Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors

Bin Zeng and Gang Tan, *Lehigh University;* Úlfar Erlingsson, *Google Inc.*

This paper is included in the Proceedings of the
22nd USENIX Security Symposium.

August 14–16, 2013 • Washington, D.C., USA

ISBN 978-1-931971-03-4

Open access to the Proceedings of the
22nd USENIX Security Symposium
is sponsored by USENIX

Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors

Bin Zeng
Department of Computer Science and Engineering
Lehigh University

Gang Tan
Department of Computer Science and Engineering
Lehigh University

Úlfar Erlingsson
Google Inc.

Abstract

Low-level Inlined Reference Monitors (IRM) such as control-flow integrity and software-based fault isolation can foil numerous software attacks. Conventionally, those IRMs are implemented through binary rewriting or transformation on equivalent low-level programs that are tightly coupled with a specific Instruction Set Architecture (ISA). Resulting implementations have poor retargetability to different ISAs. This paper introduces an IRM-implementation framework at a compiler intermediate-representation (IR) level. The IR-level framework enables easy retargetability to different ISAs, but raises the challenge of how to preserve security at the low level, as the compiler backend might invalidate the assumptions at the IR level. We propose a constraint language to encode the assumptions and check whether they still hold after the backend transformations and optimizations. Furthermore, an independent verifier is implemented to validate the security of low-level code. We have implemented the framework inside LLVM to enforce the policy of control-flow integrity and data sandboxing for both reads and writes. Experimental results demonstrate that it incurs modest runtime overhead of 19.90% and 25.34% on SPECint2000 programs for x86-32 and x86-64, respectively.

1 Introduction

Software attacks are common, from code-injection attacks to more sophisticated techniques such as Return Oriented Programming (ROP [6, 29]). ROP chains the attacked program's code snippets, referred to as *gadgets*, to achieve functionality desired by the attacker. It can bypass many defensive techniques such as StackGuard and Data Execution Prevention (DEP) [22].

Low-level inlined reference monitors (IRM [15–17]) are effective at preventing attacks against software systems. In this approach, checks are inlined into binary code to ensure critical security properties. Take the example of software-based fault isolation (SFI). It is a code-sandboxing technique that isolates untrusted modules from trusted environments [32]. By having separate code and data regions and by making the data region unexecutable, SFI prevents code-injection attacks in addition to containing faults in sandboxed modules.

Another effective IRM is control-flow integrity (CFI [1]). An essential step in many software attacks is to induce an illegal control flow transfer to maliciously injected code, or to some library function as in jump-to-libc attacks, or to some existing code snippet as in ROP attacks. CFI enforces a strong runtime guarantee that execution paths follow a predetermined control flow graph, which is constructed either by source-code analysis, binary analysis, or program profiling. CFI can greatly decrease where ROP gadgets can be discovered and further restrain the way gadgets can be chained, thus effectively mitigating ROP attacks.

Low-level IRMs such as SFI and CFI are usually implemented through low-level rewriting, either by performing binary instrumentation, assembly-code instrumentation, or by modifying a compilation tool chain's backend to emit code with embedded checks. As an example, PittSFIeld was implemented by assembly-code instrumentation [21]. Google's Native Client (NaCl [28, 37]) was built by modifying the backend of the GNU tool chain. One key benefit of rewriting at the low level is that a separate verifier can be built to check the result of rewriting. The separate verifier removes the rewriter outside of the TCB. Furthermore, in a distributed environment, only the verifier needs to be installed at the client's side. The verifier checks the security of untrusted, remotely downloaded modules. The security architecture of NaCl follows the separation between the rewriter and the verifier.

On the other hand, low-level rewriting is tightly coupled with a certain ISA, resulting in poor reusability and retargetability, and hindering optimizations. It is non-

trivial to port a low-level IRM to another ISA and existing parts are hard to reuse. For instance, NaCl's initial implementation was on x86-32 and its port to x86-64 and ARM involved significant effort in design and implementation [28]. One reason for the nontrivial effort is the differences among ISAs, including the instruction set, the available hardware features, the number and size of registers, and others. In addition, many components need to be built from scratch. A typical example is optimizations. Any decent IRM implementation requires optimizations to bring down the runtime cost. However, those optimizations are tied to an ISA and hard to reuse.

We explore the building of a retargetable framework for low-level IRMs on a high-level compiler intermediate representation; in particular, the LLVM IR [20]. The framework, called Strato[1], is general in the sense that various security policies can be conveniently enforced and much code can be reused among them and that inlined high-level checks for a specific policy can be lowered into distinct machine-code sequences. In Strato, we have enforced CFI and data sandboxing for both memory writes and reads.

IR-level rewriting comes with several benefits. First, it is retargetable. Security checks are inserted into the high-level representation. The check-insertion component is shared by all target ISAs the compiler supports. Optimizations that operate on the IR are also reused among different targets. To support a new target ISA, only the lowering from high-level checks to machine-instruction sequences needs to be changed. Even for the same ISA, it is easy to explore different machine-instruction sequences that implement the same high-level check since the lowering part can be easily changed. Our framework was originally built to support x86-32 and then extended to support x86-64; only a small amount of code was altered to retarget it for x86-64.

The second benefit of IR-level rewriting is that optimizations are easier to implement and more optimizations can be supported. An IR usually carries a wealth of structured information and attains many properties that are amenable to program analyses and optimizations. For instance, LLVM IR is in the Static Single Assignment (SSA) form [8, 9], making analysis easier to implement. In addition, LLVM IR preserves type information, loop information, and dominator-tree information, which facilitate analyses and optimizations. Finally, a high-level representation contains many fewer instructions than a typical target machine (e.g., in LLVM 2.9, LLVM IR has only 54 instructions while the x86-64 target has over 3,500). All these benefits make it easier to implement optimizers that remove or hoist security checks; we call these optimizers *security-check optimizers*.

However, the downside of pure IR-level rewriting is that it results in a larger TCB compared to low-level rewriting. The compiler backend performs sophisticated transformations to generate low-level code, including instruction selection, ISA-specific optimizations, and register allocation. Those transformations can invalidate the hypotheses assumed by a security mechanism at the high level. First, a bug in a transformation can produce insecure low-level code. A more subtle issue is that those transformations may assume a machine model different from the attack model of a low-level IRM. As a simple example, a backend transformation might assume that a variable holds the last stored value after it is loaded back from the memory location into which the variable is spilled. However, many low-level IRMs such as CFI assume the memory may change arbitrarily between any two instructions because of memory-corruption attacks. Under this attack model, a spilled variable cannot be assumed to hold the same value. Consequently, the transformation may produce insecure code according to the attack model.

Therefore, the challenge is how to perform IR-level rewriting while still preserve low-level security. Strato adopts a twofold approach. First, it includes a novel constraint encoding and checking process to propagate assumptions required by security-check optimizers. The optimizers do not remove checks; instead, they mark them as removable and attach constraints to them. After the backend transformations, Strato checks whether the constraints have been invalidated by the backend. If they are not, the unnecessary security checks are removed or hoisted. Otherwise, the checks are left intact to preserve the security of the low-level code. In other words, the security-check optimizers mark optimizations at the IR level, but the effect is taken only at the low-level code after ensuring constraints are not violated. To further ensure the trustworthiness, we implement an independent verifier to validate the final low-level code, thus removing all the instrumentations, transformations, optimizations, and constraint checking out of the TCB. The verifier helped us uncover 35 critical bugs in early versions of Strato.

The key contributions of our work are as follows.

- A reusable and retargetable framework is proposed, built and evaluated to enforce low-level IRMs on a high-level IR. To the best of our knowledge, this is the first framework that brings the benefits of high-level representations to low-level IRMs and loses no trustworthiness. On top of that, we have implemented two low-level IRMs including CFI and data sandboxing for both x86-32 and x86-64. To demonstrate the benefits of the IR-level approach, we have implemented three conventional optimizations with ease and the runtime overhead is lower than previous work.

- Two techniques are proposed to ensure trustworthiness, including the constraint encoding/checking and the low-level verifier. A constraint language is used to encode assumptions that are carried across the code-generation barrier.

- We explore and evaluate a number of alternative security-check instruction sequences for both CFI and data sandboxing. Different instruction sequences have varying overhead on different ISAs and programs. We have discovered more efficient instruction sequences than previous work.

This paper is organized as follows. Section 2 describes related work. Section 3 introduces the overview of Strato. Section 4 presents how Strato performs check instrumentation and optimization; it also presents the constraint language. Section 5 discusses the phase of constraint checking and check lowering. Section 6 elaborates on the low-level verification process. Section 7 discusses the implementation and evaluation. The last section concludes and proposes future work.

2 Related Work

Strato is inspired by many previous low-level security techniques. Its special focus is to build a retargetable infrastructure to assist the exploration and optimizations of security techniques at a high-level representation.

2.1 Inlined Reference Monitors (IRMs)

IRMs embed checks into subject programs to enforce security policies. This approach can be carried at different language levels, from source code, to an intermediate representation, or to low-level code. A typical example of source-code IRM is CCured [25], which inserts checks into C code for memory safety. At the IR level, a number of systems insert checks for various kinds of policies [11, 15, 17, 24]. At the low level, checks can be inserted to enforce policies such as control-flow integrity. Clearly, this is a well-studied research area. Our system sets itself apart by performing IR-level rewriting and preserving low-level security. The IR-level rewriting is adopted for retargetability and for the ease of implementing optimizations. At the same time, we propose techniques to ensure that IR-level rewriting is valid with respect to security policies at the low-level.

Next we discuss closely related systems in the area of IRMs and compare them with Strato.

Software-based Fault Isolation. SFI isolates untrusted or faulty modules from a trusted environment [21, 28, 31, 32, 34, 37]. In SFI, checks are inserted before memory-access and control-flow instructions to ensure memory access and control flow stay in a sandbox. A carefully designed interface is the only pathway through which sandboxed modules interact with the rest of the system. One subtle requirement of SFI and other IRMs is that the inserted security checks cannot be bypassed by computed jumps, making some form of control flow restriction necessary. Recent SFI implementations use instruction alignment for a crude form of control-flow integrity.

For efficiency, SFI is typically implemented through a combination of static verification and inlined checks. The safety of direct memory accesses and direct jumps can be statically checked. For computed memory visits and indirect jumps, checks are inlined to make sure that those operations stay in the sandbox. Traditional SFI implementations are performed through low-level rewriting. As a result, they are tightly tied to a specific target machine and difficult to port to other ISAs. One advantage of low-level rewriting is that it holds the promise of rewriting without source code being available. However, most previous SFI implementations still ask for the cooperation of the code producer by requiring assembly code or a special compiler to be used. A recent SFI system [34] makes substantial progress toward an implementation through pure binary rewriting; it remains to be seen whether the system can be generalized to IRMs other than SFI and how optimizations can be accommodated.

Control Flow Integrity. CFI ensures that runtime control flow follows a predetermined control flow graph even if the whole data memory is under the control of attackers [2, 3]. One way to enforce CFI is to insert IDs at the targets of a control transfer and a check before the control transfer [3]; the check ensures that the expected ID is at the actual control transfer destination. The system by Wang et al. enforces CFI through defunctionalization [33]. For computed control flow transfers, their system encodes all potential targets in a write-protected table and uses an index to retrieve the target. Before each computed control transfer, the index is checked to make sure that it falls into the table before it is used to fetch the target address from the table. Our system implements CFI in a way similar to the original implementation [3], but at the IR level. In addition, we explore and evaluate a number of instruction sequences for CFI enforcement and find efficient instruction sequences that reduce the runtime cost of CFI.

Combining CFI with other IRMs. On top of CFI, XFI employs a protected shadow stack to store return addresses [14]. XFI promotes control flow precision from Deterministic Finite Automata (DFA) to Pushdown Automata (PDA). However, XFI is platform specific and its runtime overhead is significant.

Our previous system [38] also implements both CFI

and data sandboxing and proposes optimizations to decrease the runtime cost. However, that implementation performs x86-32 assembly rewriting and cannot be retargeted to other ISAs. By contrast, Strato can target any ISA that a compiler supports and the instrumentation and optimizations are shared among different targets. In addition, optimizations in the previous system use the same range-analysis technique adopted in its verifier, making its trustworthiness questionable. Finally, its verifier is path insensitive and is not as accurate as the one in the new system.

LLVM IR rewriting. A number of systems perform rewriting on the LLVM IR for security. SAFECode [11, 12] is an enhanced version of LLVM that can enforce object-level integrity (which is close to type safety). SoftBound [24] also takes the approach of IR-level rewriting. It instruments the LLVM IR for enforcing spatial memory safety. However, these systems enforce their policies only at the IR level, not at the low level. Our system has to solve the key challenge of how to preserve security at the low-level even with the IR-level rewriting.

Portable Native Client (PNaCl) is an ongoing effort at Google. A white paper describes its initial design [13]. PNaCl requires code be transmitted in the LLVM IR format, with portability as the goal. After mobile IR code is downloaded into the Chrome browser, PNaCl compiles the IR code into SFI-compliant native code and reuses NaCl to constrain native code. The important difference between PNaCl and our system is that their architecture does not accommodate security optimizers that remove or hoist checks. The constraint language in our system allows optimizers to perform optimizations and attach constraints that can be checked at the low level.

2.2 Program Shepherding and Virtual Machines

Program shepherding utilizes an efficient program interpreter to enforce security at runtime [19, 27]. The interpreter can enforce various policies during program execution. Similarly, virtual machines either JIT or interpret high-level representations, enforcing relevant security policies during the process. Although many policies can be enforced conveniently in interpreters and virtual machines, the sheer size and complexity of interpreters or JIT compilers make their trustworthiness questionable [10, 18]. Furthermore, the runtime performance of interpreters and virtual machines might be problematic compared with the IRM approach. Strato incurs lower overhead and has a much smaller TCB.

3 Overview of Strato

This section elaborates on the workflow of Strato. We will discuss where checks are inserted and optimized, and where constraint checking and verification happen. We have used Strato to implement CFI and data sandboxing, two specific IRMs. Therefore, we first discuss those IRMs' attack model and security policies.

Attack Model. Strato adopts CFI's attack model [1]. We assume there is a separate code and data region. The data region is under the control of an attacker, who is modeled as a concurrent thread that can overwrite any memory location in the data region. This rather pessimistic assumption is actually realistic given the abundance of memory corruption vulnerabilities. In addition, we assume that the code region and machine registers cannot be changed by attackers. The assumption on the code region can be discharged by hardware protection such as DEP [22] or the W $\oplus$ X protection in latest x86 processors. The assumption about registers is consistent with kernel-based multithreading. Note that even though the attacker cannot directly modify the code region or registers, he/she may indirectly induce such a change. For instance, if a program loads from memory to a register, the register's new value is controlled by the attacker since the data region is controlled by the attacker. If the program further uses the register as the address of a memory-write operation, the operation may change the code region since the register's value is controlled by the attacker. Therefore, a protection mechanism must prevent such indirect effects from damaging the system.

Security policy. In Strato, we have implemented two IRMs: CFI and data sandboxing. The CFI policy is with respect to a control-flow graph, whose edges connect control-transfer instructions to allowed destination basic blocks. We say a program obeys the CFI policy if all control transfers in the program during runtime follow the control-flow graph.

The data-sandboxing policy restricts memory reads and writes. Following previous work [21, 32, 38], we place a *guard zone* immediately before the data region and another guard zone immediately after the data region. We use `gz_size` to denote the size of both guard zones. We assume that access to guard zones is hardware trapped (through page protection). Guard zones facilitate optimizations of data sandboxing. With the guard zones, a memory read or write is safe with respect to the data-sandboxing policy if its address is either in the data region or in the guard zones.

Workflow. In order for an IRM-implementation framework to be retargetable, the majority of instrumentation and optimizations need to be decoupled from a specific ISA. A high-level representation provides such a vehicle. A remaining question is where to insert the

IRM-instrumentation phase inside a compiler. A typical optimizing compiler has many layers that transform an IR program to another IR program with simplified semantics or better performance. Therefore, the IRM-instrumentation phase can be scheduled at any stage between IR generation and the backend.

On one end of the spectrum, we can schedule the IRM-instrumentation pass right after the compiler frontend; that is, after the IR is generated by the frontend. The benefit is that it can reuse a large number of existing IR-level optimization passes, which can optimize away unnecessary security checks. However, it has two drawbacks. First, since the security of low-level code generated by the compiler is what we are interested in, we need a way to ensure those IR-level optimizations do not wrongly optimize away security checks. One way to accomplish this would be to modify the optimizations to carry enough information to the low level for certification, similar to proof-carrying code. However, modifying complex compiler optimizations is non-trivial. The second and more serious drawback of scheduling the IRM-instrumentation pass right after the frontend is that existing optimizations may not be safe according to the attack model of an IRM. As we discussed before, the CFI attack model assumes that data memory is untrusted. However, a typical IR assumes a much different machine model. As an example, LLVM IR adopts an Unlimited Register Machine (URM) model in the SSA form [20]. A real machine has a limited number of registers and so LLVM IR variables may be spilled into untrusted memory locations. Therefore, if an LLVM IR optimization depends on the URM model for correctness, then the optimized result may not be safe according to CFI's attack model.

On the other end of the spectrum, the IRM-instrumentation phase can be scheduled right before the compiler backend for code generation. This is the design adopted in Strato. The downside is that existing compiler optimizations are not reused and we have to develop our own optimizations to optimize IRM checks to improve efficiency. But optimizations for security checks can be implemented straightforwardly at the LLVM IR level, which has a small number of instructions and is in the SSA form. For the policy of data sandboxing, we implemented three optimizations with ease, shared by all targets supported by LLVM. With this design, there is no need to trust or modify a large number of existing IR-level compiler optimizations. Optimizing compilers have a large code base and bugs are unavoidable [35].

Fig. 1 presents the workflow of Strato, which is implemented as extra passes added to the LLVM compiler. We next explain the steps of how source code is translated to low-level code through Strato-augmented LLVM. We add stars to those steps that are added in Strato to distinguish them from those steps already in LLVM.

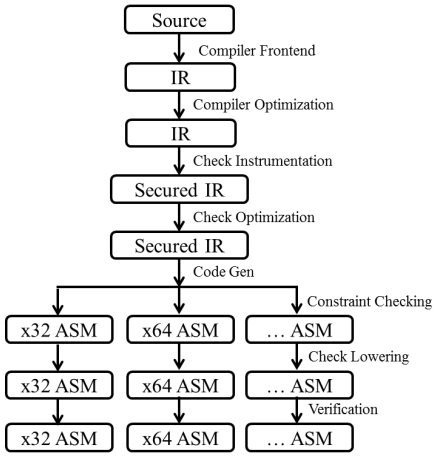

Figure 1: Workflow of Strato

(1) Compiler frontend. LLVM's clang frontend generates the IR code.

(2) Compiler optimizations. LLVM's transformations and optimizations change the IR code to simpler and optimized code.

(3) *Check insertion. Security checks are inserted before dangerous instructions to generate secured IR code. The dangerous instructions and checks inserted depend on the security policy. Since the current policy is CFI and data sandboxing, security checks are inserted before memory loads and stores as well as computed jumps. Note this step inserts more checks than necessary. Later steps will remove unnecessary checks. Security checks are inserted as LLVM intrinsic functions, which will be lowered to machine-instruction sequences at a later step (if they are not optimized away).

(4) *Check optimization. After check insertion, custom optimizations for removing security checks are performed on the IR code. We implement three effective optimizations to demonstrate the amenability of high-level IR to optimizations: redundant check elimination, sequential memory access optimization, and loop-based check optimization. Our optimizations differ from traditional ones in that no security checks are removed or moved around at this step. A check that is deemed unnecessary is marked as removable and constraints are attached to it. The check will be removed only after the constraints are checked to be valid at a later step. If those constraints are violated by later steps of the compilation, the check will not be removed.

(5) Code generation. The compiler backend performs instruction selection, instruction scheduling, ISA-

specific optimizations, and register allocation. Low-level assembly code is generated as the result.

(6) *Constraint checking. If the constraints for a check are invalidated during compiler transformations and optimizations, the check is kept intact. Otherwise, it is removed.

(7) *Check lowering. Security checks are lowered to machine-instruction sequences. Usually a security check can be implemented by multiple machine-instruction sequences. This step therefore provides a convenient place to experiment with different sequences to evaluate which one produces the best performance.

(8) *Verification. An independent verifier is run to check the low-level code. If the verification fails, then the code is rejected.

The above design makes it straightforward to adapt to a different ISA. Steps including check insertion, check optimization, and constraint checking can be reused across ISAs. The check-lowering and the verifier components need to be tailored for a new ISA. As we will discuss, the amount of effort involved to retarget Strato's implementation from x86-32 to x86-64 is small.

4 Check Instrumentation, Optimizations and the Constraint Language

The goal of security-check instrumentation is to guard dangerous operations with checks so that they cannot be abused by adversaries. For CFI, IDs are inserted before control-flow targets. Furthermore, checks are inserted before computed jumps, including indirect calls, indirect jumps and return instructions[2]; these checks ensure that the expected IDs are there at the targets of control-flow transfers [1].

For data sandboxing, Strato inserts a check before each load and store instruction; the check ensures that the instruction's memory address is within the data region. In addition, after a definition of a pointer variable, a check is inserted to ensure that the pointer is within the data region. A check is also inserted at the entry of a function for a pointer parameter. In this step, Strato inserts more checks than necessary.

Since checks are inserted at the IR level, the same protection strategy is adopted for all machine targets, including x86-32 and x86-64. This provides uniformity and enables most of the code to be shared between targets. In contrast, NaCl adopts very different protection strategies for x86-32, x86-64, and ARM (segmentation on x86-32, large addresses and guard regions on x86-64, and address masking on ARM).

After check instrumentation, optimizations are run on the secured IR to mark unnecessary checks. To demonstrate the ease of implementing optimizations at the IR level, we have implemented three optimizations for removing unnecessary data-sandboxing checks: redundant check elimination, sequential memory access optimization, and loop-based optimization. In these optimizations, checks are not removed. Rather, checks that are deemed removable are marked and constraints are attached to them. Checks whose constraints are still valid after the backend processing are removed in the later constraint-checking step.

Redundant Check Elimination. Since the check-instrumentation step inserts a check after the definition of a pointer variable and also before the use of the variable via a load or store, the checks before the uses are redundant at the IR level. Fig. 2 presents an example. Column (a) presents the original C code and column (b) presents the LLVM IR code before check instrumentation. During check instrumentation, three checks are inserted. First, a check is placed at the beginning of the function for the pointer parameter `ptr`. Second, since there are a load in block labeled `else` and a store in block labeled `then`, checks need to be placed before them. `check2` and `check3` are unnecessary at the IR level. However, they cannot be removed in the IR code because `ptr.safe` might be spilled into the untrusted stack during register allocation. Instead, `check2` is marked as removable and a constraint is attached, specifying that the check can be removed if and only if `ptr.safe` is not spilled between `check1` and `check2`. In Fig. 2(c), constraints are at lines starting with the # symbol. `check3` in block `then` is attached with a similar constraint. After register allocation, the constraint checker checks whether `ptr.safe` has been spilled. If not, the two checks are removed. Otherwise, they are kept intact.

LLVM IR is in the SSA form, making it easy to implement the above optimization. First, the def-use chain is explicit in the SSA form. Furthermore, the SSA form ensures that `ptr.safe` is not modified between `check1` and `check2`. By contrast, if carried out on machine code, the optimizer would have to perform dataflow analysis to determine whether a pointer has been guarded and modified.

Sequential Memory Access Optimization. Most programming languages support aggregate types such as structs in C and classes in C++. A common pattern exists in member accesses: a base pointer plus a constant offset is used to visit a specific member. With the guard zones before and after the data region, a memory access with a base pointer and a constant offset is safe as long as the base pointer is within the data region and the offset is smaller than the guard-zone size. This observation can be exploited to remove checks if members of an object

(a) Original C code

```
int foo
(int v, int *ptr)
{
    int tmp = 0;
    if (v > 47)
       *ptr = v;
    else
       tmp = *ptr;
    return tmp;
}
```

(b) Unsecured IR code

```
entry:
   tmp = 0
   if(v > 47) goto then
else:
   tmp = load *ptr
   goto end
then:
   store v, *ptr
end:
   ret tmp
```

(c) Secured and optimized IR code

```
entry:
    ptr.safe = call guard(ptr) // check1
    tmp = 0
    if(v > 47) goto then
else:
    ptr.safe1 = call guard(ptr.safe) // check2
    # noSpill(ptr.safe, check1, check2)
    tmp = load *ptr.safe1
    goto end
then:
    ptr.safe2 = call guard(ptr.safe) // check3
    # noSpill(ptr.safe, check1, check3)
    store v, *ptr.safe2
end:
    ret tmp
```

Figure 2: An example for illustrating redundant check elimination. `guard` is the security check to ensure that the input pointer is in the data region, which is implemented as an LLVM intrinsic function.

(a) Original C code

```
struct s
{
  long x;
  long y;
};
int sum
(struct s *p)
{
  return p->x + p->y;
}
```

(b) Unsecured IR code for `sum`

```
x = gep p, 0, 0
tmp1 = load *x
y = gep p, 0, 1
tmp2 = load *y
sum = add tmp1, tmp2
ret sum
```

(c) Secured and optimized IR code

```
p.safe = call guard(p) // check1
x = gep p.safe, 0, 0
x.safe = call guard(x) // check2
# noSpill(p.safe, check1, check2)
# sizeof(struct s)*0 + sizeof(long)*0 < gz_size
tmp1 = load *x.safe
y = gep p.safe, 0, 1
y.safe = call guard(y) // check3
# noSpill(p.safe, check1, check3)
# sizeof(struct s)*0 + sizeof(long)*1 < gz_size
tmp2 = load *y.safe
sum = add tmp1, tmp2
ret sum
```

Figure 3: An example for illustrating sequential memory access optimization.

are visited sequentially and the base pointer is shared by multiple visits.

In LLVM IR, the `getelementptr` instruction takes a base pointer and multiple indices as operands and is used to compute the address of a sub-element of an aggregate data structure. If the base pointer has been guarded, the offset is a constant, and the offset is determined to be smaller than the guard zone size, then the pointer computed by `getelementptr` does not need to be guarded again. However, the size of each member cannot be determined at the IR level because it may be target dependent. For example, type `long` takes 4 bytes in x86-32 and 8 bytes in x86-64. As a result, the check after the definition of a pointer variable through `getelementptr` can be marked as removable and attached with constraints specifying that the base pointer cannot be spilled and the final offset from the base pointer should be less than the guard-zone size.

Fig. 3 presents an example. In column (a), `struct s` contains two `long` members and the function `sum` computes their sum. In column (c), Strato inserts `check1` at the entry to function `sum` and `check2` and `check3` after each `getelementptr` instruction (abbreviated as `gep` in the figure). The constraints for `check2` specify that it can be removed if there is no spill between `check1` and `check2` for pointer `p.safe` and the offset `sizeof(struct s)*0 + sizeof(long)*0` is smaller than the guard-zone size (this condition can be determined to be true even at the IR level because it is always zero; however, values of other expressions may be target dependent). Similar constraints are attached for `check3`.

Loop-based check optimization. Loop optimization is important because programs tend to spend the majority of runtime in loops. Performance is improved if a security check inside a loop can be hoisted outside the loop. For example, if a pointer is not modified inside

(a) Original C code

```
long sum
(long *ar, long len)
{
  long rst = 0, i;
  for (i=0; i<len; ++i)
   rst += ar[i];
  return rst;
}
```

(b) Unsecured IR code

```
rst = 0
i = 0
if (len <= 0) goto end
for.body:
  ptr = gep ar, i
  tmp = load *ptr
  rst += tmp
  i += 1
  if (i >= len) goto end
  goto for.body:
end:
  ret rst
```

(c) Secured and optimized IR code

```
rst = 0
i = 0
ar.safe = call guard(ar) // check1
if (len <= 0) goto end
for.body:
  ptr = gep ar.safe, i
  ptr.safe = call guard(ptr) // check2
  # noSpill (ar.safe, check1, check2)
  # noSpill (i, check1, check2)
  # sizeof(long) * 1 < gz_size
  tmp = load *ptr.safe
  rst += tmp
  i += 1
  if (i >= len) goto end
  goto for.body:
end:
  ret rst
```

Figure 4: An example for illustrating loop optimization.

the loop, then a check for the pointer can be hoisted. As another example, if a pointer is incremented or decremented for a small stride (less than the guard-zone size) inside the loop, and there is a memory access through the pointer in the loop, then the check can be hoisted. The reason this is safe is because access to the guard zones is trapped; if the initial value of the pointer is checked to be inside the data region, then the change to the pointer in one loop iteration will make the pointer to be either in the data region or in guard zones and the access through the pointer serves as a check. This optimization follows the loop optimization described by Zeng et al. [38]; please refer to that paper for detailed analysis of the soundness of the optimization. As another optimization example, if there is a pointer that is calculated from the induction variable of the loop, the increment to the induction variable is a small stride (less than the guard zone size), and there is a memory access through the pointer in the loop, then the check can be hoisted. LLVM IR encodes loop information explicitly, making it easy to detect induction variables and strides.

In our optimizations, a hoistable check is not moved. Instead, a new check is inserted into the loop preheader and the old check is marked as removable and attached with constraints. An important constraint is that the relevant pointer cannot be spilled. Fig. 4 presents a concrete example. The program in column (a) adds elements in array `ar`. The program visits memory once per iteration. The check can be hoisted if and only if the induction variable is the only variable used to calculate the memory location and the stride is smaller than the guard-zone size and there is a memory visit using the memory location in every path inside the loop.

```
constraint ::= noSpill [var, src, dst]
             | term comparator term
term ::= term + term | term * term |
         var | constant | gz_size
comparator ::= < | > | == | >= | <=
```

Figure 5: Syntax of the constraint language.

Summary about optimizations. The three optimizations demonstrate that a high-level IR can simplify the implementations of optimizations, which are reused among target ISAs. Additional optimizations enabled by a high-level IR can further decrease the runtime cost of Strato.

The constraint language. For completeness, we present the syntax of the constraint language in Fig. 5. A check may be attached with one or more constraints. All constraints need to be satisfied in order for the check to be removed; that is, there is an implicit conjunction when interpreting a list of constraints.

The constraint `noSpill [var, src, dst]` denotes that `var` cannot be spilled between `src` and `dst`, where `src` and `dst` are program locations. Note the semantics of this constraint is that the variable cannot be spilled along *every* control-flow path from `src` to `dst`. Therefore, the `noSpill` constraints in the example of Fig. 5 effectively require that `ar.safe` and `i` cannot be spilled in the entire loop. The *comparison constraint* "`term1 comparator term2`" represents a relation between `term1` and `term2`. It can be used to encode the constraint that a constant offset should be less than the guard-zone size, as in Fig. 3.

The design of the constraint language depends on what optimizations Strato supports and also LLVM's back-

end. For instance, the `noSpill` constraint is there because LLVM's backend may break this assumption by spilling variables to memory. An IR-level optimization may check more conditions. But if there is no possibility for the backend to break a condition, that condition does not need to be encoded and propagated. For instance, in loop optimizations, the condition that there must be a memory access through the pointer in question can be checked at the IR-level alone. Another point is that it is possible new optimizations may require adding new predicates to the constraint language. We believe the constraint language can be extended straightforwardly.

5 Constraint Checking and Check Lowering

After LLVM's backend processing, Strato performs constraint checking and check lowering. Constraint checking examines the constraints attached to each check and checks whether they are valid. If the constraints are valid, the check is removed. If they are invalid, the check is lowered to a sequence of machine instructions.

Our constraint language is designed so that constraints can be checked straightforwardly at the low level, with the help of information preserved by LLVM. For example, a comparison constraint becomes constant expressions at the low level because after fixing the ISA sizes of types become constants. To check a `noSpill` constraint, the constraint checker first identifies the correspondence between IR variables and registers (LLVM preserves enough information for this purpose) and uses data-flow analysis to check whether the register that corresponds to the IR variable is moved to memory between the source and destination locations.

Remaining checks are lowered to machine-instruction sequences. A high-level check can be implemented by many machine-instruction sequences. The overhead of different sequences varies. Strato makes it easy to try different machine-instruction sequences—only the check-lowering step needs to be modified. We have evaluated a large number of machine-instruction sequences for checks in CFI and data sandboxing. We discuss examples of possible sequences next, but leave the discussion about the performance overhead of various sequences to the evaluation section.

ID encodings. Strato's CFI instrumentation requires the encoding of IDs at control-flow targets. ID-encoding instructions have to satisfy two conditions. First, the instruction must take a long immediate value as an operand, which is used to encode the ID. Second, the instruction cannot introduce side-effects that change the semantics of the program. The original CFI uses `prefetch` instructions for encoding IDs. We have evaluated a large number of alternative instructions that satisfy the two conditions. They are put into three groups. Instructions in the first group take an immediate value and assign it to a machine register. For example, "`movl $ID, %eax`" assigns the immediate value `ID` to register `eax`. It can be used to encode the ID as long as `eax` is dead at the point where the instruction is inserted.[3]. Instructions in the second group perform arithmetic operations on a register with the ID and assign the result to a register. For example, "`add $ID, %eax`" can be used to encode the ID as long as `eax` and the flags register are dead. Instructions in the last group take a register and the ID value and defines only the flags register. For example, "`cmp $ID, %eax`" can be used as long as the flags register is dead at the point of insertion.

6 Verification

Compiler optimizations and transformations are untrustworthy because compilers have a large code base and are buggy [26, 36]. Bugs in optimizations that remove security checks are even harder to catch because they do not crash programs but introduce vulnerabilities silently. To remove those optimizations out of the TCB, Strato includes a verifier at the end of the compilation pipeline to validate the assembly output of the compiler.

The verifier checks if the assembly code satisfies the CFI and data sandboxing policy. CFI verification is straightforward. The LLVM assembly preserves enough information to reconstruct a control-flow graph, which is used by the verifier to check if necessary ID-encoding and ID-checking instructions are there in the assembly. Data-sandboxing verification is more challenging because Strato's optimizers may remove or hoist checks. Strato's verifier follows the design of a previous verifier [38] to implement range analysis for data-sandboxing verification (with improvements; see below). The basic idea is to compute the ranges of registers at all program points and check if the ranges of memory addresses fall into the data region plus guard zones. The calculation of ranges uses a standard iterative algorithm until a fixed point is reached.

Strato's verifier improves the previous range-analysis verifier by adding path sensitivity. Ranges of registers may be different along different paths after a sequence of comparison and jump instructions. As an example, the assembly snippet in AT&T syntax in Fig. 6 is extracted from 175.vpr in SPECint2000. The range of register `eax` shrinks down to the data region after the `andl` masking, where `$DATA_MASK` is the constant mask for the data region. The `movl` instruction expands the range of `eax` to `[bottom, top]` because it loads from untrusted memory.[4] Without path sensitivity, the range of `eax` would remain `[bottom, top]` at the entry to the two

```
      andl $DATA_MASK, %eax
      movl (%eax), %eax
      cmpl $3, %eax
      ja .LBB5_8
      movl *.LJTI(,%eax,4), %eax
.LBB2:
      ...
.LBB8:
      ...
```

Figure 6: An example for illustrating path sensitivity in range analysis.

```
popl %ecx
cmpl $ID, 1(%ecx)
jne error
jmpl *%ecx
```

Figure 7: A wrong CFI sequence for return instructions.

successor blocks labeled with `.LBB2` and `.LBB8`; consequently, the verifier would report an out-of-range error on the `movl` instruction because its memory address is "`.LJTI + eax*4`", where .LJTI is a constant in the data region. With path sensitivity, the verifier computes the range of `eax` to be `[0, 3]` before `movl` and successfully validates that the address is within the data region plus guard zones. The verifier in the previous system [38] used instruction pattern matching to verify this pattern. By adopting a path-sensitive analysis, Strato's verifier is more general and can verify all security-check optimizations we have presented.

With the help of the verifier, we discovered 35 subtle bugs in early versions of Strato. Those bugs would be hard to discover otherwise. We classify the bugs into three groups as follows.

- Bugs in CFI instrumentation code. CFI implementation inserts IDs at branch targets and check instructions before computed jumps. The check instructions need to load the IDs at target locations. Therefore, they visit memory and need to be sandboxed as well according to data sandboxing. As an example, the snippet in Fig. 7 contains an early version of a CFI check sequence for return instructions. The return address is popped into register `ecx`, which is then used to load the ID for comparison. The `cmpl` instruction visits the code region using address `ecx+1`, which is unsafe because it is from the untrusted stack. Our verifier successfully caught this bug and we fixed the sequence by inserting a data-masking instruction on `ecx` before `cmpl`.

- Bugs in the source program. Our verifier even found a bug in the source code of `253.perlbmk`, a program in SPECint2000. The bug is a possible null-pointer dereference. The `perlbmk` program has its own `malloc` function, which can return a null pointer when a memory allocation fails and the `malloc` is inlined by the compiler. A null pointer is represented as value 0 and is outside the range of data region plus guard zones. It was caught by the verifier. We modified the source code of `perlbmk` to fix this bug.

- Bugs in LLVM intrinsic functions such as `llvm.memset` and `llvm.memcpy`. LLVM synthesizes programs into intrinsic function calls as an optimization. These intrinsic function calls can be lowered into a sequence of machine instructions or direct calls to the library functions depending on the tradeoff between code size and the function-call overhead. At the IR level, there is no way to predict how an intrinsic function will be lowered. If they are lowered into machine-instruction sequences, then their pointer arguments need to be sandboxed as they may visit memory. Our verifier caught these bugs and we fixed those machine-instruction sequences to insert data-masking instructions for pointers.

The combined verifier for x86-32 and x86-64 contains approximately 7k LOC including white space lines, comments, and debug statements. The majority of the code is a giant switch table for all the machine instructions that the verifier has to support (In LLVM 2.9, x86-64 target contains 3,747 machine instructions). The size of the verifier is a concern and we leave its verification for future work [23].

Finally, we note that our verifier performs verification at the assembly level. We use it mostly to catch bugs in the Strato compiler so that the compiler is out of the TCB. A more desirable design is to implement a verifier for binaries directly. This is actually a matter of engineering: we just need to modify the assembler to encode the control-flow graph as extra information in a binary so that the binary-level verifier can disassemble the binary reliably; the kinds of verification tasks involved are the same after disassembly.

7 Implementation and Evaluation

We have implemented Strato on top of LLVM 2.9. The check-instrumentation step is implemented as a pass and scheduled after LLVM's IR optimization passes. The security-check optimizations are performed right after check instrumentation. The constraint checking and check lowering are implemented in one pass in the

compiler backend after register allocation. The range-analysis based verifier is scheduled at the very end in the backend. Constraints are encoded as LLVM *metadata*. In total, the instrumentation and optimizations consist of approximately 3,750 lines of C++ code, shared between x86-32 and x86-64. The constraint checking and check-lowering component has 1,420 lines of C++ code with an additional 180 lines added for x86-64. The verifier includes 6,960 lines of C++ code, with 1,240 lines added for x86-64.

The object code after data-sandboxing and CFI instrumentation cannot run directly and needs specialized linker scripts and a specialized loader. We have developed linker scripts for both C and C++ programs targeting x86-32 and x86-64. The linker scripts link object code generated by LLVM to three sections (including code, data, and read-only data) at specified addresses. We have also developed a loader that loads various sections in a binary at specified addresses in the address space and sets up appropriate protection for those sections using the `mprotect` system call. We reused PittSFIeld's library wrappers and libraries for x86-32 [21]; we also adapted them for x86-64.

For evaluation, we have built and run the benchmark suites bakeoff and SPECint2000 in Strato. The bakeoff benchmark suite contains three programs: hotlist, lld, and md5. It has been used by previous code-sandboxing frameworks for evaluation [14, 16, 31]. SPECint2000 contains twelve computation-intensive programs and is widely used for compiler evaluation. All benchmark programs in bakeoff and SPECint2000 can be successfully compiled in Strato. All programs were compiled with the `-O3` full optimization level except for 254.gap in SPECint2000, which ran correctly only with `-O0` enabled due to bugs in LLVM 2.9's optimizations. All experiments were conducted on a Ubuntu 11.10 box with an Intel Core 2 Duo CPU at 3.16GHz and 8GB of RAM. Experiments were averaged over three runs and the standard deviation was less than two percent of the arithmetic mean.

Security benefits. Security benefits of Strato depend on what IRMs have been incorporated. The current implementation supports CFI and data sandboxing. The low-level output of all programs in bakeoff and SPECint2000 can be successfully verified by Strato's verifier. Assuming the verifier is correct, the compiled code of those benchmark programs satisfies the CFI and data sandboxing policy.

CFI and data sandboxing come with well-documented security benefits. The original CFI work discusses that CFI can block a test suite of 18 attack vectors, as well as some heap-overflow attacks [3]. Data sandboxing as in SFI is effective in isolating faults in untrusted modules, as reported in NaCl [37] and Robusta [30].

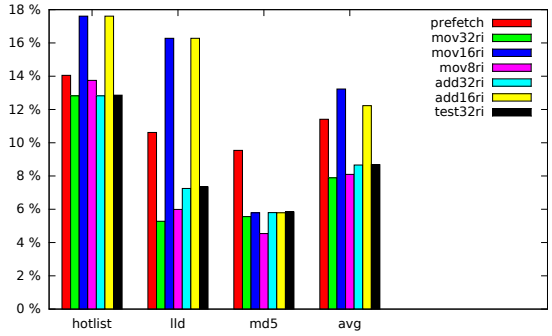

Figure 8: Performance overhead for CFI with various ID-encoding instructions on bakeoff programs.

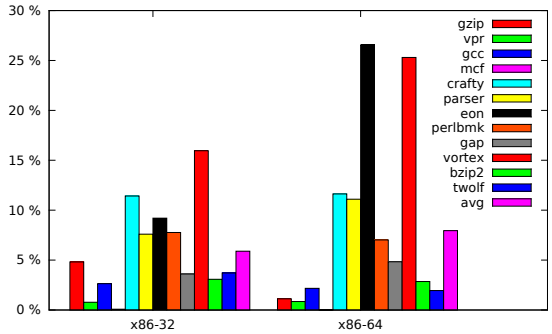

Figure 9: Performance overhead for CFI with `movl` as the ID-encoding instruction on SPECint2000.

CFI and data sandboxing cannot prevent all attacks. For example, non-control data attacks [7] cannot be prevented if critical data are stored inside the sandbox. These attacks can corrupt the data without violating a control-flow graph. However, another IRM called Data-Flow Integrity (DFI [4]) can prevent such attacks.

Performance evaluation. IRMs insert runtime checks into programs and slow down program execution. We present the performance overhead as the percentage of execution-time increase of instrumented programs compared with uninstrumented programs.

We first evaluated the performance implication of alternative machine-instruction sequences that implement the same high-level checks. We have tested a large number of alternative ID-encoding instructions, classified into three groups discussed in Sec 5. Fig. 8 presents the runtime overhead of various ID-encoding instructions on bakeoff programs for x86-32 when enforcing the CFI policy. In the figure, color bars are used for different ID encodings. The figure presents only a subset of what we have tried due to space limit. In addition, we evaluated different ID lengths such as 32-bit IDs, 16-bit IDs and 8-bit IDs. Shorter IDs do not necessarily have better performance and they shrink the space for IDs. In the figure, we use `mov32ri` to represent the case of using a 32-bit `mov` instruction that moves a 32-bit immediate value to a general register. As can be seen from the figure, most ID-encoding instructions are more efficient than `prefetch`,

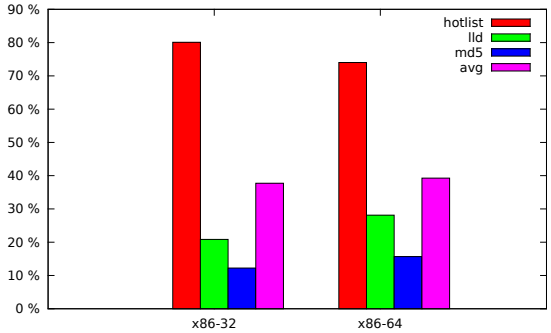

Figure 10: Performance overhead for CFI combined with data sandboxing for both reads and writes on bakeoff.

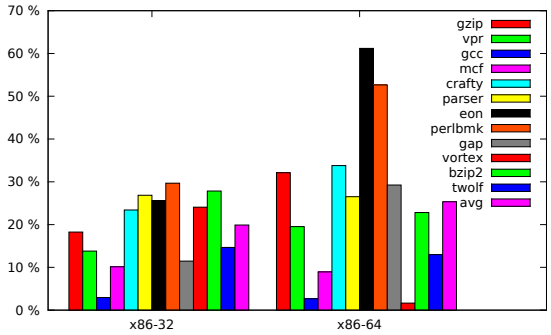

Figure 11: Performance overhead for CFI combined with data sandboxing for both reads and writes on SPECint2000.

the one used in the original CFI implementation.[5] The case of 32-bit `movl` instruction has the lowest runtime overhead.

Fig. 9 presents the performance overhead of CFI alone on SPECint2000 with `movl` as the ID-encoding instruction. CFI incurs an average of 5.89% and 7.95% slowdown on x86-32 and x86-64, respectively. Our CFI implementation is competitive with previous CFI systems. The original CFI work [3] has 15% overhead and our own previous work [38] has 7.7% overhead on x86-32.

Fig. 10 and Fig. 11 present the overhead of enforcing CFI and data sandboxing for bakeoff and SPECint2000 programs, respectively. Both cases of x86-32 and x86-64 are presented. The numbers are with respect to the case of using the `mov32ri` instruction as the ID encoding in CFI, and using the `and` instruction for sandboxing memory addresses. On average, Strato incurs 37.7% on x86-32 and 39.3% on x86-64 for bakeoff programs, and 19.9% on x86-32 and 25.3% on x86-64 for SPECint2000 programs. The high overhead on `hotlist` is due to the two checks in the inner loop of two nested loops and they cannot be optimized away or hoisted.

Strato's performance is competitive with previous SFI/CFI systems. We note most previous systems sandbox only memory writes for protecting integrity, but not memory reads for protecting confidentiality. There are many more memory reads than writes in programs.

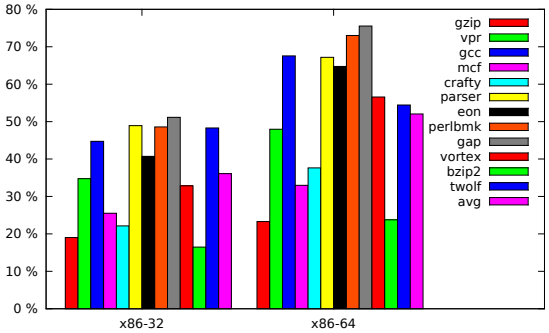

Figure 12: Code size increase on SPECint2000.

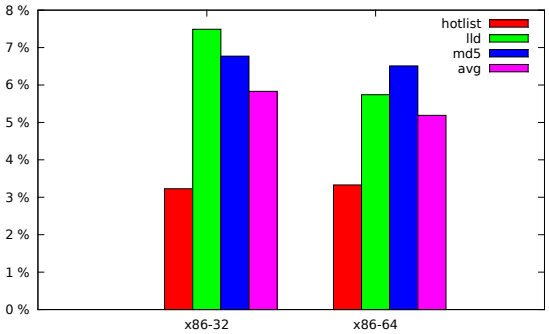

Figure 13: Code size increase on bakeoff.

Strato's data sandboxing protects both memory reads and writes. There are two other SFI/CFI systems that sandbox both reads and writes. XFI's average performance overhead for bakeoff programs are 53.7% on x86-32; in comparison, Strato's overhead is 37.7%. XFI does not report performance results for SPECint2000. Our previous system [38] also sandboxes both reads and writes at the assembly level. It reports an average overhead of 27.2% for x86-32; in comparison, Strato's overhead is 19.9%. We believe that the performance difference is because Strato's optimizations can take advantage of structured information available at the IR level. For instance, the previous system uses the dominator-tree analysis to recover loops and induction variables at the assembly level, while LLVM IR tells explicitly where loops and induction variables are. In summary, Strato provides competitive performance and provides retargetability, lacked by previous systems.

Code Size. We measured the code-size increase on SPECint2000 and bakeoff programs. Strato does not alter the data sections of programs. It increases the size of text sections by inserting extra security checks. Fig. 12 and Fig. 13 present the text-section size increase on SPEC CPU2000 and bakeoff programs, respectively. On average, the text section grows 36.10% on x86-32 and 52.05% on x86-64 for SPECint2000 programs, and 5.83% on x86-32 and 5.19% on x86-64 for bakeoff programs. Text-section size inflates more on SPECint2000 because it contains larger programs with many more

functions; the compiler aligns functions on boundaries. Although disk space is not a major problem in a typical computing environment, it may matter in embedded systems. Benchmark programs were compiled with `-O3`, which is optimized for runtime performance, not for binary size. If the binary size is a major concern, programs can be optimized with `-Os`, which uses shorter instruction sequences.

Memory Usage. We also evaluated the memory usage of Strato. The memory-footprint increase for benchmark programs is negligible since Strato does not change the data memory. It increases memory footprint only through the code-size increase, but the code section takes a small fraction of total runtime memory.

8 Conclusions and Future Work

Conclusions. We have introduced an IRM framework to enforce low-level security policies by working on a high-level intermediate representation. For retargetability, Strato performs its instrumentation and optimizations on a high-level IR, which brings the benefits of structured information and a small instruction set. In addition, we have designed techniques that deal with problems that might arise due to backend transformations and optimizations. A constraint language is proposed to propagate invariants across the backend for validation. Furthermore, a path-sensitive verifier is implemented to verify the final output of the whole framework. Our experimental results show that the framework's performance is competitive with previous sytems. Our framework explores an alternative design point of how low-level IRMs can be implemented. This design point provides retargetability, performance, trustworthiness, and ease of implementation.

Future work. There are many other low-level IRMs that we can incorporate into Strato, including data-flow integrity [4] and fine-grained access control of memory [5]. The workflow of Strato is general enough to accommodate those IRMs, but individual components such as check insertion and lowering need to be updated for a particular IRM. We are interested in designing a check-optimization engine that can be shared by many IRMs; for example, optimizations such as redundant check elimination can be shared. Ultimately, we are interested in generalizing Strato so that it is parametrized by a policy specification, which guides the phases of check insertion, optimization, and lowering.

9 Acknowledgments

We thank Cliff Biffle for providing us with his initial development, useful documents, and suggestions. We also thank Mengtao Sun for his help to the project and anonymous reviewers for their insightful comments. This research is supported by US NSF grants CCF-0915157, CCF-1149211, CCF-1217710, and a research award from Google.

References

[1] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security* (New York, NY, USA, 2005), CCS '05, ACM, pp. 340–353.

[2] Abadi, M., Budiu, M., Erlingsson, Ú., and Ligatti, J. A theory of secure control flow. In *ICFEM* (2005), K.-K. Lau and R. Banach, Eds., vol. 3785 of *Lecture Notes in Computer Science*, Springer, pp. 111–124.

[3] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow integrity: principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur. 13*, 1 (Nov. 2009), 4:1–4:40.

[4] Castro, M., Costa, M., and Harris, T. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006), pp. 147–160.

[5] Castro, M., Costa, M., Martin, J.-P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., and Black, R. Fast byte-granularity software fault isolation. In *ACM SIGOPS Symposium on Operating Systems Principles (SOSP)* (2009), pp. 45–58.

[6] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security* (New York, NY, USA, 2010), CCS '10, ACM, pp. 559–572.

[7] Chen, S., Xu, J., Sezer, E. C., Gauriar, P., and Iyer, R. K. Non-control-data attacks are realistic threats. In *In USENIX Security Symposium* (2005), pp. 177–192.

[8] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (New York, NY, USA, 1989), POPL '89, ACM, pp. 25–35.

[9] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490.

[10] Dean, D., Felten, and Wallach. Java security: From hotjava to netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 1996), SP '96, IEEE Computer Society, pp. 190–.

[11] Dhurjati, D., and Adve, V. S. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE* (2006), pp. 162–171.

[12] Dhurjati, D., Kowshik, S., and Adve, V. S. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI* (2006).

[13] Donovan, A., Muth, R., Chen, B., and Sehr, D. PNaCl: Portable Native Client Executables (white paper). `http://src.chromium.org/viewvc/native_client/data/site/pnacl.pdf`, 2010.

[14] Erlingsson, U., Abadi, M., Vrable, M., Budiu, M., and Necula, G. C. XFI: software guards for system address spaces.

In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 75–88.

[15] ERLINGSSON, U., AND SCHNEIDER, F. B. IRM enforcement of java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2000), SP '00, IEEE Computer Society, pp. 246–.

[16] ERLINGSSON, U., AND SCHNEIDER, F. B. SASI enforcement of security policies: a retrospective. In *Proceedings of the 1999 workshop on New security paradigms* (New York, NY, USA, 2000), NSPW '99, ACM, pp. 87–95.

[17] EVANS, D., AND TWYMAN, A. Flexible policy-directed code safety. In *IEEE Symposium on Security and Privacy (S&P)* (1999), pp. 32–45.

[18] GOVINDAVAJHALA, S., AND APPEL, A. W. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2003), SP '03, IEEE Computer Society, pp. 154–.

[19] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.

[20] LATTNER, C., AND ADVE, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California, Mar 2004).

[21] MCCAMANT, S., AND MORRISETT, G. Evaluating SFI for a CISC architecture. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15* (Berkeley, CA, USA, 2006), USENIX-SS'06, USENIX Association.

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

[23] MORRISETT, G., TAN, G., TASSAROTTI, J., TRISTAN, J.-B., AND GAN, E. Rocksalt: Better, faster, stronger SFI for the x86. In *ACM Conference on Programming Language Design and Implementation (PLDI)* (2012), pp. 395–404.

[24] NAGARAKATTE, S., ZHAO, J., MARTIN, M. M. K., AND ZDANCEWIC, S. Softbound: highly compatible and complete spatial memory safety for C. In *PLDI* (2009), pp. 245–258.

[25] NECULA, G., MCPEAK, S., AND WEIMER, W. CCured: type-safe retrofitting of legacy code. In *29th ACM Symposium on Principles of Programming Languages (POPL)* (2002), pp. 128–139.

[26] REGEHR, J. The future of compiler correctness, August 2010.

[27] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *Annual Computer Security Applications Conference* (2002), pp. 209–218.

[28] SEHR, D., MUTH, R., BIFFLE, C., KHIMENKO, V., PASKO, E., SCHIMPF, K., YEE, B., AND CHEN, B. Adapting software fault isolation to contemporary cpu architectures. In *Proceedings of the 19th USENIX conference on Security* (Berkeley, CA, USA, 2010), USENIX Security'10, USENIX Association, pp. 1–1.

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

[30] SIEFERS, J., TAN, G., AND MORRISETT, G. Robusta: Taming the native beast of the JVM. In *17th ACM Conference on Computer and Communications Security (CCS)* (2010), pp. 201–211.

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

[32] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles* (New York, NY, USA, 1993), SOSP '93, ACM, pp. 203–216.

[33] WANG, Z., AND JIANG, X. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 380–395.

[34] WARTELL, R., MOHAN, V., HAMLEN, K. W., AND LIN, Z. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference* (2012), pp. 299–308.

[35] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in c compilers. *SIGPLAN Not. 46*, 6 (June 2011), 283–294.

[36] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 283–294.

[37] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORM, T., OKASAKA, S., NARULA, N., FULLAGAR, N., AND INC, G. Native client: A sandbox for portable, untrusted x86 native code. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy* (2009).

[38] ZENG, B., TAN, G., AND MORRISETT, G. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *18th ACM Conference on Computer and Communications Security* (Oct. 2011), ACM.

Notes

[1]The name Strato comes from stratosphere, which is an intermediate layer of Earth's atmosphere that contains ozone absorbing ultraviolet light from the Sun.

[2]Return instructions are changed to a sequence of pop, check, and indirect jump instructions to prevent a concurrent attacker from modifying the stack after the check.

[3]`eax` is a caller-saved register and is dead at the entry to a function. Furthermore, dead registers can be identified through liveness analysis

[4]In our implementation, bottom is 0 and top is the maximum unsigned integer, which is $2^{32} - 1$ for x86-32 and $2^{64} - 1$ for x86-64.

[5] Since the original CFI work, some versions of Intel and AMD hardware changed the behavior of `prefetch`; it becomes more expensive, since it pulls in TLB entries. As a result, choice of IDs used in `prefetch` greatly affects its runtime cost.