including indirect calls, target a legal function in the bitcode or a runtime library function. Furthermore, because return addresses are protected, a callee function always returns the control to its caller. For indirect branches, their targets are in a set of basic-block entries.

Major contributions of WebC include the following.

1. WebC provides a framework for incorporating legacy C/C++ components into web browsers in a much more portable way than directly constraining native code.

2. We have performed evaluation of WebC using a set of benchmark programs and case studies. Experiments show that WebC adds modest runtime overhead and requires no change to source code when porting existing legacy applications.

The rest of the paper is organized as follows. In Section 2, we discuss the overflow of WebC. In Section 3, we present WebC's transformation that converts untrusted bitcode into code in WebC's security language. Section 4 presents evaluation results. We then discuss limitations and future work in Section 5, related work in Section 6, and conclude in Section 7.

## 2 WebC workflow

WebC is implemented as a Chrome Pepper Plugin API (PPAPI) plug-in. We next illustrate its workflow using a simple example, as depicted in Figure 1. In this example, imagine a webpage[4)] that contains the following tag loaded in Chrome.

```
<object id="plugin2d"
type="application/ppapi-webc"
style="width:100px;height:100px"
ppsrc="http://example.com/client.ll">
```

4) From http://example.com.

The type attribute of the tag instructs Chrome to forward the tag to WebC. WebC then downloads client.ll as identified in the ppsrc attribute. Code in client.ll is in the LLVM bitcode format.

After receiving client.ll, WebC constructs a logical module space. The space includes an immutable code region where code will be loaded, a read-only region for storing symbol-link pools, a protected stack, and multiple memory chunks. Symbol-link pools contain entries for functions and labels that indirect branches may jump to. The data region contains multiple memory chunks, which are allocated on demand by the memory allocator. As we will explain, there is also a special memory chunk called the shadow memory, which is used to control memory access.

WebC then performs bitcode-level transformation to convert client.ll into code in the WebC security language. Among many transformations, this step inserts checks before dangerous instructions, including memory operations, indirect calls, and branches. Because by this point the module space has been constructed, those checks can have hard-coded constants (e.g., the beginning address of the shadow memory area). Transformations will be presented in detail in Section 3.

WebC then starts an LLVM virtual machine with the transformed client.ll as the input. The virtual machine optimizes and converts the transformed code into native code. The module in native code is then loaded in the logical module space: the native code is loaded into the code region, symbol-link pools are filled with call and branch targets, global variables are put into a memory chunk and initialized, and so on. Afterward, WebC invokes PPP_InitializeModule implemented in client.ll to initialize the module.

The bitcode module can interact with JavaScript code in the webpage through the PostMessage system provided by PPAPI. Through WebC, JavaScript code can send a message to the bitcode module and vice versa. For example, JavaScript code may send a message to the tag that loads client.ll. This message is received by WebC, which is then forwarded to the bitcode module.

## 3 WebC transformation and checking

WebC's security is enforced by performing transformation on untrusted bitcode to code in the WebC security language. We first discuss the main steps in transformation. We then discuss how a few issues are dealt with to cover the full bitcode language.

### 3.1 WebC transformation

Since memory accesses, indirect calls, and indirect branches in bitcode can potentially violate the security policy (memory isolation and control-flow integrity), WebC inserts checks before those instructions. Those checks are inlined in the transformed program.

**Memory access.** Before a memory access, a check is inserted to ensure the access stays within the data region, protecting the browser's integrity and confidentiality. WebC's data region is discontiguous and consists of a collection of memory chunks. In SFI systems, the data region is a contiguous memory region and it is straightforward to restrict a memory access. With a discontiguous data region, a different memory-protection scheme is required.

We propose a novel memory-protection technique using a shadow memory area. It takes advantage of hardware page protection and goes as follows: (1) Assume the size of a physical memory page is 4 KB ($2^{12}$ bytes). (2) The data region has multiple memory chunks, each of size $2^n$ bytes (with $n \geqslant 12$); all addresses in a memory chunk share the same upper bits; for instance, when $n = 24$, the address range of one example memory chunk is [0xAB000000, 0xABFFFFFF]. (3) One memory chunk is special, called the shadow memory area; it is a collection of physical pages; we call these pages shadow pages. (4) One memory chunk corresponds to one shadow page. (5) The access permission for a memory chunk is made the same as the access permission for the corresponding shadow page in the shadow area; for instance, if a memory chunk is readable and writable, then the corresponding shadow page is made readable and writable through page protection, respectively; if a memory chunk is not accessible to the application, then the corresponding shadow page is neither readable nor writable.

With this setup, WebC rewrites bitcode so that an access to an address in a memory chunk is preceded by an access to an address in the corresponding shadow page. Remember that the access permission of a memory chunk is made the same as the corresponding shadow page. Therefore, if the original access is disallowed by the policy, the inserted access to the shadow page would generate a hardware trap[5)]. Such a trap is caught by WebC's runtime. Our implementation terminates the application when a trap occurs, but sophisticated recovery could be performed.

Specifically, an access through address $X$ in "load/store $X$" is instrumented as

$$\begin{aligned}&\text{load } ((X \gg (n-12)) \mid \$\text{ShadowTag})\\&\text{load/store } X\end{aligned}$$

In the above, the ShadowTag is the beginning address of the shadow area. The right-shift operation in "$X \gg (n-12)$" maps all addresses in one memory chunk into an address in a physical page. The bitwise or operation with ShadowTag turns it into an address in the corresponding shadow page.

In our implementation, $n = 24$, meaning that each memory chunk has 16 MB. In a 32-bit architecture, there are at most $2^{32-24} = 2^8$ memory chunks and therefore $2^8$ shadow pages. Suppose the ShadowTag is 0x1C000000. Then, the check before a memory access is "load ($(x \gg 12)$ | 0x1C000000)". As a result, a memory access in the range of [0xAB000000, 0xABFFFFFF] is preceded by a memory access in the range of [0x1C0AB000, 0x1C0ABFFF].

Our scheme for protecting a discontiguous data region has a number of benefits. First, all memory shadow pages are adjacent, which produce good memory-caching effect. Second, the inserted memory access has no data dependence to the original memory access, which allows out-of-order execution in modern central processing units (CPUs).

WebC uses the custom BGET[6)] memory allocator to allocate memory chunks on demand. Memory chunks are allocated by calling the libc function memalign. When the program invokes routines such as malloc and calloc, the WebC runtime first tries to find an available memory chunk that can satisfy the memory-allocation request. If it fails, the runtime calls memalign to find another available memory chunk. When some memory needs to be freed, WebC identifies its owner memory chunk and frees it in the memory chunk. When all memory in a memory chunk has been freed, the memory chunk is deallocated.

**Indirect calls.** Before an indirect function call (i.e., a call through a function pointer), a check is inserted to enforce the target to be a legal function entry. This is to enforce control-flow integrity. The original control-flow integrity (CFI) work [8] instruments machine code to insert IDs into the immutable code region and performs runtime checks before computed jumps. This technique would not be easy to implement at the bitcode level because it is unclear how to insert arbitrary IDs into bitcode. WebC implements a different technique for control-flow integrity. At a high level, the technique consists of three steps: (1) WebC builds a symbol-link pool, which contains an entry for each function; (2) function-address constants in the bitcode are replaced by the address of the corresponding symbol link; and (3) before an indirect call, a check is inserted to ensure that the call targets one of the symbol links in the pool.

Figure 2 depicts the contents of the function–symbol–link pool for an example program with two functions called foo and compare. For each function, WebC generates a function entry in the table at a 16-byte aligned address. The entry stores the encoding of a jump instruction that jumps to the corresponding function; therefore, if an indirect call targets the entry, the control will be transferred to the correct function. Each function also has another entry in the table, which is the proxy function in case it is used as a callback function in the runtime library; more about why this is needed will be discussed later. The entry of the proxy function has an 8-byte offset from the entry of the original function so that it is easy for the WebC runtime to locate the proxy-function entry based on the entry of the original function. A final note is that the pool resides in a read-only region to prevent the pool from being modified.

5) The reader may wonder why not directly apply page protection to memory chunks. This would be insufficient because in WebC the data region of a web application is in the same address space as the browser. The browser itself may contain readable/writable memory pages that should not be touched by the application. The shadow pages in WebC is like an indirect table that allows the application to access only its own memory pages.

6) Walker J. The BGET memory allocator. http://www.fourmilab.ch/bget/, 1996.

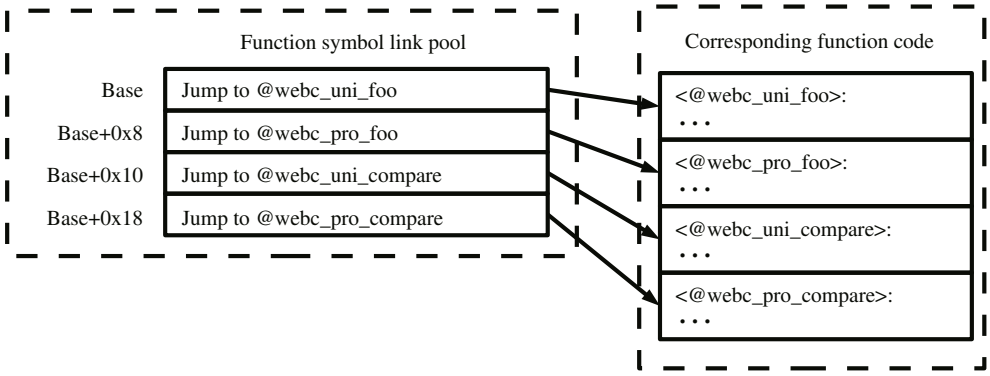


**Figure 2** Function symbol-link pool.

The checking of whether an indirect call targets a valid symbol link is efficient. The number of entries in the symbol-link pool is a power of two; if the number of legal functions is not exactly a power of two, WebC pads the pool with entries that jump to error handling functions. With this setup, it is easy to check whether an indirect call targets a valid symbol link. For instance, if the address range of the symbol-link pool is [0x08602000, 0x08602FFF] and the function address is in pointer ptr, then WebC checks if (ptr & 0xFFFFF00F) == 0x08602000 holds. The last four bits must be zero because each entry in the pool is 16-byte aligned.

**Indirect branches.** An indirect-branch instruction (indirectbr) is supposed to transfer the control to the beginning of a basic block within the current function. The way an indirect branch is handled in WebC is similar to how an indirect call is handled, except there is a symbol-link pool for each indirect branch, while a single symbol-link pool is shared by all indirect calls. LLVM's bitcode requires that each indirectbr be annotated with possible destinations; those destinations are used to build the symbol-link pool for the indirect branch. Furthermore, WebC replaces basic-block constants with the address of the corresponding entry in the pool. Similar to the case of indirect calls, WebC performs checks to ensure that an indirect-branch target falls into those allowed basic-block destinations.

**A two-stack model.** A return instruction retrieves a return address from the stack and transfers the control to the address. Since the stack resides in memory, the worry is that an attacker might modify the return address on the stack and violate control-flow integrity. One way to recover security would be to instrument return instructions to check the return address is valid before the control transfer. However, this approach would result in high overhead since return instructions are common.

WebC uses an alternative approach. It adopts a two-stack model. In this model, the runtime stack is separated into a data stack and a protected stack. Function arguments are passed through the data stack located inside the data region. Return addresses reside in the protected stack outside the data region so that the untrusted code cannot modify it. With return addresses protected, there is no need to instrument return instructions. As a result, the common case of direct calls and returns has no extra overhead.

WebC transforms the input bitcode to conform to the two-stack model. We first discuss a basic two-stack model in which all function arguments are passed through the data stack; later, we will introduce an optimized two-stack model in which some arguments are not passed through the data stack. In the basic two-stack model, all functions are transformed to take exactly one argument: the top of the data stack. A caller of a function pushes function arguments into the data stack and passes the data-stack top to the callee. The callee uses the data-stack top to retrieve arguments from the data stack. Arguments are stored in the data stack in the right-to-left order, compatible with conventions used in variable-argument functions.

**A transformation example.** We next use a simple example to illustrate the transformation process. For simplicity, the example illustrates only the memory-access transformation and the basic two-stack model. The example uses the following C source code.

```
i32 compare (i32*p0,i32*p1) {
        x0 = load i32*p0          ; Memory load via p0
        x1 = load i32*p1          ; Memory load via p1
        x2 = icmp(>,x0,x1)        ; Integer comparison
        x3 = zext i1 x2 to i32    ; Zero extension
        ret i32 x3
}
@a = global i32, align 4
@b = global i32, align 4
i32 foo () {
        x = i32 call compare (i32*@a,i32*@b)
        ret i32 x
}
```

**Figure 3** The example bitcode.

```
1 i32 webc_uni_compare (i8*dt) {
2    t0 = getelementptr i8*dt, i32 0
3    t1 = cast i8*t0 to i32*
4    p0 = load i32*t1
5    t2 = getelementptr i8*dt, i32 4
6    t3 = cast i8*t2 to i32**
7    p1 = load i32**t3
8    MAScheck i32*p0
9    x0 = load i32*p0
10   MAScheck i32*p1
11   x1 = load i32*p1
12   x2 = icmp(>,x0,x1)
13   x3 = zext i1 x2 to i32
14   ret i32 x3
15 }
16 @webc_gvar_a = global i32, align 4
17 @webc_gvar_b = global i32, align 4
18 i32 webc_uni_foo (i8*dt) {
19   t0 = getelementptr i8*dt, i32–8
20   t1 = cast i8*t0 to i32**
21   store i32*webc_gvar_a, i32**t1
22   t2 = getelementptr i8*dt, i32–4
23   t3 = cast i8*t2 to i32**
24   store i32*webc_gvar_b, i32**t3
25   dt'= getelementptr i8**dt, i32–8
26   x = i32 call webc_uni_compare (i8*dt')
27   ret i32 x
28 }
```

**Figure 4** The example bitcode after transformation.

```
int compare(int *p0, int *p1)  return *p0 > *p1;
int a = 3;
int b = 4;
int foo()  return compare(&a, &b);
```

The LLVM bitcode after compilation is in Figure 3. For readability, we use a simplified syntax for LLVM bitcode; the simplified syntax omits many LLVM attributes. We have also added comments to aid understanding.

Figure 4 presents the bitcode after WebC's transformation. Each function in the original bitcode has a corresponding function in the transformed bitcode. After transformation, both arguments p0 and p1 in function compare are passed through the data stack accessible through the data-stack top *dt*. Lines 2–4 and lines 5–7 retrieve arguments p0 and p1 from the data stack, respectively. Instruction getelementptr is for performing pointer arithmetic and does not touch memory. Lines 9 and 11 correspond to the memory loads in the original program. Before them, there are MAScheck operations to ensure that the addresses stay within the data region. Note that we use pseudo-check instructions in the code for clarity; in real transformed code, a check operation is a bitwise shift followed by a bitwise or instruction (and a few type casts), as discussed in Subsection 3.1. Function foo is transformed into webc_uni_foo. Code in lines 19–24 stores arguments in the data stack. Line 25 creates the new data-stack top and the next line makes a function call using the new data-stack top.

Before proceeding, we discuss a few important points. First, in the transformed code there are no check operations before loads/stores through the data stack. Readers may wonder whether that is a security risk. The short answer is that WebC reserves a special memory chunk for holding the data stack and sets up hardware-trapped guard zones around the data stack. It maintains the invariant that the data-stack top always stays in the memory chunk and uses guard zones to guarantee a load/store

through the data-stack top plus a constant offset stays either in the memory chunk or the guard zones. Second, similar to the original stack, both the data stack and the protected stack are per thread so that multithreading is supported. Finally, WebC also implements an optimized two-stack model inspired by XFI [7]. In the basic model, all parameters are passed through the data stack in memory, which may result in high overhead. In the optimized model, if a parameter's address is not taken, then it is left alone during transformation and not passed through the data stack. The following example illustrates the optimized two-stack model. Suppose we have a function as follows.

$$\text{i32 f (i32 } a\text{, i32 } b\text{) ... } \&a \text{ ...}$$

Then after transformation, its signature becomes

$$\text{i32 f (i32*}dt\text{, i32 } b\text{) ...}$$

Parameter $a$ is passed through the data stack, since its address is taken and thus it is subject to pointer arithmetic. Parameter $b$ is left alone and will be most likely passed through a register in the machine code.

### 3.2 Extensions for full bitcode

In this section, we discuss a few aspects that have been ignored in our previous discussion but are needed to cover the full bitcode language.

**Interaction with libraries.** WebC supports a bitcode module to invoke a restricted set of system calls and the PPAPI library, which is the Chrome interface between plug-ins and the browser. The code in those libraries is not subject to the WebC security model and is trusted. WebC's two-stack model complicates the interaction between the bitcode module and libraries. As discussed, the bitcode is transformed to use the two-stack model. However, library code is uninstrumented and uses the protected stack for computation. Therefore, switching between the two modes is necessary when crossing the boundary of the bitcode and libraries.

WebC addresses this problem using automatically generated wrapper functions. For a function named foo, WebC generates two wrapper functions: webc_pro_foo and webc_uni_foo. The function webc_pro_foo has the same function type as foo, while webc_uni_foo passes arguments by the data stack and has the same return type as foo. For an application-defined function, the "pro" (proxy) version copies arguments from the protected stack to the data stack and then invokes the "uni" version, which implements the actual functionality. In this way, if a library function requires a function pointer, the address of the "pro" version can be passed. WebC also generates a "pro" version and a "uni" version for a library function. The "uni" version copies arguments from the data stack to the real stack, and then invokes the "pro" version, which jumps to the real library function. In this way, the address of the "uni" version of a library function can be passed to an application-defined function as a function pointer.

**Stack allocation.** The LLVM alloca instruction allocates memory on the stack frame. In WebC, stack allocation is on the data stack. In particular, WebC's instrumentation replaces alloca by a getelementptr instruction that computes the new data-stack top based on the old top and the size of the allocation (the instruction cannot directly change the old top because of the single-static assignment requirement of bitcode). To prevent data-stack overflow after stack allocation, a check operation is inserted to force the new data-stack top stays within the data region. Without the check, an attacker could perform a sequence of alloca instructions and be able to read and write outside of the data region.

**Unconventional control transfers.** WebC's two-stack model complicates the support for unconventional control transfers such as setjmp/longjmp and exceptions. Those constructs are simulated in WebC. For instance, the simulation of setjmp needs to remember tops of both the protected stack and the data stack and the simulation of longjmp recovers these two-stack tops.

**Variable-argument functions.** WebC supports variable-argument functions. First, the caller pushes arguments to the data stack in the right-to-left order. Second, when a variable-argument function is compiled by LLVM to bitcode, LLVM replaces macros such as va_start by an LLVM intrinsic function.

**Table 1** Line of architecture-dependent code in WebC

| Code category | x86-32 | x86-64 |
|---|---|---|
| Configuration | 16 | 16 |
| Type | 13 | 12 |
| Indirect call/branch pool | 21 | 21 |
| va_start | 24 | 70 |
| Total | 74 | 119 |

**Table 2** Line of architecture-dependent code for WebC and NaCl

| Name | x86-32 | x86-64 |
|---|---|---|
| WebC | 74 | 119 |
| NaCl | ∼1000 [2] | ∼2000 [6] |

WebC then simulates those intrinsic functions. For instance, the intrinsic function for va_start is simulated by calculating a constant offset from the data-stack top.

**C++ features.** C++ is supported by LLVM. Most of the support of C++ in WebC is inherited from LLVM. A few cases need special handling, including exceptions and runtime type information (RTTI).

To support try/catch/throw, WebC reuses the runtime's existing support of excepting handling by compiling the LLVM bitcode into object code, linking the object file into the system's shared objects, and loading the shared objects using the system's shared object loading. In this way, if the original host OS supports try/catch/throw, then WebC supports the same functionality.

To support RTTI, WebC does not change names of C++ variables such as type names and type information of basic types so that when the generated shared object is loaded into the target program, WebC can reuse the OS's executable loader to find the right variables.

## 4 Evaluation

We have performed preliminary evaluation of WebC using benchmark programs and a few case studies. This section presents the evaluation results. The evaluation was performed on a system with an Intel Xeon E5620 CPU (2.40 GHz and 16 cores) and 12 GB of memory. The OS is Ubuntu 11.04 with Linux kernel version 2.6.39 for the x86-32. For x86-64, the OS is Ubuntu 13.04 with Linux kernel version 3.8.0. All programs were compiled through LLVM 3.0 with the maximum level of optimization on.

**Portability.** WebC was first implemented on x86-32, and then ported to x86-64. The implementation includes about 5600 lines of code (LOC) in total for both x86-32 and x86-64. Most of the codes are common in both architectures since the design of WebC does not use any architecture-dependent feature. There are 76 and 118 lines of architecture-dependent code for x86-32 and x86-64, respectively. Table 1 presents the details of WebC's architecture-dependent code.

In Table 1, Configuration denotes the LOC for macro definition codes when WebC is configured for the specific architecture, such as the the size of address space, the size of each memory chunk, and the size of each data stack. Type refers to generating the data types used in the transformation. Indirect call/jump pool represents the LOC for generating indirect call/branch pool since the x86-32 and x86-64 have different opcode formats for indirect call/branch instruction. The first three categories have similar number of LOC, because they are common in both architectures. va_start denotes the LOC for handling va_start. The number of LOC varies a lot because va_start macro has different assumption about the parameters[7] on x86-32 and x86-64. WebC must simulate for both architectures.

Table 2 compares the LOC between WebC and NaCl for porting on x86-32 and x86-64. Compared with NaCl, WebC requires significantly less effort to port to a different architecture, since the design of

7) Matz M, Hubicka J, Jaeger A, et al. System V application binary interface AMD64 architecture processor supplement draft version 0.99.6. http://www.x86-64.org/documentation/abi.pdf, 2013.

**Table 3** SFI benchmark program overhead

| Name | WebC None (%) | WebC WCJ (%) | WebC RWCJ (%) | XFI write (%) | XFI read-write (%) |
|---|---|---|---|---|---|
| hotlist | 0.20 | 0.19 | 19.03 | 4 | 94 |
| lld | 2.73 | 14.44 | 24.66 | 27 | 60 |
| md5 | −38.50 | −29.44 | −18.95 | 3 | 7 |

WebC does not rely on the architecture feature. Hence, most of the codes in WebC can be reused in other architecture. However, NaCl adopts different security policy on different architecture, which makes it hard to reuse existing code. For example, on x86-32, NaCl uses hardware segmentation to protect the memory writing. On x86-64, NaCl requires the special register r15 as the base of the accessible memory region and emits address checking code to protect the memory writing.

**Runtime overhead.** The main runtime overhead of a transformed program is the checking operation before memory reads and writes. For evaluating the runtime overhead, we used the benchmark suites bakeoff and SPLASH-2. The bakeoff suite is a standard benchmark suite used by previous code-sandboxing frameworks such as SFI and XFI systems. It contains three programs: hotlist, lld, and md5. The SPLASH-2 benchmark suite [9] contains 14 multithreaded, computation-intensive benchmark programs. Each program was run three times and we report the average overhead.

All programs were run in WebC in three modes: the none mode, the write mode, and the read-and-write mode. In the none mode, only the transformation for the two-stack model was performed; no runtime checks were inserted before memory reads/writes and indirect calls/branches. This mode was unsafe and we used it only for comparison. In the write-only mode, checks were inserted before memory writes and indirect calls/branches (on top of the transformation for the two-stack model). The write-only mode protects browser integrity and can prevent most attacks (almost all attacks involve at least one memory write). We use WCJ to denote the write-only mode. In the read-and-write mode, checks were also inserted before memory reads. The read-and-write mode protects both browser integrity and browser confidentiality and can be useful in highly sensitive contexts (such as military applications). We use RWCJ for this mode.

Table 3 presents the runtime overhead for programs in bakeoff. The first three columns include numbers for the none mode, the write mode, and the read-and-write mode, respectively. For comparison, the last two columns present the overhead for XFI [7]. The XFI write mode performs only write checks, indirect call, and jump checks. It corresponds to the write-only mode in WebC. The XFI read mode also performs read checks. It corresponds to the read-and-write mode in WebC. The overhead of WebC is much less than XFI.

Figures 5 and 6 present the overheads for SPLASH-2 programs on x86-32 and x86-64, respectively. The performance was also measured in the three modes. The figure shows that programs in the none mode have almost the same performance as the baseline. As we can see from the figure, the average overhead of the write-only mode is 15.9% on x86-32 and 9.8% on x86-64, respectively. The read-and-write mode's overhead is of course higher, with the average being 33.1% on x86-32 and 26.1% on x86-64. We believe the overhead is already acceptable for many applications.

Note for some benchmark programs a mode with more checking may be faster than a mode with less checking. We believe this is because the back-end optimizer and code generator may rearrange and generate code in ways that result in better cache performance. For the same reason, certain programs in a checking mode may run faster than the baseline. This is a phenomenon that has been observed in other IRM systems (e.g. [2]).

Experiments show that WebC's performance is better than previous code-sandboxing frameworks at the machine-code level such as XFI. We believe this is because of two reasons. First, WebC's IR-level design allows the back-end to optimize the transformed code aggressively. For example, if there are two consecutive memory reads through the same pointer, the optimizer can remove the checking before the second read. Second, WebC's transformation can take advantage of rich meta-information at the IR level for better performance. Our optimized two-stack model is an example of that. In comparison,

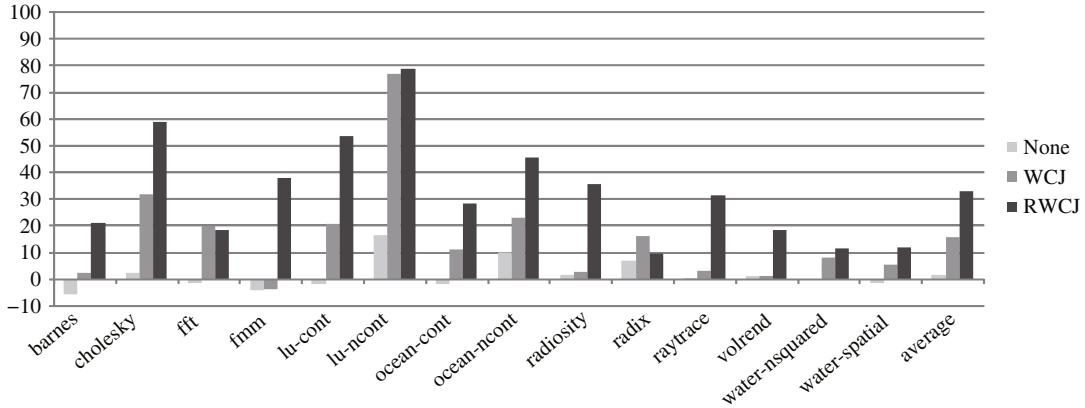


**Figure 5** Runtime overhead (%) of WebC's x86-32 implementation for SPLASH-2 programs.

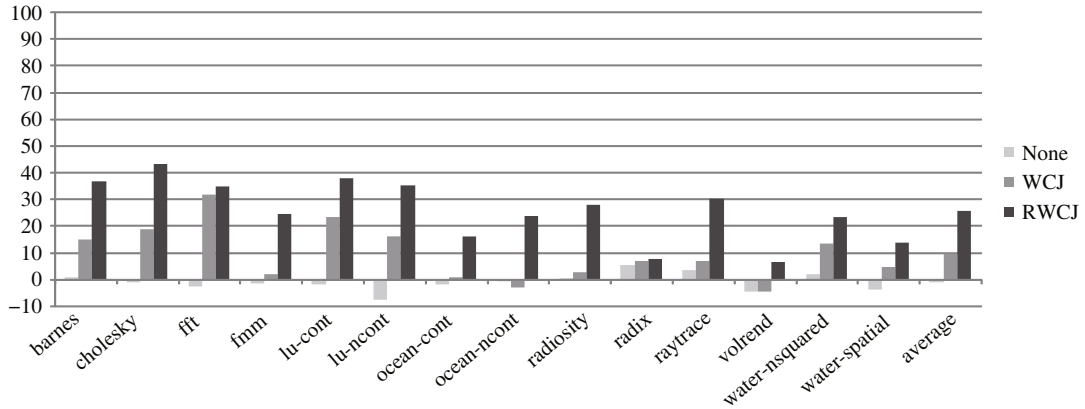


**Figure 6** Runtime overhead (%) of WebC's x86-64 implementation for SPLASH-2 programs.

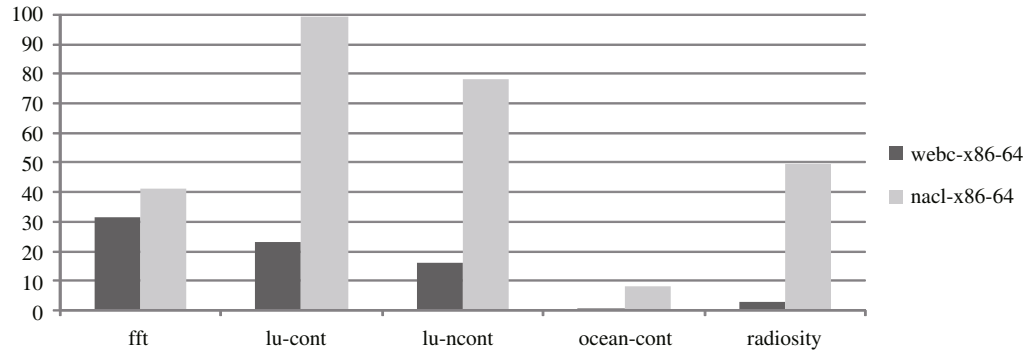


**Figure 7** Runtime overhead (%) of WebC's write-only mode and NaCl on x86-64 for SPLASH-2 programs.

machine-code level SFI implementations have much less meta-information to work with and optimizations are more difficult to apply, resulting in higher overhead. For instance, PittSFIeld [5] incurs about 20% overhead when checking memory writes and computed jumps (but not reads). NaCl x86-32 has a small runtime overhead, but it relies on hardware segmentation, which is unavailable on x86-64 and ARM. As a result, NaCl's implementations on these processors have to rely on a software-only solution; to avoid high-performance penalty when checking both reads and writes, they check only memory writes. Figure 7 presents the comparison between WebC's write only mode and NaCl on x86-64 for SPLASH-2. We choose these programs because other programs require input from disk file or standard input which is not permitted in NaCl. WebC outperforms NaCl because of the back-end optimization.

**Code size.** Table 4 presents the code size of the benchmark programs. Row Raw (KB), NaCl (KB), and WebC (KB) denotes the original size of the benchmark programs in kilobytes and the size of the

**Table 4** Code size increasing between NaCl and WebC

| Program | fft | lu-cont | lu-ncont | ocean-cont | radiosity | AVE.INC (%) |
|---|---|---|---|---|---|---|
| Raw (KB) | 13.8 | 16.9 | 16.4 | 66.6 | 52.1 | |
| NaCl (KB) | 19.7 | 19.0 | 16.8 | 100.4 | 79.2 | 32.2 |
| WebC (KB) | 25.7 | 23.3 | 21.9 | 81.4 | 129.4 | 65.7 |

**Table 5** Stability to attack

| Attacking | Response |
|---|---|
| Dangling pointer dereference | Caught by memory access checking |
| Malicious memory access | Caught by memory access checking |
| Malicious function call | Caught by indirect call checking |
| Stack overflow | Not affecting the protected stack |

benchmark programs generated by NaCl and by WebC in kilobytes, respectively. Column AVE.INC (%) denotes the average code size increasing in NaCl and WebC. The average code size increasing in NaCl is 32.2% and the average code size increasing in WebC is 65.7%. The code size increasing in WebC is mainly due to the "pro" version of the functions which is in a tolerable range.

**Case studies.** We investigated a few case studies to check how easy it is to incorporate legacy applications into Chrome through WebC. These case studies include the Colossal Cave Adventure game, the Simple DirectMedia Layer (SDL) demo program, and the Jump'n Bump game.

Colossal Cave Adventure is originally a text-based game developed in C. There is a NaCl port of the game, which replaces the console display with a text-area tag in a webpage. The webpage has an HyperText Markup Language (HTML) input form. Through the form, JavaScript code reads user input and sends the input as a PPAPI message to a bitcode module that is compiled from the source code of the game. After the bitcode module computes some response, the response is sent to JavaScript code through the PPAPI messaging system. JavaScript then displays the response in the text area.

SDL is a multimedia library that provides low level access to devices, including audio, keyboard, and mouse as well as 3D OpenGL and 2D framebuffer. We ported NaCl's SDL-PPAPI interface to WebC so that a bitcode module can invoke SDL services. With the SDL-PPAPI interface, SDL games such as Jump'n Bump can be run in WebC.

All case studies were run in WebC successfully without any change to the original source code. Of course, some codes were added to exchange messages between the program and JavaScript; but the original source code was unchanged.

**Stability.** To verify if WebC can detect the malware, we manually created several attacking programs, including the dangling pointer dereference, malicious memory read and write, return oriented programming attack, and malicious calling. Table 5 presents the responses of WebC on those attacks. WebC successfully defends against all these attacks.

## 5 Limitations and future work

Using the LLVM bitcode as the mobile code format, WebC provides better code portability than NaCl, which uses native code. On the other hand, bitcode is not completely machine independent and depends on integer sizes and other issues such as how long double and bit fields in structs are supported. Some ongoing effort attempts to create a format of machine-independent LLVM bitcode[8]. Most ideas of WebC can be migrated to such a machine-independent format when it becomes mature.

One drawback of WebC is that it trusts the back-end optimizer and code generator. The optimizer

8) Kang J G. More target independent LLVM bitcode. In: Proceedings of LLVM European User Group Meeting, London, 2011. http://llvm.org/devmtg/2011-09-16/EuroLLVM2011-MoreTargetIndependentLLVMBitcode.pdf.

might wrongly remove checks[9] and the code generator might emit wrong machine code. Furthermore, the register allocator of the back-end might spill a bitcode local variable into memory, resulting in additional memory accesses in the machine code. For smaller trusted computing base (TCB), we desire a machine-code verifier that checks the security of the machine code. In this setup, code is still instrumented at the bitcode level, but the LLVM back-end is modified to emit annotations with machine code to facilitate verification.

WebC uses memory chunk for protecting data outside of the data region. Similar to NaCl, WebC also uses hardware page protection of the OS to protect pages in the memory chunk. The performance overhead of hardware page protection is modest. The limitation is that data within the same page has the same read and write permission. Hence, the user should carefully arrange the data so that read only and writable data are not in the same page.

There are many optimizations WebC can adopt to reduce runtime overhead further. Previous work [10] has shown that once fine-grained control-flow integrity is enforced, aggressive static analysis and transformation can significantly reduce the overhead of checking memory reads and writes. For instance, when a checking is performed inside a loop, it is sometimes safe to hoist the checking outside the loop. We believe these optimizations can be adapted to the bitcode level straightforwardly.

## 6 Related work

Ensuring safe execution of legacy C/C++ components in a trusted host environment has been studied extensively in the literature. One approach is to rewrite the unsafe C/C++ code in a type-safe C variant, such as Cyclone [11]. However, it is impractical to rewrite manually a large amount of legacy C/C++ components. Another approach is proof-carrying code [12], but automatic generation of safety proofs in general is a nontrivial task. Static bug finders such as Fortify and Coverity are useful for finding security-relevant bugs, but they usually target certain kinds of bugs while ignoring other kinds; they do not provide security guarantees.

WebC follows the approach of inlined reference monitors (IRM [13–15]), which inline checks into an untrusted program to enforce a security policy. Many IRMs perform rewriting at the machine-code level or changes the compiler to integrate runtime checks into the emitted machine code, including SFI [2,4–6,12,16–19], CFI [8], and XFI [7]. Different from SFI, WebC protects a discontiguous data region, which makes memory management more flexible. WebC's security is enforced at an IR level. In the approach of IR-level rewriting, the compiler back-end has to be trusted. The upside is that it provides better portability and allows the optimizer an opportunity to optimize the rewritten result for better performance. Furthermore, the IR-level rewriting is more flexible as it can take advantage of rich meta-information. For instance, there is no need to instrument return instructions in WebC, thanks to its transformation; after transformation, return addresses can be protected in a separate region. By contrast, previous SFI/CFI implementations instrument return instructions. By protecting return addresses, WebC ensures that a callee returns the control to its caller, while CFI can only ensure that the callee returns to some possible caller. On the other hand, WebC's protection of return addresses introduces issues with unconventional control-transfer mechanisms such as setjmp/longjmp. WebC's separation of the data stack and the protected stack is similar to the two-stack model of XFI [7].

The SAFECode project [20,21] is closely related. SAFECode is an enhanced version of LLVM that can enforce object-level integrity (which is close to type safety). WebC enforces a weaker policy than SAFECode. In some sense, WebC's policy and transformation are similar to the case when there is only one untyped pool in SAFECode. On the other hand, WebC's transformation does not rely on sophisticated whole-program compiler analysis such as pointer analysis and facilitates separate compilation. Furthermore, WebC's protection of control flow is more complete; it also protects return addresses, which are missing in SAFECode. SoftBound [22] takes a similar approach of IR-level rewriting as WebC. It instruments bitcode for enforcing spatial memory safety, which is stronger than WebC's memory-isolation

9) Recall that our instrumentation inserts before a memory access an extra load instruction, which loads from the shadow memory area. We mark the load instruction volatile to ensure that load instruction is not optimized away.

policy. WebC is concerned with protecting browser integrity and confidentiality. Therefore, a weaker memory policy is sufficient. Consequently, the performance overhead of WebC is much lower than SoftBound.

The goal of Xax [3,23] is also to allow safe execution of legacy code in browsers. It takes a rather different design from WebC and relies on new OS-level abstractions such as picoprocessors. Compared with Xax, WebC is a complete use-space implementation and does not modify the OS kernel.

The legacy code can also be deployed in web browser as a plug-in through a browser-specific interface. For example, Firefox provides the NPAPI interface [1]; Chrome provides the PPAPI interface[10]; and Internet Explorer provides the browser helper object (BHO) interface[11]. Since the plug-in is implemented in a native language such as C/C++, the legacy code can be reused in the browsers to extend its functionality. However, these approaches do not check the security of the plug-ins, which makes the malicious plug-in potentially harm the browser or even the host OS. Compared with them, WebC's security policy enforces the plug-ins against malicious operations.

WebC's goal is similar to PNaCl[1] that also targets at porting legacy code in web browser. PNaCl also requires legacy code be transmitted in the bitcode format, with portability as the goal. There are some key design differences between WebC and PNaCl though. After mobile bitcode is downloaded into the Chrome browser, PNaCl compiles bitcode into SFI-compliant native code and reuses NaCl to constrain native code. In contrast, WebC performs bitcode-level transformation for security. WebC reuses the LLVM toolchain and OS' basic services such as page protection. As a result, WebC's design is more portable than PNaCl, meaning that WebC's transformation can be reused across architectures. Another difference between WebC and PNaCl is that WebC supports a discontiguous data region and much stronger control-flow integrity. PNaCl is tied to NaCl and supports only a contiguous data region and coarse-grained control-flow integrity: the jump target of an indirect call/branch is aligned at 16 or 32 bytes.

## 7 Conclusion

WebC enables legacy code to be safely executed in a web browser. WebC uses a data region to separate the guest application memory from the host web browser. It further uses indirect call/branch tables to enforce strong control-flow integrity. The experimental results show that performance penalty is modest on most benchmark programs. Compared with previous approaches, WebC adopts a unique design where transformation is performed at an intermediate-language level. This design provides better portability and results in better performance.

### References

1 Oeschger. API reference: netscape Gecko plugins 2.190 pgs. Netscape Communication, 2002
2 Yee B, Sehr D, Dardyk G, et al. Native client: a sandbox for portable, untrusted x86 native code. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, 2009. 79–93
3 Douceur J R, Elson J, Howell J, et al. Leveraging legacy code to deploy desktop applications on the web. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation, San Diego, 2008. 339–354
4 Wahbe R, Lucco S, Anderson T, et al. Efficient software-based fault isolation. In: Proceedings of ACM Symposium on Operating Systems Principles, New York, 1993. 203–216
5 McCamant S, Morrisett G. Evaluating SFI for a CISC architecture. In: Proceedings of USENIX Security Symposium, Vancouver, 2006. 209–224
6 Sehr D, Muth R, Biffle C, et al. Adapting software fault isolation to contemporary CPU architectures. In: Proceedings of USENIX Security Symposium, Washington DC, 2010. 1–12
7 Erlingsson U, Abadi M, Vrable M, et al. XFI: software guards for system address spaces. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, 2006. 75–88
8 Abadi M, Budiu M, Erlingsson U, et al. Control-flow integrity. In: Proceedings of the 12th ACM Conference on Computer and Communications Security, Alexandria, 2005. 340–353

10) Important concepts for working with PPAPI. https://code.google.com/p/ppapi/wiki/Concepts, 2010.
11) Schreiner T, Sudds J. Building Browser Helper Objects with Visual Studio 2005. http://msdn.microsoft.com/en-us/library/bb250489.aspx, 2006.

9 Woo S C, Ohara M, Torrie E, et al. The SPLASH-2 programs: characterization and methodological considerations. In: Proceedings of International Symposium on Computer Architecture, Santa Margherita Ligure, 1995. 24–36

10 Zeng B, Tan G, Morrisett G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, Chicago, 2011. 29–40

11 Jim T, Morrisett J G, Grossman D, et al. Cyclone: a safe dialect of C. In: Proceedings of USENIX Annual Technical Conference, Monterey, 2002. 275–288

12 Necula G. Proof-carrying code. In: Proceedings of the 24th ACM Symposium on Principles of Programming Languages, New York, 1997. 106–119

13 Erlingsson U, Schneider F B. SASI enforcement of security policies: a retrospective. In: Proceedings of New Security Paradigms Workshop, Ontario, 1999. 87–95

14 Evans D, Twyman A. Flexible policy-directed code safety. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, 1999. 32–45

15 Erlingsson U, Schneider F B. IRM enforcement of Java stack inspection. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, 2000. 246–255

16 Small C. A tool for constructing safe extensible C++ systems. In: Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems, Portland, 1997. 175–184

17 Ford B, Cox R. Vx32: lightweight user-level sandboxing on the x86. In: Proceedings of USENIX Annual Technical Conference, Boston, 2008. 293–306

18 Zeng B, Tan G, Erlingsson U. Strato: a retargetable framework for low-level inlined-reference monitors. In: Proceedings of USENIX Security Symposium, Washington DC, 2013. 369–382

19 Morrisett G, Tan G, Tassarotti J, et al. RockSalt: better, faster, stronger SFI for the x86. In: Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, Beijing, 2012. 395–404

20 Dhurjati D, Kowshik S, Adve V. SAFECode: enforcing alias analysis for weakly typed languages. In: Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, 2006. 144–157

21 Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, 2006. 162–171

22 Nagarakatte S, Zhao J, Martin M M, et al. SoftBound: highly compatible and complete spatial memory safety for C. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, 2009. 245–258

23 Howell J, Parno B, Douceur J R. How to run POSIX apps in a minimal picoprocess. In: Proceedings of the USENIX Annual Technical Conference, San Jose, 2013. 321–332