A Design and Verification Methodology for Secure Isolated Regions

Rohit Sinha[1] Manuel Costa[2] Akash Lal[3] Nuno P. Lopes[2]
Sriram Rajamani[3] Sanjit A. Seshia[1] Kapil Vaswani[2]

[1]University of California at Berkeley, USA
{rsinha,sseshia}@eecs.berkeley.edu

[2]Microsoft Research, UK [3]Microsoft Research, India
{manuelc,akashl,nlopes,sriram,kapilv}@microsoft.com

Abstract

Hardware support for isolated execution (such as Intel SGX) enables development of applications that keep their code and data confidential even while running on a hostile or compromised host. However, automatically verifying that such applications satisfy confidentiality remains challenging. We present a methodology for designing such applications in a way that enables certifying their confidentiality. Our methodology consists of forcing the application to communicate with the external world through a narrow interface, compiling it with runtime checks that aid verification, and linking it with a small runtime library that implements the interface. The runtime library includes core services such as secure communication channels and memory management. We formalize this restriction on the application as *Information Release Confinement* (IRC), and we show that it allows us to decompose the task of proving confidentiality into (a) one-time, human-assisted functional verification of the runtime to ensure that it does not leak secrets, (b) automatic verification of the application's machine code to ensure that it satisfies IRC and does not directly read or corrupt the runtime's internal state. We present /CONFIDENTIAL: a verifier for IRC that is modular, automatic, and keeps our compiler out of the trusted computing base. Our evaluation suggests that the methodology scales to real-world applications.

Categories and Subject Descriptors D.4.6 [*Operating Systems*]: Security and Protection — Information flow controls; D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Programming Languages*]: Specifying and Verifying and Reasoning about Programs

Keywords Enclave Programs; Secure Computation; Confidentiality; Formal Verification

1. Introduction

Building applications that preserve confidentiality of sensitive code and data is a challenging task. An application's secrets can be compromised due to a host of reasons including vulnerabilities in the operating system and hypervisor, and insufficient access control. Recognizing this problem, processor vendors have started to support hardware-based *containers* (such as Intel SGX enclaves [21] and ARM TrustZone trustlets [3]) for isolating sensitive code and data from hostile or compromised hosts. We refer to containers that provide isolation as Secure Isolated Regions (SIR). We assume a powerful adversary who controls the host OS, hypervisor, network, storage and other datacenter nodes as illustrated in Figure 1. Since applications must still rely on the compromised host OS for basic services such as storage and communication, the burden of programming SIRs correctly and ensuring confidentiality remains with the programmer. In this paper, we propose a design methodology and tool support for building code to run in SIRs that can be automatically certified to preserve confidentiality, even in the presence of such a powerful adversary.

Confidentiality can be expressed as an information-flow policy that tracks the flow of secrets through the program, and checks whether secrets may explicitly or implicitly flow to some state that the adversary can observe [7, 8, 16, 31]. Some approaches to certifying confidentiality use programming languages that can express information-flow policies [17, 36, 43]. However, these approaches require use of particular programming languages and incur a heavy annotation burden. More importantly, they place the compiler and the language runtime in the Trusted Computing Base (TCB) [37]. In this paper, we instead explore certifying the machine code loaded into SIRs, to avoid these trust dependencies.

Recent work [40] can provide assurance that machine code satisfies general confidentiality policies, but it does not

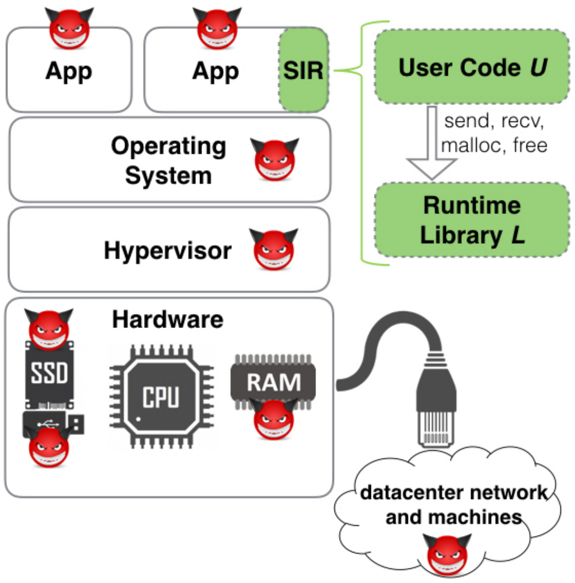

Figure 1. Threat Model: The adversary controls the host OS, hypervisor, and any hardware beyond the CPU package, which may include both RAM chips and storage devices. The adversary also controls all other machines and the network. The SIR is the only trusted software component.

scale to large programs. In this paper, we propose a new methodology for certifying confidentiality that addresses this limitation. We propose to verify a specific confidentiality policy: the code inside the SIR can perform arbitrary computations within the region, but it can only generate output data through an encrypted channel. We refer to this property as *Information Release Confinement* or *IRC*. This is a meaningful confidentiality policy because it guarantees that attackers can only observe encrypted data. We exclude covert channels and side channels (e.g., timing, power) from our threat model. Previous experience from building sensitive data analytics services using SIRs suggests that IRC is not unduly restrictive. For example, in VC3 [38], only map and reduce functions are hosted in SIRs; the rest of the Hadoop stack is untrusted. Both mappers and reducers follow a stylized idiom where they receive encrypted input from Hadoop's untrusted communication layer, decrypt the input, process it, and send encrypted output back to Hadoop. No other interaction between these components and the untrusted Hadoop stack is required.

Scalably verifying IRC is challenging, because we aim to have a verification procedure that can automatically certify machine code programs, without assuming the code to be type safe or memory safe; for example, programs may have exploitable bugs or they may unintentionally write information out of the SIR through corrupted pointers. Our approach is based on decomposing programs into a user application (U) that contains the application logic and a small runtime library (L) that provides core primitives such as memory management and encrypted channels for communication. We restrict the interaction between the user application and the untrusted platform to the narrow interface implemented by L. A key contribution of this paper is how this methodology enables decomposing the confidentiality verification in two parts. For L, we need to verify that it implements a secure encrypted channel and memory management correctly; L is a small library that can be written and verified once and for all. For U, we need to verify a series of constraints on memory loads, stores, and control-flow transitions. Specifically, we need to check that stores do not write data outside the SIR, stores do not corrupt control information (e.g., return addresses and jump tables) inside the SIR, stores do not corrupt the state of L, loads do not read cryptographic state from L (since using cryptographic state in U could break the safety of the encrypted channel [9]), calls go to the start of functions in U or to the entry points of API functions exported by L, and jumps target legal (i.e. not in the middle of) instructions in the code. These checks on U are a weak form of control-flow integrity, and along with restrictions on reads and writes, enforce a property which we call WCFI-RW. We show that the functional correctness of L combined with WCFI-RW of U implies our desired confidentiality policy (IRC).

In this paper, we formalize WCFI-RW and propose a two-step process to automatically verify that an application satisfies WCFI-RW. We first use a compiler that instruments U with runtime checks [38]. Next, we automatically verify that the instrumentation in the compiled binary is sufficient for guaranteeing WCFI-RW. Our verifier generates verification conditions from the machine code of the application, and automatically discharges them using an SMT solver. This step effectively removes the compiler as well as third-party libraries from the TCB. By verifying these libraries, users can be certain that they do not leak information either accidentally, through exploitable bugs, or by intentionally writing data out of the SIRs.

Our runtime checks are related to previous work on software fault isolation (SFI [27, 39, 44, 47]), but we note that a simple SFI policy of sandboxing all the code in the SIR would not be enough to guarantee IRC, because U and L share the same memory region. Our checks implement a form of fine-grained memory protection inside the SIR, which we use to separate the state of U and L and guarantee the integrity of control data. Moreover, our checks work well for SGX enclaves on x64 machines (our target environment), whereas previous work would require significant modifications to work efficiently in this environment. One of our key contributions is showing that the instrumentation of U, together with the properties of L, guarantees IRC; this work is the first that verifies and guarantees IRC using such instrumentation.

Our approach is significantly different from verifying arbitrary user code with unconstrained interfaces for communication with the untrusted platform. We require no annotations from the programmer — all of U's memory is considered secret. The TCB is small — it includes the verifier but does not include U or the compiler. Furthermore, the assertions required to prove WCFI-RW are simple enough to be discharged using an off-the-shelf Satisfiability Modulo Theories (SMT) solver. The verification procedure is modular, and can be done one procedure at a time, which enables the technique to scale to large programs. Our experiments suggest that our verifier can scale to real-world applications, including map-reduce tasks from VC3 [38].

In summary, the main contributions of this paper are: (1) a design methodology to program SIRs using a narrow interface to a library, (2) a notion called Information Release Confinement (IRC), which allows separation of concerns while proving confidentiality in the presence of a privileged adversary (that controls the OS, hypervisor, etc.), and (3) a modular and scalable verification method for automatically checking IRC directly on application binaries.

2. Overview

Secure Isolated Regions. Protecting confidentiality of applications using trusted processors is a topic of wide interest [3, 24, 28, 34]. These processors provide primitives to create memory regions that are isolated from all the other code in the system, including operating system and hypervisor. We refer to such an abstraction as a *Secure Isolated Region* or *SIR*. The processor monitors all accesses to the SIR: only code running in the SIR can access data in the SIR. As an example, Intel's SGX instructions enable the creation of SIRs (called "enclaves") in the hosting application's address space. The SIR can access the entire address space of the hosting application, which enables efficient interaction with the untrusted platform. External code can only invoke code inside the region at statically-defined entry-points (using a call-gate like mechanism). The processor saves and restores register context when threads exit and resume execution inside the SIR. To protect against physical attacks on the RAM, the processor also encrypts cache lines within the SIR on eviction; the cache lines are decrypted and checked for integrity and freshness when they are fetched from physical memory. SGX also supports attestation and sealing. Code inside an SIR can get messages signed using a per processor private key along with a cryptographic digest of the SIR. This enables other trusted entities to verify that messages originated from the SIR. Primitives to create SIRs can also be provided by hypervisors [11, 20, 45], with the caveat that the hypervisor becomes part of the TCB, but in this paper we assume SIRs are provided directly by the processor. Regardless of the primitives and infrastructure used to implement SIRs, application developers who write the code that runs inside SIRs are responsible for maintaining confidentiality of secrets managed by the SIR. The goal of this paper is to provide a methodology and tool support for helping application developers ensure confidentiality.

Threat Model We assume a user that wishes to protect the confidential data processed by an application U. The application runs inside an SIR in a hostile or compromised host. We assume that U is not designed to write confidential data outside the SIR. However, U may have bugs such as accidental writes of confidential data to non-SIR memory, as well as exploitable bugs, such as buffer overflows, use-after-free, and dereferences of uninitialized or corrupted pointers, which could result in confidential data leaking out of the SIR. U may also include third-party libraries that intentionally try to write confidential data outside the SIR. Therefore, we treat U's code as untrusted and verify that U cannot leak secrets even if it has exploitable bugs.

The illustration of our threat model in Figure 1 lists all the system components that are under the attacker's control. The adversary may fully control all of the hardware in the host computer, including disks, network cards, and all the chips (including RAM chips) in the system, except the processor that runs the SIR. The adversary may record, replay, and modify network packets or files, as well as read or modify data after it leaves the processor chip using physical probing, direct memory access (DMA), or similar techniques. We assume that the adversary cannot physically attack the processor to extract secrets. The adversary may also control all of the system software in the host computer, including the operating system and the hypervisor. This adversary is general enough to model privileged malware running in the kernel, as well as a malicious system administrator who may try to access the data by logging into the host and inspecting disks and memory. Denial-of-service attacks, side-channel attacks, and covert-channel attacks are outside the scope of this paper.

Verifying confidentiality. We illustrate the challenges in verifying that code running inside an SIR satisfies confidentiality. Consider the `Reduce` method in Figure 2, which acts as a reducer in a map-reduce job. The reducer receives encrypted a key and list of values from different mappers. The method first provisions an encryption key (of type `KeyAesGcm`, not to be confused with the data key) by setting up a secure channel with a component that manages keys (not shown for brevity). It decrypts the key and values, computes the sum of all values, and writes the output to a buffer. The buffer is encrypted and written to a location outside the region specified by the map-reduce framework.

Proving that this `Reduce` method preserves confidentiality is challenging. The code writes the result of the computation to a stack-allocated buffer without checking the size of the inputs. This may result in a vulnerability that can be exploited to overwrite the return address, execute arbitrary code and leak secrets. Therefore, the proof must show that the cumulative size of key and result does not exceed the buffer size. Such a proof may require non-modular reasoning since the sizes may not be defined locally. Furthermore, `Reduce` writes to a location outside the SIR. The proof must show that the data written is either encrypted or independent of secrets. The latter requires precise, fine-grained tracking of secrets in SIR's memory. Also, unrelated to the application logic, we note that the `Reduce` method manually pro-

```
void Reduce(BYTE *keyEnc, BYTE *valuesEnc, BYTE *outputEnc) {
  KeyAesGcm *aesKey = ProvisionKey();

  char key[KEY_SIZE];
  aesKey->Decrypt(keyEnc, key, KEY_SIZE);

  char valuesBuf[VALUES_SIZE];
  aesKey->Decrypt(valuesEnc, valuesBuf, VALUES_SIZE);
  StringList *values = (StringList *) valuesBuf;

  long long usage = 0;
  for (char *value = values->begin();
    value != values->end(); value = values->next()) {
    long lvalue = mystrtol(value, NULL, 10);
    usage += lvalue;
  }

  char cleartext[BUF_SIZE];
  sprintf(cleartext, "%s %lld", key, usage);
  aesKey->Encrypt(cleartext, outputEnc, BUF_SIZE);
}
```

(a) Sample reducer method

```
void Reduce(Channel<char*>& channel) {
  char key[KEY_SIZE];
  channel.recv(key, KEY_SIZE);

  char valuesBuf[VALUES_SIZE];
  channel.recv(valuesBuf, VALUES_SIZE);
  StringList *values = (StringList *) valuesBuf;

  long long usage = 0;
  for (char *value = values->begin();
    value != values->end(); value = values->next()) {
    long lvalue = mystrtol(value, NULL, 10);
    usage += lvalue;
  }

  char cleartext[BUF_SIZE];
  sprintf(cleartext, "%s %lld", key, usage);
  channel.send(cleartext);
}
```

(b) Reducer method using secure channels

Figure 2. Reducer method illustrating the challenges of proving confidentiality.

visions its encryption keys. Therefore, a proof of confidentiality must also show that the key exchange protocol is secure, and that the keys are neither leaked by the application, nor overwritten by an adversary. Finally, the proof must also show that the compilation to machine code preserves semantics of source code. Therefore, building a scalable and automatic verifier for confidentiality is challenging for arbitrary user code.

Restricted interface. We propose a design methodology to separate various concerns in ensuring confidentiality, and enable simpler and scalable tools to automatically verify confidentiality. In our methodology, the user application U is statically linked with a runtime library L that supports a narrow communication interface. During initialization, the runtime is configured to setup a secure channel with another trusted entity and provision secrets, e.g., the encryption key. The user application can use runtime APIs to allocate memory (via `malloc` and `free`) and send or receive data over the secure channel (via `send` and `recv`). Figure 2(b) shows the `Reduce` method that has been rewritten to use this interface. The method calls `recv` to retrieve data from the channel, which reads and decrypts encrypted values from outside the SIR. After computing the result, the method calls `send` to write data to the channel, which internally encrypts the data and writes it to a location outside the SIR. Observe that there are no writes to non-SIR memory directly from this `Reduce` method.

Restricting the application to this interface serves an important purpose — it allows us to decompose the task of verifying confidentiality into two sub-tasks: (1) checking that the user application U communicates with the external world *only* through this interface, and (2) checking that the implementation of the interface in L does not leak secrets. This paper focuses on Information Release Confinement (IRC): U can only write to non-SIR memory by invoking the `send` API. Proving information leakage properties of the implementation of `send` (e.g., strong encryption property, resistance to various side channels, etc.) would require one-time human-assisted verification of the library code L, and is left for future work. We feel that IRC is an important stepping stone for achieving confidentiality guarantees of SIRs against various adversaries.

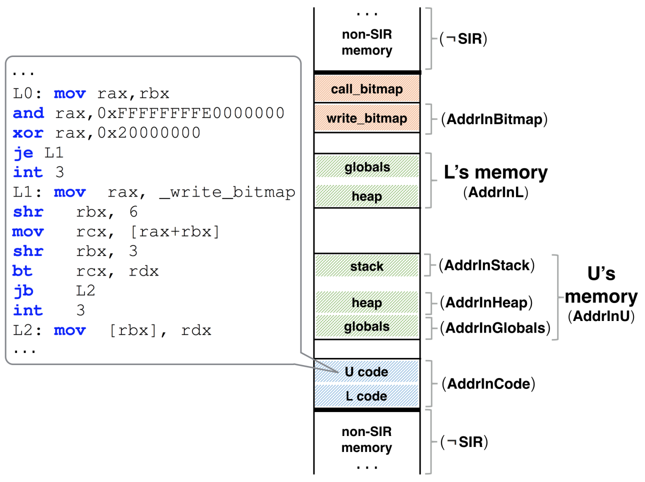
```
...
L0: mov rax,rbx
and rax,0xFFFFFFFFE0000000
xor rax,0x20000000
je L1
int 3
L1: mov  rax, _write_bitmap
shr   rbx, 6
mov   rcx, [rax+rbx]
shr   rbx, 3
bt    rcx, rdx
jb    L2
int   3
L2: mov  [rbx], rdx
...
```

Figure 3. Memory layout of SIR and instrumentation sequence for unsafe stores in the application code (U).

We do not find this narrow interface to be restrictive for our target applications: trusted cloud services, which are typically implemented as a collection of distributed and trusted entities. In this setting, the application (U) only sends and receives encrypted messages from remote parties. We use this approach in practice to build sensitive data analytics, database services, key manager, etc.

Checking IRC. For scalable and automatic verification, we further decompose IRC into a set of checks on U and contracts on each of the APIs in L. First, we verify that U satisfies WCFI-RW: U transfers control to L only through the API entry points, and it does not read L's memory or write outside U's memory. Apart from the constraints on reads and writes, WCFI-RW requires a weak form of control-flow integrity in U. This is needed because if an attacker hijacks the control flow of U (e.g., by corrupting a function pointer or return address), then U can execute arbitrary instructions, and we cannot give any guarantees statically. Next, we identify a small contract on L that, in conjunction with WCFI-RW property of U, is sufficient for proving IRC. The contracts on L (defined in section 4.2) ensure that only `send` gets to write to non-SIR memory, and that L does not modify U's state to an extent that WCFI-RW is compromised.

From here on, we describe how we check WCFI-RW directly on the machine code of U, which enables us to keep the compiler and any third-party application libraries out of the TCB. Third-party libraries need to be compiled with our compiler, but they can be shipped in binary form. Furthermore, to help the verification of WCFI-RW, our (untrusted) compiler adds runtime checks to U's code, and we use the processor's paging hardware to protect L's memory — although the attacker controls the page tables, the processor protects page table entries that map to SIR memory. We use the same runtime checks on memory writes and indirect calls as VC3 [38], but we implement efficient checks on memory reads using the paging hardware.

Next, we describe the checks done by the VC3 instrumentation. We first note that U and L share the same address space (Figure 3). The code segments of both U and L are placed in executable, non-writable pages. The region also contains a stack shared by both U and L. We isolate L from U by using a separate heap for storing L's internal state. The compiler enforces WCFI-RW by instrumenting U with the following checks:

- **Protecting return addresses**: To enforce that writes through pointers do not corrupt return addresses on the stack, the compiler maintains a bitmap (`write_bitmap` in Figure 3) to record which areas in U's memory are writable. The `write_bitmap` is updated at runtime, while maintaining the invariant that a return address is never marked writable. Address-taken stack variables are marked writable by inlined code sequences in function prologues, and heap allocations and address-taken globals are marked writable by the runtime library. `store` instructions are instrumented with an instruction sequence (instructions from label L1 to L2 in Figure 3) that reads the `write_bitmap` and terminates the SIR program if the corresponding bit is not set. Note that the bitmap protects itself: the bits corresponding to the bitmap are never set.
- **Protecting indirect control flow transfers**: The compiler maintains a separate bitmap (`call_bitmap` in Figure 3) that records the entry points of procedures in U and APIs of L. The compiler instruments indirect calls with an instruction sequence that reads the bit corresponding to the target address, and terminates the SIR program if that bit is unset. Indirect jumps within the procedure are also checked to prevent jumps to the middle of instructions; the checks consist of a range check on indices into jump tables (which are stored in read-only memory).
- **Preventing writes outside SIR**: The compiler adds range checks to prevent writes to non-SIR memory (instructions from label L0 to L1 in Figure 3).

We note that the VC3 instrumentation offers a stronger guarantee than IRC, and prevents several forms of memory corruption inside the SIR. Specifically, the checks guarantee: 1) integrity of all data that is not address-taken, 2) detection of all sequential overflows and underflows on heap and stack, 3) integrity of return addresses, and 4) forward edge CFI. Future work may be able to verify of all these properties, but for this work we focus on verifying that the VC3 instrumentation is sufficient to guarantee WCFI-RW, because WCFI-RW together with the correctness of L implies IRC (as we show in Theorem 1), and IRC is a strong property that provides meaningful security guarantees.

Preventing U from accessing L's memory is also important because L keeps cryptographic secrets; allowing U to read such secrets could break the typical assumptions of encryption algorithms (e.g., a key should not be encrypted with itself [9]). We achieve this efficiently by requiring L to store any such secrets in its own separate heap, and using SGX memory protection instructions [21] to set page permissions to disable read/write access before transferring control to U. Note that we cannot use page permissions to prevent writes to non-SIR memory, because those page tables are controlled by a privileged adversary, e.g., kernel-level malware.

Let's reconsider the `Reduce` method in Figure 2(b). Compiling this method with the preceding runtime checks and necessary page permissions ensures that any vulnerability (e.g., line 17) cannot be used to violate WCFI-RW. Any such violation causes the program to halt.

To achieve good performance, the compiler omits runtime checks that it can statically prove to be redundant (e.g., writes to local stack-allocated variables). However, compilers are large code bases and have been shown to have bugs and produce wrong code at times [6, 23, 26, 29, 46]. The rest of the paper describes our approach for automatically verifying that the machine code of U satisfies WCFI-RW, thereby removing the compiler from the TCB. We also show that such a U satisfies IRC when linked with an L that satisfies our contract (defined in section 4.2).

3. Formal Model of User Code and Adversary

In this section we define a language, as shown in Figure 4, to model the user code U. Each machine code instruction in U is translated to a sequence of statements (*Stmt*) in the presented language. This translation is intended to reduce the

$$
\begin{array}{lcl l l}
\mathsf{v} & \in & \mathit{Vars} & & \\
c & \in & \mathit{Constants} & & \\
q & \in & \mathit{Relations} & & \\
f & \in & \mathit{Functions} & ::= & \mathsf{add} \mid \mathsf{xor} \mid \mathsf{extract} \mid \mathsf{concat} \mid \ldots \\
e & \in & \mathit{Expr} & ::= & \mathsf{v} \mid c \mid f(e,\ldots,e) \\
\phi & \in & \mathit{Formula} & ::= & \mathsf{true} \mid \mathsf{false} \mid e = e \mid \\
 & & & & q(e,\ldots,e) \mid \phi \wedge \phi \mid \neg\phi \\
s & \in & \mathit{Stmt} & ::= & \mathsf{store_n}(e,e) \mid \mathsf{v} := \mathsf{load_n}(e) \mid \\
 & & & & \mathsf{v} := e \mid \mathtt{jmp}\ e \mid \mathtt{call}\ e \mid \mathsf{ret} \mid \\
 & & & & \mathtt{assert}\ \phi \mid \mathtt{assume}\ \phi \mid \mathsf{havoc}\ \mathsf{v} \mid \\
 & & & & \mathsf{skip} \mid s \diamond s \mid s\ ;\ s
\end{array}
$$

Figure 4. Syntax of U code.

complexity in directly modeling machine code for complex architectures (such as x64), and instead leverage tools that lift machine code to simpler, RISC-like instruction sets [10]. Furthermore, U can be thought of as a set of procedures (with a unique entry point), where each procedure is a sequence of statements $s_0; s_1; \ldots; s_n$. For this paper, we also assume that execution within an SIR is single-threaded.

Variables in Vars consist of `regs`, `flags`, and `mem`. `regs` are CPU registers (e.g., rax, r8, rsp, rip, etc.) that are 64-bit values in the case of x64. CPU flags (e.g., CF, ZF, etc.) are 1-bit values. The instruction pointer (rip) stores the address of the next instruction to be executed and is incremented automatically after every instruction except in those that change the control flow. Memory (mem) is modeled as a map from 64-bit bit-vectors to 8-bit bit-vectors.

Memory accesses are encoded using $\mathsf{load_n}$ and $\mathtt{store_n}$ functions, where n denotes the size of the data in bytes. $\mathsf{load_n}$ and $\mathtt{store_n}$ are axiomatized using SMT's theory of arrays. Bit-vector operations (add, sub, etc) are axiomatized using SMT's bit-vector theory.

Assignment statements can be one of following two forms: (1) $\mathsf{v} := e$ sets $\mathsf{v} \in \mathit{Vars}$ to the value of expression e, (2) $\mathsf{reg} := \mathsf{load_n}(e)$ sets $\mathsf{reg} \in \mathtt{regs}$ to the value of the memory at address e. Statement `assume` ϕ blocks when executed in a state that does not satisfy ϕ, and is equivalent to a no-op when executed in a state that satisfies ϕ. Executing `assert` ϕ in a state that does not satisfy ϕ leads to an error.

Control flow is changed with `call`, ret, and `jmp` statements, which override the value of rip. A procedure call returns to the following instruction (we verify a form of control flow integrity to guarantee this). The ret statement terminates execution in the procedure and returns back to the caller. Both `call` and ret are semantically equivalent to the x64 call and return instructions, respectively — `call` pushes the return address on the stack, and the ret instruction pops the stack before returning. `jmp` e encodes a jump to an arbitrary location, either in the current procedure or the beginning of a procedure (as an artifact of tail call optimizations).

The choice statement $s \diamond t$ non-deterministically executes either s or t. Choice statements together with assume statements are used to model conditional statements. A typical conditional statement if (ϕ) $\{s\}$ else $\{t\}$ is modeled as $\{\mathtt{assume}\ \phi\ ;\ s\} \diamond \{\mathtt{assume}\ \neg\phi\ ;\ t\}$.

The havoc v statement assigns a fresh, symbolic value to the variable $\mathsf{v} \in \mathit{Vars}$. When havocing memory variables, a havoc statement may optionally specify a predicate to characterize which memory locations are modified; for instance, havoc_ϕ `mem` statement scrambles all memory locations specified by predicate ϕ. Intuitively, this creates a new memory $\mathtt{mem}'$ whose content is the same as in memory `mem` for all addresses that do not satisfy ϕ, as follows:

$$
\begin{aligned}
&\mathtt{assume}\ \forall a.\ \neg\phi(a) \Rightarrow \mathtt{mem}'[a] = \mathtt{mem}[a];\\
&\mathtt{mem} := \mathtt{mem}'
\end{aligned}
$$

We note that the user code U does not contain havoc statements. However, we use the havoc statement to model actions of the adversary, as we show later.

Software interrupts (e.g., int 3 instruction) terminate the execution of a SIR; we model them using `assume false`. Other exceptions (such as division by zero, general protection faults, etc.) also cause the SIR to terminate, which simplifies both the modeling and verification. We do not find this to be a limitation in our experience writing SIR programs.

We define state σ to be a valuation of all variables in Vars. Let $\sigma(\mathsf{v})$ be the value of a variable $\mathsf{v} \in \mathit{Vars}$ in state σ and similarly let $\sigma(e)$ be the valuation of expression e in state σ. Let $\mathtt{stmt}(\sigma)$ be the statement executed in state σ (computed from the instruction pointer and the code in memory). The semantics of a statement $s \in \mathit{Stmt}$ is given by relation $\mathcal{R}$ over pairs of pre and post states, where $(\sigma, \sigma') \in \mathcal{R}$ if and only if $s = \mathtt{stmt}(\sigma)$ and there is an execution of s starting at σ and ending in σ'. We define operational semantics for a subset of Stmt in Figure 5, and use standard semantics for the remaining statements. A sequence $\pi = [\sigma_0, \ldots, \sigma_n]$ is called an *execution trace* if $(\sigma_i, \sigma_{i+1}) \in \mathcal{R}$ for each $i \in \{0, \ldots, n-1\}$. We also use $\mathtt{stmt}(\pi)$ to denote the sequence of statements executed in π. Furthermore, in order to decouple the verification of U from the verification of L, we only let π to "step into" procedures of U, i.e., procedures in L's code are executed atomically and therefore only contribute with one state transition to execution traces.

The initial state σ_{entry} is the result of a jump from L's code into U's entry point. Prior to the jump, L performs the necessary initialization to support the memory management (`malloc` and `free`) and communication (`send` and `recv`) services. For instance, L may engage in a key exchange protocol with remote entities to establish the key used by `send` and `recv`. σ_{entry} assigns an arbitrary value to addresses of `mem` that include the non-SIR memory, U's stack, and U's heap. σ_{entry} also assigns an arbitrary value to all `regs`, `flags`, except for the stack pointer rsp which points to U's stack.

Modeling the adversary Our formal model of the adversary is similar to Moat [40]. The adversary may force the host to transfer control from the SIR to the adversary's code at any time during the execution of the SIR (by generating

$$\langle \mathsf{store_n}(e_a, e_d), \sigma\rangle \Downarrow \sigma\Big[\mathtt{mem} \mapsto \sigma(\mathtt{mem})[\sigma(e_a) := \sigma(e_d)]\Big]$$

$$\langle \mathsf{reg} := \mathsf{load_n}(e_a), \sigma\rangle \Downarrow \sigma\Big[\mathsf{reg} \mapsto \sigma(\mathtt{mem})[\sigma(e_a)]\Big]$$

$$\langle \mathsf{reg} := e, \sigma\rangle \Downarrow \sigma\Big[\mathsf{reg} \mapsto \sigma(e)\Big]$$

$$\langle \mathtt{jmp}\ e, \sigma\rangle \Downarrow \sigma\Big[\mathsf{rip} \mapsto \sigma(e)\Big]$$

$$\begin{aligned}\langle \mathtt{call}\ e, \sigma\rangle \Downarrow \sigma\Big[&\mathsf{rip} \mapsto \sigma(e),\\ &\mathsf{rsp} \mapsto \sigma(\mathsf{rsp} - 8),\\ &\mathtt{mem} \mapsto \sigma(\mathtt{mem})[\sigma(\mathsf{rsp} - 8) := \mathsf{next}(\sigma(\mathsf{rip}))]\Big]\end{aligned}$$

$$\begin{aligned}\langle \mathsf{ret}, \sigma\rangle \Downarrow \sigma\Big[&\mathsf{rip} \mapsto \sigma(\mathtt{mem})[\sigma(\mathsf{rsp})],\\ &\mathsf{rsp} \mapsto \sigma(\mathsf{rsp} + 8)\Big]\end{aligned}$$

Figure 5. Operational semantics of $s \in \mathit{Stmt}$: $(\sigma, \sigma') \in \mathcal{R}$ iff $\langle s, \sigma\rangle \Downarrow \sigma'$ and $\mathtt{stmt}(\sigma) = s$. $\sigma\Big[\mathsf{x} \mapsto \mathsf{y}\Big]$ denotes a state that is identical to σ, except variable x evaluates to y. The memory update expression $\mathtt{mem}[\mathsf{x} := \mathsf{y}]$ returns a new memory that is equivalent to $\mathtt{mem}$, except for index x — multibyte-sized accesses follow the processor's endianness semantics. $\mathsf{next}(e)$ is the address of the subsequent instruction in U after decoding the instruction starting at address e.

an interrupt, for example). Once the CPU transfers control from the SIR to the adversary, the adversary may execute an arbitrary sequence of instructions before transferring control back to the SIR. The adversarial operations include arbitrary updates to non-SIR memory, privileged state accessible to the OS and hypervisor layers (e.g. page tables), registers, and devices. Moat defines the following active adversary $\mathcal{H}$, which only havocs non-SIR memory, and proves a theorem that $\mathcal{H}$ models all adversarial operations in our threat model.

DEFINITION 1. **Havocing Active Adversary $\mathcal{H}$.**
Between any consecutive statements in an execution of U, $\mathcal{H}$ may observe and perform a single havoc *on* $\mathtt{mem}_{\neg\mathtt{SIR}}$*, where* $\mathtt{mem}_{\neg\mathtt{SIR}}$ *is the set of locations outside the SIR boundary* $(\neg\mathtt{SIR})$*.*

Composing U with active adversary $\mathcal{H}$ We transform each statement s within U to:

$$\mathsf{havoc}_{\neg\mathsf{SIR}}\ \mathtt{mem};\ s \qquad (1)$$

This composed model, hereby called $U_{\mathcal{H}}$, encodes all possible behaviors of U in the presence of $\mathcal{H}$.

4. Formalizing Confidentiality

Confidentiality is typically defined as a hyper-property [12], where we require that the adversary cannot infer secret state based on observations. Automatically checking if a program satisfies confidentiality usually requires precise tracking of the memory locations that may contain secrets during execution. This is accomplished with either a whole-program analysis of $U_{\mathcal{H}}$, which is hard to scale for machine code, or fine-grained annotations which specify locations with secrets, which is cumbersome for machine code [40]. We follow a different approach in order to scale the verification to large programs without requiring any annotation.

We expect $U_{\mathcal{H}}$ to communicate with non-SIR entities, but follow a methodology where we mandate all communication to occur via the `send` and `recv` APIs in L. We require (and verify) that $U_{\mathcal{H}}$ does not write to non-SIR memory. Instead, $U_{\mathcal{H}}$ invokes `send`, which takes as an argument a pointer to the input buffer, and encrypts and integrity-protects the buffer before copying out to non-SIR memory. This methodology leads to a notion called *information release confinement* (IRC), which mandates that the only statements in π that update non-SIR memory (apart from $\mathcal{H}$'s havoc operations) are the $\mathtt{call}$ send statements.

DEFINITION 2. **Information Release Confinement.** *An execution trace $\pi = [\sigma_0, \ldots, \sigma_n]$ of $U_{\mathcal{H}}$ satisfies information release confinement or IRC, if all updates to the adversary observable state (i.e.,* $\mathtt{mem}_{\neg\mathtt{SIR}}$*) in π are caused by either* $\mathtt{call}$ send *statements or* $\mathsf{havoc}_{\neg\mathsf{SIR}}$ $\mathtt{mem}$ *operations (from $\mathcal{H}$), i.e., IRC(π) iff:*

$$\begin{aligned}&\forall\, i \in \{0, \ldots, n-1\}\ .\\ &\quad (\mathtt{stmt}(\sigma_i) \neq \mathtt{call}\ \mathsf{send}\ \wedge \mathtt{stmt}(\sigma_i) \neq \mathsf{havoc}_{\neg\mathsf{SIR}}\ \mathtt{mem}) \Rightarrow\\ &\quad (\forall\, a.\ \neg\mathtt{SIR}(a) \Rightarrow \sigma_i(\mathtt{mem})[a] = \sigma_{i+1}(\mathtt{mem})[a]) \qquad (2)\end{aligned}$$

$U_{\mathcal{H}}$ satisfies IRC iff all traces of $U_{\mathcal{H}}$ satisfy IRC.

IRC and Confidentiality. IRC is an important building block in guaranteeing confidentiality. It ensures that, in any execution, the only outbound communication with the environment is via `send`. Hence, we can arrange for `send` to encrypt all its received data before transmitting it, to prevent explicit information leaks. In order for the encrypted data to be confidential, we additionally need to ensure that the encryption key in L does not leak or gets overwritten. The definition of IRC enables us to separate properties we require from the application code $U_{\mathcal{H}}$, and properties we require from L, in order to guarantee confidentiality.

It is important to note that IRC is not sufficient for protecting secrets from all side channels. Observations of the number and timing of `send` invocations, memory access patterns, electromagnetic radiation, etc. potentially reveal secrets. Nevertheless, if an application $U_{\mathcal{H}}$ satisfies IRC, then we can eliminate certain channels and obtain various degrees of confidentiality by imposing additional constraints on the implementation of `send`. We can arrange for `send` to output messages with constant-sized buffers (using appropriate padding) to prevent the adversary from making any inference based on message sizes. In addition, we can arrange for `send` to do internal buffering and produce sequences of output messages that are separated by an interval that is independent of secrets, to prevent the adversary from making any inference based on timing. These defenses impose additional constraints only on the implementation of `send`. We plan to explore guaranteeing such properties of our `send` implementation in future work.

In the remainder of this section, we formalize the properties on both $U_{\mathcal{H}}$ and L, such that the SIR satisfies the IRC property. To decouple the verification, we decompose IRC into 1) checking WCFI-RW of $U_{\mathcal{H}}$, and 2) checking correctness properties of L's API implementation.

4.1 WCFI-RW Property of $U_{\mathcal{H}}$

WCFI-RW further decomposes into the following two properties:

(a) A weak form of control flow integrity (CFI) of $U_{\mathcal{H}}$. A trace π of $U_{\mathcal{H}}$ satisfies weak CFI if 1) each `call` statement in π targets the starting address of a procedure in U or API entry point of L, 2) each ret statement in π uses the return address saved by the matching `call` statement in π, 3) each `jmp` statement in π targets a legal instruction within the procedure or the starting address of a procedure in U.

(b) $U_{\mathcal{H}}$ does not read from or write to L's memory, and does not write to non-SIR memory.

WCFI-RW guarantees that $U_{\mathcal{H}}$ only calls into L at allowed API entrypoints, which allows us to soundly decouple the verification of $U_{\mathcal{H}}$ from L. WCFI-RW prevents a large class of CFI attacks (e.g., ROP attacks): backward control edges (returns) are fully protected, and forward edges (calls and jumps) are significantly constrained. Furthermore, observe that by preventing jumps into the middle of instructions, we guarantee that the code of $U_{\mathcal{H}}$ that we statically verify is same that will execute at runtime. However, this form of CFI is weaker than standard CFI [2] because it allows a procedure in $U_{\mathcal{H}}$ to call any other procedure in $U_{\mathcal{H}}$. In other words, a program that satisfies WCFI-RW may exhibit control transfers that are not present in the source program, and this can bootstrap certain control-flow attacks. Nevertheless, for any such attack that is not blocked by WCFI-RW, we prove that they still cannot break IRC (soundness theorem in section 4.3); in the end, the attacker only observes encrypted values.

To formalize WCFI-RW, we construct a monitor automaton $\mathcal{M}$ (defined next) from $U_{\mathcal{H}}$ to check whether $U_{\mathcal{H}}$ satisfies WCFI-RW, similar in spirit to prior works on CFI [2, 19]. $\mathcal{M}$ is synchronously composed with $U_{\mathcal{H}}$, such that the statements executed by $U_{\mathcal{H}}$ form the input sequence of $\mathcal{M}$. We say that WCFI-RW is violated whenever $\mathcal{M}$ reaches a stuck state during the execution of $U_{\mathcal{H}}$. The formalization of WCFI-RW requires the following predicates over addresses in the region (illustrated in Figure 3). For any SIR address a, $\mathsf{AddrInHeap}(a)$ is true if a belongs to U's heap. $\mathsf{AddrInStack}(a)$ is true if a belongs to the SIR's stack (which is shared by both U and L). $\mathsf{AddrInU}(a)$ is true if a belongs to U's memory (stack, globals, or heap), and $\mathsf{AddrInL}(a)$ is true if a belongs to L's memory (globals or heap). $\mathsf{AddrInCode}(a)$ is true if a belongs to SIR's code (either U or L's code). Finally, $\mathsf{writable}(\mathtt{mem}, a)$ is true iff the bit corresponding to address a is set in the `write_bitmap`.

DEFINITION 3. **WCFI-RW Monitor Automaton**
$\mathcal{M} = (\mathcal{Q}, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ is a generalized pushdown automaton where $\mathcal{Q} = \{\sigma\}$ is its set of states, $\Sigma = \{$ `call` e, ret, `jmp` e, $\mathsf{v} := e$, $\mathsf{v} := \mathsf{load_n}(e_a)$, $\mathsf{store_n}(e_a, e_d)$, $\mathsf{havoc}_{\neg\mathsf{SIR}}\ \mathtt{mem}$ $\}$ is its set of inputs, $\Gamma = \{a \mid \mathsf{AddrInCode}(a)\}$ is its finite set of stack symbols, $q_0 = \sigma_{entry}$ is its initial state (σ_{entry} being the machine state following the jump from L into U's entry point), $Z_0 = a_{entry}$ is its initial stack, $F = Q$ is its set of accepting states, δ: $(\mathcal{Q} \times \Sigma \times \Gamma^*) \rightarrow \mathcal{P}(\mathcal{Q} \times \Gamma^*)$ is its transition function. Furthermore, let $\mathsf{next}(\mathsf{rip})$ be the address of the subsequent instruction in U after decoding the instruction starting at address rip, and σ' be the state resulting from executing a statement $s \in Stmt$ starting in σ, i.e., $(\sigma, \sigma') \in \mathcal{R}$ (as per the operational semantics in Figure 5). The transition function δ is as follows:

$$\delta(\sigma, \mathtt{call}\ e, \gamma) = \begin{cases} \{(\sigma', \mathsf{next}(\sigma(\mathsf{rip})) \cdot \gamma)\} & \textit{iff}\ \psi_{\mathtt{call}} \\ \emptyset & \textit{otherwise} \end{cases}$$

where $\psi_{\mathtt{call}} \doteq \sigma(e)$ *is the address of a procedure entry in U*

$$\delta(\sigma, \mathtt{call}\ \mathsf{x}, \gamma) = \{(\sigma', \gamma)\}$$

where $\mathsf{x} \in \{\mathsf{malloc}, \mathsf{free}, \mathsf{send}, \mathsf{recv}\}$

$$\delta(\sigma, \mathsf{ret}, a \cdot \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff}\ \sigma(\mathsf{mem})[\sigma(\mathsf{rsp})] = a \\ \emptyset & \textit{otherwise} \end{cases}$$

$$\delta(\sigma, \mathtt{jmp}\ e, \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff}\ \psi_{\mathtt{jmp}} \\ \emptyset & \textit{otherwise} \end{cases}$$

where $\psi_{\mathtt{jmp}} \doteq \psi_{\mathtt{call}} \vee \sigma(e)$ *is an instr. in current procedure*

$\delta(\sigma, \mathsf{flag} := e, \gamma) = \{(\sigma', \gamma)\}$ *where* $\mathsf{flag} \in \mathsf{flags}$

$$\delta(\sigma, \mathsf{rsp} := e, \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff}\ \psi_{\mathsf{rsp}} \\ \emptyset & \textit{otherwise} \end{cases}$$

where $\psi_{\mathsf{rsp}} \doteq \mathsf{AddrInU}(\sigma(e))$

$\delta(\sigma, \mathsf{reg} := e, \gamma) = \{(\sigma', \gamma)\}$ *where* $\mathsf{reg} \in \mathsf{regs} \setminus \{\mathsf{rsp}\}$

$$\delta(\sigma, \mathsf{reg} := \mathsf{load_n}(e_a), \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \psi_{\mathsf{load}} \\ \emptyset & \textit{otherwise} \end{cases}$$

where $\psi_{\mathsf{load}} \doteq \neg\mathsf{AddrInL}(\sigma(e_a)) \wedge \neg\mathsf{AddrInL}(\sigma(e_a)+n-1)$

$$\delta(\sigma, \mathsf{store_n}(e_a, e_d), \gamma) = \begin{cases} \{(\sigma', \gamma)\} & \textit{iff } \psi_{\mathsf{store}} \\ \emptyset & \textit{otherwise} \end{cases}$$

where $\psi_{\mathsf{store}} \doteq \mathsf{AddrInU}(\sigma(e_a)) \wedge \mathsf{AddrInU}(\sigma(e_a) + n - 1)$

$$\delta(\sigma, \mathsf{havoc}_{\neg\mathsf{SIR}}\ \mathtt{mem}, \gamma) = \{(\sigma', \gamma)\}$$

DEFINITION 4. **WCFI-RW**
WCFI-RW is violated in an execution trace $\pi = [\sigma_0, \ldots, \sigma_n]$ *when no transition exists in* $\mathcal{M}$ *for a statement in* $\mathtt{stmt}(\pi)$ *i.e.* $\mathcal{M}$ *gets stuck. Formally, WCFI-RW*(π) *iff (with starting state* $\sigma_0 = \sigma_{entry}$ *and initial stack* $\gamma_0 = a_{entry}$*):*

$$\exists \gamma_0, \ldots, \gamma_n \in \Gamma^* . \bigwedge_{k=0}^{n-1} (\sigma_{k+1}, \gamma_{k+1}) \in \delta(\sigma_k, \mathtt{stmt}(\sigma_k), \gamma_k)$$

$U_{\mathcal{H}}$ *satisfies WCFI-RW if all traces of* $U_{\mathcal{H}}$ *satisfy WCFI-RW.*

The role of the pushdown stack in Definition 4 is to match the calls and returns. $\mathcal{M}$ only modifies the pushdown stack on `call` and ret statements, and updates the state as per the operational semantics defined in Figure 5. We now describe each case in the definition of the transition function δ of $\mathcal{M}$. On a `call` to a procedure in U, $\mathcal{M}$ pushes the return address (i.e., the address of the subsequent instruction) onto the pushdown stack, for use by the ret statement. On a `call` to L's API, since L only contributes one step to the trace, and since correctness of L's APIs (section 4.2) guarantees that the `call` returns to the call site, $\mathcal{M}$ does not push the return address onto the pushdown stack. A ret produces a valid transition only when the topmost symbol on the pushdown stack matches the return address on the machine's stack; this transition pops the topmost element off the pushdown stack. A `jmp` produces a valid transition if it targets a legal instruction in the current procedure, or the beginning of a procedure in U. Assignment to rsp succeeds if the new value is an address in U's memory — this constraint is needed because `call` and ret accesses `mem` at address rsp, and WCFI-RW requires stores to be contained within U's memory. Other registers and flags can be assigned to arbitrary values. Finally, to satisfy WCFI-RW's constraints on reads and writes, a `load` proceeds iff the address is not within L's memory, and a `store` proceeds iff the address is within U's memory.

4.2 Correctness of L's API Implementation

While we strive for full functional correctness of L, the following contract (in conjunction with WCFI-RW of $U_{\mathcal{H}}$) is sufficient for proving IRC of the SIR.

(a) `malloc(size)` (where the return value `ptr` is the starting address of the allocated region) must not 1) modify non-SIR memory or stack frames belonging to U, 2) make any stack location writable, or 3) return a region outside U's heap. Formally, when $\mathtt{stmt}(\sigma) = \mathtt{call}$ malloc, we write $(\sigma, \sigma') \in \mathcal{R}$ iff ψ_{malloc} holds, where ψ_{malloc} is the conjunction of:

▷ $\forall a.\ (\neg\mathtt{SIR}(a) \vee (\mathsf{AddrInStack}(a) \wedge a \geq \sigma(\mathsf{rsp}))) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$

▷ $\forall a.\ \mathsf{AddrInStack}(a) \Rightarrow (\mathsf{writable}(\sigma(\mathsf{mem}), a) \Leftrightarrow \mathsf{writable}(\sigma'(\mathsf{mem}), a))$

▷ $\sigma'(\mathsf{ptr}) = 0\ \vee\ ((\sigma'(\mathsf{ptr}) \leq \sigma'(\mathsf{ptr}) + \sigma(\mathsf{size}))\ \wedge \mathsf{AddrInHeap}(\sigma'(\mathsf{ptr}))\ \wedge \mathsf{AddrInHeap}(\sigma'(\mathsf{ptr}) + \sigma(\mathsf{size})))$

First, by forbidding `malloc` from modifying U's stack above rsp, we prevent `malloc` from overwriting return addresses in U's stack frames. Second, by forbidding `malloc` from making a stack location writable, we prevent a return address from being corrupted later by code in U — `malloc` should only modify the `write_bitmap` to make the allocated region writable. Both restrictions are paramount for preventing WCFI-RW exploits. Finally, we require `malloc` to return a region from U's heap (or the null pointer), and ensure that a machine integer overflow is not exploited to violate IRC.

(b) `free(ptr)` must not 1) modify non-SIR memory or stack frames belonging to U, or 2) make any stack location writable. Formally, when $\mathtt{stmt}(\sigma) = \mathtt{call}$ free, we write $(\sigma, \sigma') \in \mathcal{R}$ iff ψ_{free} holds, where ψ_{free} is the conjunction of:

▷ $\forall a.\ (\neg\mathtt{SIR}(a) \vee (\mathsf{AddrInStack}(a) \wedge a \geq \sigma(\mathsf{rsp}))) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$

▷ $\forall a.\ \mathsf{AddrInStack}(a) \Rightarrow (\mathsf{writable}(\sigma(\mathsf{mem}), a) \Leftrightarrow \mathsf{writable}(\sigma'(\mathsf{mem}), a))$

These constraints are equivalent to the constraints on `malloc`, and are likewise paramount for preventing WCFI-RW exploits. Note that we do not require `malloc` to return a previously unallocated region, nor do we require `free` to mark the freed region as invalid; full functional correctness would require such properties. WCFI-RW does not assume any invariants on the heap values, and therefore vulnerabilities such as use-after-free do not compromise WCFI-RW.

(c) `send(ptr, size)` must not make any address writable or modify the stack frames belonging to U. Formally, when $\mathtt{stmt}(\sigma) = \mathtt{call}$ send, we write $(\sigma, \sigma') \in \mathcal{R}$ iff ψ_{send} holds, where ψ_{send} is:

▷ $\forall a.\ (\mathsf{AddrInBitmap}(a)\ \vee\ (\mathsf{AddrInStack}(a)\ \wedge a \geq \sigma(\mathsf{rsp}))) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$

`send` is used to encrypt and sign the message buffer before writing to non-SIR memory, and it is the only API call that is allowed to modify non-SIR memory. However, we forbid `send` from modifying a caller's stack frame or the bitmap. By preventing `send` from modifying U's stack above rsp, we prevent `send` from overwriting a return address in any of U's stack frames. Furthermore, `send` cannot modify the bitmap and make any location writable, thereby preventing a return address from being modified later by some code in U.

(d) `recv(ptr,size)` must 1) check that the destination buffer is a writable region in U's memory, and 2) not modify any memory location outside the input buffer. Formally, when $\texttt{stmt}(\sigma) = \texttt{call}$ recv, we write $(\sigma, \sigma') \in \mathcal{R}$ iff ψ_{recv} holds, where ψ_{recv} is the conjunction of:

- $\triangleright$ $\forall a.\ (\sigma(\mathsf{ptr}) \leq a < \sigma(\mathsf{ptr}) + \sigma(\mathsf{size})) \Rightarrow (\mathsf{writable}(\sigma(\mathsf{mem}),a) \ \wedge \mathsf{AddrInU}(a))$
- $\triangleright$ $\forall a. \neg(\sigma(\mathsf{ptr}) \leq a < \sigma(\mathsf{ptr}) + \sigma(\mathsf{size})) \Rightarrow \sigma(\mathsf{mem})[a] = \sigma'(\mathsf{mem})[a]$
- $\triangleright$ $\sigma(\mathsf{ptr}) \leq \sigma(\mathsf{ptr}) + \sigma(\mathsf{size})$

`recv` is used to copy an encrypted, signed message from non-SIR memory, decrypt it, verify its signature, and copy the cleartext message buffer to U's memory. The first two constraints ensure that the message is written to a writable memory region within U (which guarantees that a return address is not modified by `recv`), and that the cleartext message is not written out to non-SIR memory. The final constraint ensures that an integer overflow is not exploited to violate IRC.

In addition to the contracts above, we check the following contracts for each of `malloc`, `free`, `send`, and `recv`:

- page permissions, following the API call, are set to prevent read and write access to L's memory. Write access is disabled to prevent U from corrupting L's state, whereas read access is disabled to prevent reading L's secrets.
- stack pointer rsp is restored to its original value.
- the API call satisfies the application binary interface (ABI) calling convention (Windows x64 in our case)

For the purposes of this paper, we assume that the implementation of L satisfies the above contracts. Since L is written once, and used inside all SIRs, we could potentially verify the implementation of L once and for all manually.

4.3 Soundness

THEOREM 1. *If $U_{\mathcal{H}}$ satisfies WCFI-RW and the implementation of L's API satisfies the correctness properties given in section 4.2, then $U_{\mathcal{H}}$ satisfies IRC.*

A proof of Theorem 1 is given in supplement material [1].

5. Verifying WCFI-RW

In the remainder of this paper, we describe an automatic, static verifier for proving that a developer-provided $U_{\mathcal{H}}$ satisfies the WCFI-RW property. Verifying such a property at machine code level brings up scalability concerns. Our benchmarks consist of SIR programs that are upwards of 100 KBs in binary size, and therefore whole-program analyses would be challenging to scale. Intra-procedural analysis, on the other hand, can produce too many false alarms due to missing assumptions on the caller-produced inputs and state. For instance, the caller may pass to its callee a pointer to some heap allocated structure, which the callee is expected to modify. Without any preconditions on the pointer, a modular verifier might claim that the callee writes to non-SIR memory, or corrupts a return address, etc.

Instead of verifying WCFI-RW of arbitrary machine code, our solution is to generate machine code using a compiler that emits runtime checks to enforce WCFI-RW, and automatically verify that the compiler has not missed any check. Our compiler emits runtime checks that enforce that unconstrained pointers (e.g., from inputs) are not used to corrupt critical regions (e.g., return addresses on the stack), write to non-SIR memory, or jump to arbitrary code, etc. As our experiments show, the presence of these checks eliminates the unconstrained verification environments described above. Consequently, most verification conditions (VCs) that we generate can be discharged easily. Even in cases where the compiler eliminates checks for efficiency, the compiler does not perform any inter-procedural optimization, and we demonstrate that a modular verifier can prove that eliminating the check is safe.

5.1 Runtime Checks

We use the compiler to 1) prepend checks on `store` instructions to protect return addresses in the stack, 2) prepend checks on `store` instructions to prevent writes to non-SIR memory, and 3) prepend checks on indirect `call` and `jmp` instructions to enforce valid jump targets. We also use the processor's page-level access checks for efficiently preventing reads and writes to L's memory by code in $U_{\mathcal{H}}$.

Runtime Checks on Stores: To enforce that writes through pointers do not corrupt return addresses on the stack, the compiler maintains a bitmap (see `write_bitmap` in Figure 3) to record which areas in U's memory are writable, while maintaining the invariant that a return address is never marked writable. The `write_bitmap` maps every 8-byte slot of U's memory to one bit, which is set to one when those 8 bytes are writable. The bitmap is updated at runtime, typically to mark address-taken local variables and `malloc`-returned regions as writable, and to reset the bitmap at procedure exits or calls to `free`. For instance, if a caller expects a callee to populate the results in a stack-allocated local variable, the caller must mark the addresses of that local variable as writable before invoking the callee. A `store` instruction is prepended with an instruction sequence that reads the `write_bitmap` and terminates the SIR program if the corresponding bit is zero (see instructions from L1 to L2 in Figure 3). This check on `store` can enable stronger properties than backward edge CFI in that it also prevents many forms of memory corruptions. While this gives us stronger security guarantees at runtime, we only require a weak form of CFI for proving WCFI-RW.

In addition, the compiler prepends `store` with range checks that prevent writes outside the SIR region (see instructions from L0 to L1 in Figure 3).

Finally, we use the processor's paging instructions to revoke write permissions on L's memory while $U_{\mathcal{H}}$ executes, and to reinstate write permissions following an API call to L — we also use page permissions to make code pages and the `call_bitmap` non-writable at all times. The SIR provider guarantees that the processor respects the page permissions of SIR memory. In the case of Intel SGX 2.0, the processor provides the emodpr instruction to change page permissions of enclave (SIR) memory. The processor hardware performs the page-level access checks without any noticeable performance overhead. Note that we cannot use the page permissions to prevent writes to non-SIR memory because the adversary $\mathcal{H}$ controls the page tables for all non-SIR memory.

Runtime Checks on Loads: WCFI-RW mandates that $U_{\mathcal{H}}$ does not `load` from L's memory. This ensures that $U_{\mathcal{H}}$ never reads secrets such as the secure channel's cryptographic keys, which is necessary because strong encryption properties no longer hold if the key itself is used as plain text. To that end, L disables read access to its memory by setting the appropriate bits in the page tables. On each API call, L first sets the page permissions to allow access to its own memory, and resets it before returning back to $U_{\mathcal{H}}$.

On a side note, although we would like $U_{\mathcal{H}}$ to only read SIR memory (and use recv to fetch inputs from non-SIR memory), we avoid introducing range checks for two reasons: 1) WCFI-RW does not require this guarantee, and 2) loads are frequent, and the range checks incur significant additional performance penalty.

Runtime Checks on Indirect Control Transfers: The compiler maintains a separate bitmap (see `call_bitmap` in Figure 3) that records the entry points of procedures in U and APIs of L. The `call_bitmap` maps every 16-byte slot of U's memory to one bit, and the compiler accordingly places each procedure's entry point in code at a 16-byte aligned address. The compiler prepends indirect calls with an instruction sequence that reads the bit within `call_bitmap` corresponding to the target address, and terminates the SIR program if that bit is zero. Indirect jumps to within the procedure are also checked to prevent jumps to the middle of x64 instructions, which can lead to control-flow hijacks.

The reader may question our choice of runtime checks, as one could simply instrument instructions implementing the validity checks on the corresponding transitions in the WCFI-RW monitor $\mathcal{M}$ (from Definition 3). However, this would require us to build a provably correct implementation of a *shadow stack* within SIR memory, and use the shadow stack during `call` and `ret` instructions to check that the processor uses the same return address as the one pushed by the matching `call` instruction. However, it is non-trivial to protect the shadow stack from code running at the same privilege level — doing so might require the very techniques that we use in our runtime checks.

5.2 Static Verifier for WCFI-RW

We present a modular and fully automatic program verifier, called /CONFIDENTIAL, for checking that the compiler-generated machine code satisfies the WCFI-RW property. Since runtime checks incur a performance penalty, the compiler omits those checks that it considers to be redundant. For instance, the compiler eliminates checks on writes to local scalar variables (and therefore does not need to make them writable in the `write_bitmap`) and to variables whose addresses are statically known. The compiler also tries to hoist checks out of loops whenever possible. These optimizations do not compromise WCFI-RW, and /CONFIDENTIAL proves them safe so as to avoid trusting the compiler. Since we verify WCFI-RW at the machine code level, we do not need to trust the implementation of these optimizations.

/CONFIDENTIAL is based on a set of proof obligations for verifying that the output machine code (modeled as $U_{\mathcal{H}}$) satisfies WCFI-RW. It modularly verifies each procedure in isolation and is still able to prove WCFI-RW for the entire program — this soundness guarantee is formalized as a theorem that we present later in this section. It is important to note that modular verification is possible because the compiler does not perform any global analysis to optimize away the runtime checks. /CONFIDENTIAL generates proof obligations for each `store`, `load`, `call`, `ret`, `jmp`, and `rsp` update in the procedure. While generating proof obligations, /CONFIDENTIAL does not distinguish statements based on whether they originated from U's source code or the runtime checks, since the compiler is untrusted. These proof obligations are implemented by instrumenting $U_{\mathcal{H}}$ with static assertions, which are discharged automatically using an SMT solver by the process of VC generation. We present the instrumentation rules in Table 1, and describe them below.

The instrumentation rules use the following functions, which are defined for a given U:

- $\mathsf{policy}(e)$ is true iff address e is the starting address of a procedure in U or an API entrypoint of L. This predicate is consistent with the state of `call_bitmap`, which remains constant throughout the SIR's execution.
- $\mathsf{writable}(\mathtt{mem}, e)$ is true iff the bit corresponding to address e is set to one in the `write_bitmap` region of `mem`.
- $\mathsf{b}(\mathtt{mem}, e_a, e_d)$ is a partial function (only defined for values of e_a for which $\mathsf{AddrInBitmap}(e_a)$ holds) that returns the largest address that is marked writable as a result of executing $\mathtt{store}(e_a, e_d)$
- $\mathsf{start}(p)$ returns the starting address of procedure p
- $\mathsf{end}(p)$ returns the ending address of procedure p
- $\mathsf{legal}(e)$ is true for any e that is the starting address of an instruction in U — we need this predicate because x64 instructions have variable lengths.

Stmt s	Instrumented Stmt $I(s)$
`call` e	`assert` $\mathsf{policy}(e) \ \wedge \ (\forall i.\ (\mathsf{AddrInStack}(i) \wedge i < \mathsf{rsp}) \Rightarrow \neg \mathsf{writable}(\mathtt{mem}, i)) \ \wedge \ (\mathsf{rsp} \leq \mathsf{old}(\mathsf{rsp}) - 32)$; `call` e
$\mathtt{store}_{\mathsf{n}}(e_a, e_d)$	`assert` $(\bigvee_i^{\{e_a, e_a+n-1\}} (\mathsf{AddrInStack}(i) \wedge i \geq \mathsf{old}(\mathsf{rsp}) \wedge \neg(\mathsf{old}(\mathsf{rsp}) + 8 \leq i < \mathsf{old}(\mathsf{rsp}) + 40))) \Rightarrow \mathsf{writable}(\mathtt{mem}, e_a)$; `assert` $(\bigwedge_i^{\{e_a, \ldots, e_a+n-1\}} (\mathsf{AddrInBitmap}(i) \Rightarrow (\mathsf{b}(\mathtt{mem}, i, e_d[8*(i+1-e_a) : 8*(i-e_a)]) < \mathsf{old}(\mathsf{rsp}) - 8)))$; `assert` $\mathtt{SIR}(e_a)$; $\mathtt{store}_{\mathsf{n}}(e_a, e_d)$
$\mathsf{rsp} := e$	`assert` $(e[3:0] = 000 \wedge e \leq \mathsf{old}(\mathsf{rsp}))$; $\mathsf{rsp} := e$
ret	`assert` $(\mathsf{rsp} = \mathsf{old}(\mathsf{rsp})) \wedge (\forall i.\ (\mathsf{AddrInStack(i)} \wedge i < \mathsf{old}(\mathsf{rsp})) \Rightarrow \neg \mathsf{writable}(\mathtt{mem}, i))$; ret
`jmp` e	`assert` $(\mathsf{start}(p) \leq e < \mathsf{end}(p)) \rightarrow \mathsf{legal}(e)$; `assert` $\neg(\mathsf{start}(p) \leq e < \mathsf{end}(p)) \rightarrow (\mathsf{rsp} = \mathsf{old}(\mathsf{rsp}) \ \wedge \ \mathsf{policy}(e) \ \wedge \ (\forall i.\ (\mathsf{AddrInStack}(i) \wedge i < \mathsf{rsp}) \Rightarrow \neg \mathsf{writable}(\mathtt{mem}, i))$; `jmp` e

Table 1. Instrumentation rules for modularly verifying WCFI-RW

- $\mathsf{old}(\mathsf{rsp})$ is the value of rsp at procedure entry, and is modeled as a symbolic variable because the procedure may be called at an arbitrary depth in the call stack.

Functions policy, start, and end are defined by parsing the executable (DLL format) produced by the compiler. legal is defined by disassembling the executable code, which is a precursor to the formal modeling step that produces $U_{\mathcal{H}}$. Since the memory layout and code pages remain constant throughout execution, these functions are defined once for a given U and are independent of the current state. Functions writable and b are evaluated on the contents of `write_bitmap` within `mem`, and their definition involves a `load` from `mem` and several bitvector operations. We also recall predicates AddrInStack, AddrInBitmap, AddrInL, and `SIR` from section 4, which are used to define various regions in the SIR's memory (see Figure 3).

Static Assertions on Calls: On each statement of the type `call` e, we assert that 1) the target address e is either a procedure in U or an API entrypoint in L, 2) all addresses in the callee's stack frame are initially unwritable, and 3) the caller follows the Windows x64 calling convention by allocating 32 bytes of scratch space for use by the callee.

Static Assertions on Stores: $U_{\mathcal{H}}$ may invoke $\mathtt{store}_{\mathsf{n}}(e_a, e_d)$ on an arbitrary virtual address e_a with arbitrary data e_d. Hence, we must argue that the proof obligations prevent all `store` instructions that violate WCFI-RW. We case split this safety argument for each memory region in the virtual address space (Figure 3). The `call_bitmap`, the code pages, and L's memory are marked non-writable in the page tables — `store` to these areas results in an exception, followed by termination. Within U's memory, /CONFIDENTIAL treats all writes to the heap and globals as safe because WCFI-RW does not require any invariants on their state — while the heap and global area may store code and data pointers, /CONFIDENTIAL instruments the necessary assertions on indirect control transfers and dereferences, respectively. We are left with potential stores to U's stack, `write_bitmap`, and non-SIR memory, and their proof obligations are:

- $\mathsf{AddrInStack}(e_a)$: if e_a is an address in a caller's stack frame but not in the 32-byte scratch space (which is addressed from $\mathsf{old}(\mathsf{rsp}) + 8$ to $\mathsf{old}(\mathsf{rsp}) + 40$), then e_a must be marked by the `write_bitmap` as writable. On the other hand, $U_{\mathcal{H}}$ is allowed to write arbitrary values to the current stack frame or the 32-byte scratch space.
- $\mathsf{AddrInBitmap}(e_a)$: only addresses in the current stack frame can be made writable. It suffices to check that the largest address whose write permission is being toggled is below $\mathsf{old}(\mathsf{rsp})$. Note that unaligned stores may change two words at once. Since the instrumentation code only checks the write permissions of the first word (for performance reasons), we further restrict the largest address to be below $\mathsf{old}(\mathsf{rsp}) - 8$ to account for unaligned stores of up to 8 bytes.
- $\mathsf{SIR}(e_a)$: WCFI-RW mandates that $U_{\mathcal{H}}$ does not `store` to non-SIR memory, for which /CONFIDENTIAL generates a proof obligation: `assert` $\mathtt{SIR}(e_a)$.

Static Assertions on Assignments to* rsp*: For each statement of the type $\mathsf{rsp} := e$, we check that the new stack pointer e 1) is 8-byte aligned, 2) does not point to a caller's stack frame (i.e., must not be greater than the old rsp). The constraint that the rsp never points to a caller's stack frame is necessary for modular verification. We use a guard page (i.e., a page without read or write page permissions) to protect against stack overflows — in the case where the procedure needs stack space larger than a page, we check that the compiler introduces a dummy load that is guaranteed to hit the guard page and cause an exception, thus preventing the procedure from writing past the guard page.

Static Assertions on Returns: For each ret statement, we check that 1) rsp has been restored to its original value, and 2) the procedure has reset the `write_bitmap` so that all addresses in the current stack frame are unwritable.

Static Assertions on Jumps: A `jmp` is safe if it either targets a legal address within the current procedure p (i.e., not in the middle of an instruction), or the start of a procedure (often used for performing tail calls). In the case of `jmp` to a procedure, we check the same properties as a `call` instruction, except that rsp is restored to its original value.

Syntactic Check for SIR instructions: Code in $U_{\mathcal{H}}$ runs at the same privilege level as L, and hence may invoke instructions to override page permissions (such as emodpr in SGX 2.0). We guard against this vulnerability by simply checking for the presence of special SIR instructions (all SGX instructions) in $U_{\mathcal{H}}$, which is captured by a regular expression on the disassembled machine code. Though not strictly required for WCFI-RW, /CONFIDENTIAL forbids all SGX instructions because of certain instructions (such as egetkey for accessing the sealing key) which return cryptographic secrets to the user code.

5.3 Optimization to the Proof Obligations

If we can estimate the stack size needed for a procedure, then we can optimize the proof obligations for `store`, ret, and `call` statements (see Table 2). The modifications are:

- `store`: further assert that updating the `write_bitmap` does not mark any address below the estimated stack space to be writable.
- `call`: further assert that the current rsp is not within the estimated stack space (which would otherwise falsify the callee's precondition that the stack space is non-writable). The modified assertion on `store` also allows us to omit the proof obligation that all addresses below the current rsp are non-writable (prior to the `call`).
- ret: now asserts non-writability of only the addresses in the estimated stack space, instead of all addresses below the old(rsp). Since the range of addresses is bounded, we instantiate the $\forall$ quantifier, and help the SMT solver to eliminate hundreds of timeouts in our experiments.

Although the optimization is sound for any positive value of estimate, we are able to compute a precise estimate for all of our benchmarks. The estimate is computed by aggregating all the stack subtractions and checking that there is no assignment to rsp within a loop. If rsp is assigned within a loop body, then the optimization is disabled. Furthermore, this optimization may lead to false positives in rare cases of safe programs, in which case we fall back to the unoptimized implementation. For instance, this may happen if a procedure decrements rsp after making a procedure `call`, but we have not encountered such cases in our evaluation.

5.4 Soundness

The following theorem states that our proof obligations imply WCFI-RW.

THEOREM 2. *(Soundness of I) Let p be a procedure, and $I(p)$ be procedure p instrumented with the assertions given in Table 1 (and with optimizations in Table 2). If for each p in $U_{\mathcal{H}}$, $I(p)$ is safe (i.e., no trace of $I(p)$ violates such an assertion), then $U_{\mathcal{H}}$ satisfies WCFI-RW.*

A proof of Theorem 2 is given in supplement material [1].

6. Implementation

We develop a toolchain for building IRC-preserving SIRs. The developer first compiles U's code (written in C/C++) using the VC3 compiler [38], to insert checks on indirect control-flow transfers, and checks on stores to prevent tampering of return addresses and to prevent stores from exceeding the SIR boundary — the instrumentation also protects against several classes of memory corruption errors, but we do not leverage these guarantees for proving IRC.

/CONFIDENTIAL takes as input a DLL with U's code, and an implementation of L that provides the required guarantees. First, /CONFIDENTIAL parses the DLL to extract all the procedures. Next, for each procedure, /CONFIDENTIAL invokes the Binary Analysis Platform (BAP [10]) to lift the x64 instructions to statements in our language (Figure 4), which are then converted to BoogiePL [15]. Indirect `jmp` and `call` instructions require some preprocessing before modeling them in BoogiePL, and we discuss this detail later in this section.

Next, for proving WCFI-RW, /CONFIDENTIAL instruments each procedure in BoogiePL with `assert` statements as given in Table 1 and Table 2. The Boogie verifier [4] generates VCs and automatically discharges them using the Z3 SMT solver [14]. If all `assert` statements in all procedures are valid, then by Theorem 1 and Theorem 2, $U_{\mathcal{H}}$ satisfies IRC. /CONFIDENTIAL checks the validity of each `assert` in parallel, which in combination with the modular analysis, allows /CONFIDENTIAL to scale to realistic SIR programs.

Modeling Procedure Calls Since the analysis is modular, /CONFIDENTIAL replaces each procedure call by a havoc to the machine state in lieu of specific procedure summaries. The havoc is performed by assigning fresh, symbolic values to volatile registers and CPU flags, and assigning a fresh, symbolic memory (called new_mem below) which is subject to certain constraints as shown below. We encode the constrained havoc to machine state using the following statements in order, which are instrumented after the `call` statement in $U_{\mathcal{H}}$.

▷ `assume` $\forall i.\ (\mathsf{AddrInStack}(i) \land i < \mathsf{rsp} \land \neg \mathsf{writable}(\mathtt{mem}, i)) \Rightarrow \mathsf{load}_8(\mathtt{mem}, i) = \mathsf{load}_8(\mathtt{new\_mem}, i)$
A callee procedure may have an unbounded number of store instructions that can modify any memory location marked writable in the `write_bitmap` — the havoc must preserve the non-writable locations in the current stack frame because our instrumentation guarantees that a callee cannot change their writability (as opposed to the heap which may be made writable by calling `malloc`, and then modified).

▷ `assume` $\forall i.\ \mathsf{AddrInStack}(b(\mathtt{mem}, i, 11111111)) \Rightarrow \mathsf{load}_1(\mathtt{mem}, i) = \mathsf{load}_1(\mathtt{new\_mem}, i)$
Our instrumentation guarantees that a portion of the `write_bitmap` (specifically the part that controls stack addresses) is restored on ret, which validates this assumption.

Stmt s	Instrumented Stmt $I(s)$
$\texttt{store}_{\mathsf{n}}$ (e_a, e_d)	$\texttt{assert}\ (\bigvee_i^{\{e_a, e_a+n-1\}} (\mathsf{AddrInStack}(i) \wedge i \geq \mathsf{old(rsp)} \wedge \neg(\mathsf{old(rsp)} + 8 \leq i < \mathsf{old(rsp)} + 40))) \Rightarrow \mathsf{writable}(\texttt{mem}, e_a)$; $\texttt{assert}\ \bigwedge_i^{\{e_a, \ldots, e_a+n-1\}} (\mathsf{AddrInBitmap}(i) \Rightarrow (\mathsf{old(rsp)} - \mathsf{estimate} \leq \mathsf{b}(\texttt{mem}, i, e_d[8*(i+1-e_a) : 8*(i-e_a)]) < \mathsf{old(rsp)} - 8))$; $\texttt{assert}\ \texttt{SIR}(e_a)$; $\texttt{store}_{\mathsf{n}}(e_a, e_d)$
$\texttt{call}\ e$	$\texttt{assert}\ \mathsf{policy}(e)\ \wedge\ (\mathsf{rsp} \leq \mathsf{old(rsp)} - 32)\ \wedge\ (\mathsf{rsp} \leq \mathsf{old(rsp)} - \mathsf{estimate})$; $\texttt{call}\ e$
ret	$\texttt{assert}\ (\mathsf{rsp} = \mathsf{old(rsp)})\ \wedge\ (\forall i.\ (i < \mathsf{old(rsp)} \wedge i \geq \mathsf{old(rsp)} - \mathsf{estimate}) \Rightarrow \neg\mathsf{writable}(\texttt{mem}, i))$; ret

Table 2. Optimized instrumentation rules for `store`, `call`, and ret statements

- ▷ mem := new_mem
  This step assigns a new memory that is related to the old memory by the above `assume` statements.
- ▷ havoc rax, rcx, rdx, r8, r9, r10, r11;
  This step havocs all volatile registers (as defined by the Windows x64 calling convention) with fresh, symbolic values.
- ▷ havoc ZF, AF, OF, SF, CF, PF;
  The callee may cause arbitrary updates to the CPU flags, which is modeled by this havoc.

The constrained havoc above models an arbitrary U procedure that has an unbounded number of statements in any order — we prove this lemma within the proof of the soundness theorem 2. The constrained havoc (in the statements above) is followed by a jump to the next instruction, as computed during disassembly. This is sound because WCFI-RW guarantees that the callee uses the return address placed by the caller. There is a caveat that a call to L's API is replaced by its contract (defined in section 4.2) in lieu of the constrained havoc defined above.

We also assume the following preconditions at the beginning of each procedure:

- ▷ $\forall i.\ (\mathsf{AddrInStack}(i) \wedge i \geq \mathsf{old(rsp)}) \Rightarrow \neg\mathsf{writable}(\texttt{mem}, i)$
  This assumption treats all addresses in the local stack frame as non-writable upon procedure entry. It is upon the procedure to explicitly update the `write_bitmap` to make parts of its stack frame writable. This precondition is sound since we enforce it at all call sites.
- ▷ $\mathsf{AddrInStack}(\mathsf{old(rsp)}) \wedge \mathsf{old(rsp)}[3:0] = 000$
  We assume the initial stack pointer must be within the stack region and that it is 8-byte aligned. This precondition is sound because we enforce this property on every assignment to rsp.

Modeling Indirect Control Transfers. In order to use VC Generation and SMT solving, we need to statically approximate the set of jump targets for each register-based indirect jump. While we can use standard off-the-shelf value analysis, we observed that the compiler idiomatically places a jump table in memory, which is indexed dynamically to compute the target. Correspondingly, our verifier determines the base address of the jump table and reads its contents to compute the set of potential jump targets; this step is not trusted. An indirect jump is then modeled as a "switch" over direct jump statements to the potential targets, with the default case being `assert` false. The presence of `assert` false allows the approximation step to be untrusted. Indirect calls are handled using a similar method as direct calls; we introduce a constrained havoc on writable memory, volatile registers, and all CPU flags.

Modeling Havocs from $\mathcal{H}$. While our adversary model requires inserting $\mathsf{havoc}_{\neg\mathsf{SIR}}$ `mem` before each statement in U, it is efficient and sound to do so only before `load` statements [40]. We havoc the result of a `load` statement if the address is a location in non-SIR memory; reg := $\mathsf{load}_{\mathsf{n}}(e)$ is transformed to if $(\texttt{SIR}(e))$ {reg := $\mathsf{load}_{\mathsf{n}}(e)$} else {havoc reg}.

Verifying Procedures with Loops. /CONFIDENTIAL uses a candidate loop invariant that a portion of the `write_bitmap` (specifically the part that controls stack addresses) is preserved across loop iterations — we expect this invariant to hold because 1) the VC3 compiler tends to set the `write_bitmap` only in the procedure's prologue, which occurs before loop bodies, and 2) our proof obligations guarantee that callees preserve this portion of the `write_bitmap`. Empirically, we find that this loop invariant is sufficient for proving our assertions within loop bodies.

7. Evaluation

We evaluate /CONFIDENTIAL on several SIR programs that process sensitive data, and we summarize the results in Table 3 and Figure 6. We choose the three largest MapReduce examples from VC3 [38]: Revenue, IoVolumes, and UserUsage. UserUsage and IoVolumes processes sensitive resource usage data from a cloud platform. IoVolumes processes storage I/O statistics, and UserUsage aggregates the total execution time per user. Revenue reads a log file from a website and calculates the total ad revenue per IP address. Each of these benchmarks implement the mappers and reducers within the SIRs, and place large parts of the untrusted Hadoop stack outside the SIR boundary. Performance evaluation of these applications is described in [38], which reports that the average cost of the run-time checks is 15%. We also experiment with three SPEC CPU2006 benchmarks: bzip2, astar, and lbm.

As Table 3 and Figure 6 show, /CONFIDENTIAL is able to prove almost all assertions needed to check WCFI-RW in less than 20 seconds each, which demonstrates the potential in scaling our approach. We performed full verification of the lbm benchmark (i.e., no timeouts or false positives), which took roughly 3 hours of wall clock time. We also discovered few procedures across many benchmarks that have instructions that BAP [10] could not process, and we plan to experiment with alternative tools in future.

All benchmarks were compiled with the optimization level at -O2. All experiments were performed on a machine with 160GB RAM and 12 Intel Xeon E5-2440 cores running at 2.40GHz. As mentioned previously, /CONFIDENTIAL parallelizes the verification by spawning several instances of the Z3 SMT solver, where each instance is responsible for proving one of the instrumented static assertions.

Benchmark	Code Size	Verified Asserts	Timed out Asserts	False Positives
UserUsage	14 KB	2125	2	4
IoVolumes	17 KB	2391	2	0
Revenue	18 KB	1534	3	0
lbm	38 KB	1192	0	0
astar	115 KB	6468	2	0
bzip2	155 KB	10287	36	0

Table 3. Summary of results.

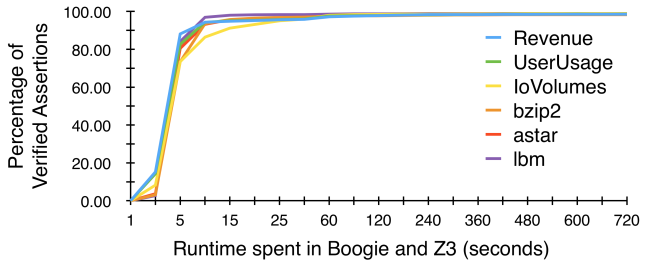

Figure 6. Summary of performance results

False Positives. We found four assertions that produced spurious counterexample traces, all within a single procedure of UserUsage. The violating procedure is a C++ constructor method, which writes the `vtable` pointer in the newly allocated object. Since the memory allocator terminates the SIR if it fails to allocate memory, the compiler optimizes away the range checks on the pointer returned by the memory allocator — this is the only observed instance of the compiler performing global optimization. Since /CONFIDENTIAL does not do any global analysis, it flags this method as unsafe. We could fix the issue by disabling this optimization.

Timeouts. Prior to implementing the optimizations in section 5.3, we had experienced several hundred timeouts. After the optimizations, only roughly 0.2% of all assertions (across all benchmarks) do not verify within the 30 minute timeout, as shown in Table 3. The main source of complexity in the generated VC comes from the combination of quantifiers and multiple theories such as arrays and bitvectors that are typically hard for SMT solvers. Another reason is the presence of a few large procedures in the SPEC benchmarks — largest procedure has above 700 x64 instructions and 5200 BoogiePL statements — which generated large SMT formulas. These large procedures (considering those above 420 x64 instructions and 3500 BoogiePL statements) accounted for 31 (69%) timeouts. We found that all of these assertions are associated with `store` instructions, and they enforce that the `store` targets a writable region in memory — one feature of these large procedures is heavy use of the stack space in memory, potentially causing the SMT solver to reason heavily about aliasing in order to prove that the target address is marked non-writable. Ten (22%) of the timeouts are on assertions associated with ret instructions, where the solver struggled to prove that all locations in the current stack frame are made non-writable (even with the optimization in section 5.3) — unless the procedure explicitly resets the `write_bitmap` prior to the ret, the SMT solver must prove that none of the stores in the procedure are unsafe. The remainder of the timeouts, roughly 9%, were neither on the return instructions, nor in large procedures — we find that they are associated with `store` instructions, where the solver is able to prove that the `store` targets the `write_bitmap` but not whether the written value is safe.

We manually investigated the assertions that time out (by experimenting at the level of the BoogiePL program) and were able to prove some of them using additional invariants and abstractions, without requiring any specific knowledge of the benchmark or its source code. Although we machine check these proofs (using Boogie and Z3), we continue to count them as timeouts in Table 3, since our goal is to have a fully automatic verifier of WCFI-RW. For roughly half of the timeouts observed on ret instructions, we hypothesized intermediate lemmas (a few instructions prior to the ret) and simplified the VC by introducing a `havoc` to `mem` followed by an `assume` that is strong enough needed to prove the final assertion — we also prove that the intermediate lemma holds prior to the `havoc`, making this transformation sound. Specifically, for a stack address a, we either 1) havoc the `write_bitmap` if the procedure contains instructions to reset the `write_bitmap` corresponding to address a, and these instructions are sufficient to prove the final assertion $\neg$writable(`mem`, a), or 2) we introduce `assert` $\neg$writable(`mem`, a) at earlier points in the program, if the procedure does not make a writable. This approach eliminates 6 of the 10 timeouts on ret instructions.

We also experimented with the 31 timeouts on `store` instructions within the large procedures. With the exception of 3 of these 31 timeouts, we were not able to get Z3 to prove the assertions, even after simplifying the VC with intermediate lemmas and havoc statements. These 3 assertions were at relatively shallow depths in the control flow graph of the procedure, where there are fewer loads and stores leading to the assertion. Finally, we tried the CVC4 [5] solver, but we did not succeed in eliminating any more timeouts.

Having performed this investigation, we are hopeful that with improving SMT solvers and better syntactic heuristics for simplifying the VCs, we will eliminate all timeouts.

8. Related Work

Confinement of programs to prevent information release has been studied for many years [22]. We are interested in achieving confinement in user programs, even with buggy or malicious privileged code. We use trusted processors [3, 24, 28, 34] to create isolated memory regions where we keep confidential application code and data, but our techniques are also applicable if isolation is provided by the hypervisor [11, 20, 45], or runtime checks [13]. Independently of the isolation provider, we need to reason about the application code to provide formal guarantees of confinement, which is the goal of our work.

Our method to check for WCFI-RW draws inspiration from prior work to perform instrumentation in order to satisfy control-flow integrity (CFI [2, 33]) and software fault isolation (SFI [27, 39, 44]). Like SFI, we introduce run-time checks to constrain memory references. Our run-time checks are similar to the ones used in VC3 [38], but importantly we use the paging hardware to check reads, which is more efficient than relying on compiler instrumentation. Unlike VC3, we verify that our checks guarantee IRC. Native Client [47] also enforces a form of SFI, but its run-time checks for 64-bit Intel processors would require us to create SIRs with a minimum size of 100GB [39], which is not practical for our target environment (x86-64 CPUs with SGX extensions). This is because the SIR's address space must be statically configured and physically mapped by the CPU upon the SIR's creation, whereas the 64-bit Native Client scheme was implemented in a setting where the virtual address space can be large. Our run-time checks also enforce stronger security properties; for example, Native Client does not guarantee that calls return to the instruction immediately after the call. Native Client ultimately enforces a different policy: it aims to sandbox browser extensions and trusts the host OS, while we aim to isolate an application from a hostile host. This requires us to model a powerful, privileged adversary ($\mathcal{H}$) while reasoning about the application's execution.

There have also been efforts to perform SFI with formally verified correctness guarantees. RockSalt [30] uses Coq to reason about an x86 processor model and guarantee SFI; it works for 32-bit x86 code, while our system works for the x86-64 ISA. ARMor [49] uses HOL to reason about ARM processor model and guarantee SFI. Native Client, XFI [18] and [48] include verifiers that work on machine code. Our verification scheme is different from these works since it uses different runtime checks (which provide stronger guarantees) and it supports aggressive compiler optimizations that remove redundant checks. We require more complex reasoning and thus use an SMT solver to build our verifier.

Unlike all of the above works, our ultimate goal is preserving confidentiality of a trusted application running in an untrusted and hostile host. Our specific definition of WCFI-RW, together with contracts we assume on the library methods guarantees IRC, which is the novel aspect of our work. We also prove that all the pieces (the compiler checks, the static verification, and the contracts on library methods) all combine together and guarantee IRC. Moat [40] has the same goal as our work, and the main difference is that Moat works for any code, and our work requires the application to perform all communications through a narrowly constrained interface. On the flip-side, Moat performs global analysis, tracks secrets in a fine-grained manner, and is not scalable beyond programs containing few hundred x86 instructions. In contrast, our approach is modular, avoids fine-grained tracking of secrets, and hence scales to larger programs. As mentioned before, our notion of confidentiality does not prevent information leaks via side channels such as memory access patterns. This channel has been addressed in GhostRider [25], which presents a co-designed compiler and hardware (containing Oblivious RAM) for guaranteeing memory trace oblivious computation.

Translation validation (e.g., [32, 35, 41, 42]) is a set of techniques that attempt to prove that compiler optimizations did not change the semantics of the program given as input (after the optimizer run). Our work is similar in spirit to translation validation since we use an off-the-shelf, untrusted compiler and validate whether the code it produced satisfies the security properties we are interested in.

9. Conclusion

We presented a methodology for designing Secure Isolated Regions, which enables certification of applications that need their code and data to remain confidential. Our methodology comprises enforcing the user code to communicate with the external world through a narrow interface, compiling the user code with a compiler that inserts run-time checks that aid verification, and linking it with a verified runtime that implements secure communication channels. We formalized the constraints on user code as Information Release Confinement (IRC), and presented a modular automatic verifier to check IRC. We believe that IRC, together with additional requirements on the implementation of the runtime, can guarantee a strong notion of confidentiality.

Acknowledgments

The authors from the University of California at Berkeley are supported in part by the NSF STARSS 1528108 and SRC 2460.001 grants and by Microsoft Research. We gratefully acknowledge our shepherd Michael Hicks and the anonymous reviewers for their insightful feedback.

References

[1] https://slashconfidential.github.io.

[2] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[3] ARM Security Technology - Building a Secure System using TrustZone Technology. ARM Technical White Paper.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, 2005.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[6] S. Bauer, P. Cuoq, and J. Regehr. Deniable backdoors using compiler bugs. *International Journal of PoC||GTFO*, 0x08:7–9, June 2015.

[7] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Technical Report MTR-2997, MITRE Corp., 1975.

[8] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, 1977.

[9] J. Black, J. Rogaway, and T. Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *SAC*, 2002.

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *CAV*, 2011.

[11] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS*, 2008.

[12] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, Sept. 2010.

[13] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *ASPLOS*, 2014.

[14] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[15] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.

[16] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976.

[17] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[18] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. Necula. XFI: software guards for system address spaces. In *OSDI*, 2006.

[19] A. Fontaine, P. Chifflier, and T. Coudray. Picon : Control flow integrity on llvm ir. In *SSTIC*, 2015.

[20] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. InkTag: Secure applications on an untrusted operating system. In *ASPLOS*, 2013.

[21] Intel Software Guard Extensions Programming Reference. Available at https://software.intel.com/sites/default/files/329298-001.pdf, 2014.

[22] B. Lampson. A note on the confinment problem. *Communications of the ACM*, 16(10), 1976.

[23] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[24] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS*, 2000.

[25] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi. Ghostrider: A hardware-software system for memory trace oblivious computation. In *ASPLOS*, 2015.

[26] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015.

[27] S. McCamant and G. Morrisett. Evaluating SFI for a CISC architecture. In *Usenix Security*, 2008.

[28] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[29] R. Morisset, P. Pawan, and F. Z. Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, 2013.

[30] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[31] A. C. Myers and B. Liskov. A decentralized model for information flow control. In *SOSP*, 1997.

[32] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[33] B. Niu and G. Tan. Modular control flow integrity. In *PLDI*, 2014.

[34] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, 2013.

[35] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS*, 1998.

[36] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[37] J. H. Saltzer and M. D. Schroeder. Formal verification of a realistic compiler. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[38] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *S&P*, 2015.

[39] D. Sehr, R. Muth, C. L. Biffle, V. Khimenko, E. Pasko, B. Yee, K. Schimpf, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *Usenix Security*, 2010.

[40] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani. Moat: Verifying confidentiality of enclave programs. In *CCS*, 2015.

[41] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.

[42] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[43] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, Jan. 1996.

[44] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*, 1993.

[45] J. Yang and K. G. Shin. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *VEE*, 2008.

[46] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[47] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *S&P*, 2009.

[48] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[49] L. Zhao, G. Li, B. D. Sutter, and J. Regehr. Armor: Fully verified software fault isolation. In *EMSOFT*, 2011.