ConfLLVM: A Compiler for Enforcing Data Confidentiality in Low-Level Code

Ajay Brahmakshatriya∗
MIT, USA
ajaybr@mit.edu

Piyus Kedia∗
IIIT Delhi, India
piyus@iiitd.ac.in

Derrick P. McKee∗
Purdue University, USA
mckee15@purdue.edu

Deepak Garg
MPI-SWS, Germany
dg@mpi-sws.org

Akash Lal
Microsoft Research, India
akashl@microsoft.com

Aseem Rastogi
Microsoft Research, India
aseemr@microsoft.com

Hamed Nemati
CISPA, Saarland University, Germany
hnemati@kth.se

Anmol Panda
Microsoft Research, India
t-anpand@microsoft.com

Pratik Bhatu∗
AMD, India
pbhatu@amd.com

Abstract
We present a compiler-based scheme to protect the confidentiality of sensitive data in low-level applications (e.g. those written in C) in the presence of an active adversary. In our scheme, the programmer marks sensitive data by lightweight annotations on the top-level definitions in the source code. The compiler then uses a combination of static dataflow analysis, runtime instrumentation, and a novel taint-aware form of control-flow integrity to prevent data leaks even in the presence of low-level attacks. To reduce runtime overheads, the compiler uses a novel memory layout.

We implement our scheme within the LLVM framework and evaluate it on the standard SPEC-CPU benchmarks, and on larger, real-world applications, including the NGINX webserver and the OpenLDAP directory server. We find that the performance overheads introduced by our instrumentation are moderate (average 12% on SPEC), and the programmer effort to port the applications is minimal.

1 Introduction
Many programs compute on private data: Web servers use private keys and serve private files, medical software processes private medical records, and many machine learning models are trained on private inputs. Bugs or exploited vulnerabilities in these programs may leak the private data to public channels that are visible to unintended recipients. For example, the OpenSSL buffer-overflow vulnerability Heartbleed [6] can be exploited to exfiltrate a web server’s private keys to the public network in cleartext. Generally speaking, the problem here is one of information flow control [28]: We would like to enforce that private data, as well as data derived from it, is never sent out on public channels unless it has been intentionally declassified by the program.

The standard solution to this problem is to use static dataflow analysis or runtime taints to track how private data flows through the program. While these methods work well in high-level, memory-safe languages such as Java [37, 40] and ML [46], the problem is very challenging and remains broadly open for low-level compiled languages like C that are not memory-safe. First, the cost of tracking taint at runtime is prohibitively high for these languages (see Section 9). Second, static dataflow analysis cannot guarantee data confidentiality because the lack of memory safety allows for buffer overflow and control-flow hijack attacks [1, 2, 4, 11, 49, 54], both of which may induce data flows that cannot be anticipated during a static analysis.

One possible approach is to start from a safe dialect of C (e.g. CCured [42], Deputy [24], or SoftBound [41]) and leverage existing approaches such as information-flow type systems for type-safe languages [32, 50]. The use of safe dialects, however, (a) requires additional annotations and program restructuring that cause significant programming overhead [38, 42], (b) are not always backwards compatible with legacy code, and (c) have prohibitive runtime overhead making them a non-starter in practical applications (see further discussion in Section 9).

In this paper, we present the first end-to-end, practical compiler-based scheme to enforce data confidentiality in C programs even in the presence of active, low-level attacks. Our scheme is based on the insight that complete memory
safety and perfect control-flow integrity (CFI) are neither sufficient nor necessary for preventing data leaks. We design a compiler that is able to guarantee data confidentiality without requiring these properties.

We accompany our scheme with formal security proofs and a thorough empirical evaluation to show that it can be used for real applications. Below, we highlight the main components of our scheme.

**Annotating private data.** We require the programmer to add private type annotations only to top-level function signatures and globals’ definitions to mark private data. The programmer is free to use all of the C language, including pointer casts, aliasing, indirect calls, varargs, variable length arrays, etc.

**Flow analysis and runtime instrumentation.** Our compiler performs standard dataflow analysis, statically propagating taint from global variables and function arguments that have been marked private by the programmer, to detect any data leaks. This analysis assumes the correctness of declared taints of each pointer. These assumptions are then protected by inserting runtime checks that assert correctness of the assumed taint at runtime. These checks are necessary to guard against incorrect casts, memory errors and low-level attacks. Our design does not require any static alias analysis, which is often hard to get right with acceptable precision.

**Novel memory layout.** To reduce the overhead associated with runtime checks, our compiler partitions the program’s virtual address space into a contiguous public region and a disjoint, contiguous private region, each with its own stack and heap. Checking the taint of a pointer then simply reduces to a range check on its value. We describe two partitioning schemes, one based on the Intel MPX ISA extension [7], and the other based on segment registers (Section 3). Our runtime checks do not enforce full memory safety: a private (public) pointer may point outside its expected object but our checks ensure that it always dereferences somewhere in the private (public) region. These lightweight checks suffice for protecting data confidentiality.

**Information flow-aware CFI.** Similar to memory safety, we observe that perfect CFI is neither necessary nor sufficient for information flow. We introduce an efficient, taint-aware CFI scheme. Our compiler instruments the targets of indirect calls with magic sequences that summarize the output of the static dataflow analysis at those points. At the source of the indirect calls, a runtime check ensures that the magic sequence at the target is taint-consistent with the output of the static dataflow analysis at the source (Section 4).

**Trusted components.** For selective declassification, as required for most practical applications, we allow the programmer to re-factor trusted declassification functions into a separate component, which we call \( T \). The remaining untrusted application, in contrast, is called \( U \). Code in \( T \) is not subject to any taint checks and can be compiled using a vanilla compiler. It can access all of \( U \)’s memory, specifically, it can perform declassification by copying data from \( U \)’s private region to \( T \)’s public region (e.g. after encrypting the data), \( T \) has its own separate stack and heap, and we re-use the range checks on memory accesses in \( U \) to prevent them from reading or writing to \( T \)’s memory. We give an example and general guidelines for refactoring a \( U \) and \( T \) in Section 2.

**Formal guarantees.** We have formalized our scheme using a core language of memory instructions (load and store) and control flow instructions (goto, conditional, and direct and indirect function call). We prove a termination-insensitive non-interference theorem for \( U \), assuming that the \( T \) functions it calls are non-interfering. In other words, we prove that if two public-equivalent configurations of \( U \) take a step each, then the resulting configurations are also public-equivalent. Our formal model shows the impossibility of sensitive data leaks even in the presence of features like aliasing and casting, and low-level vulnerabilities such as buffer overflows and ROP attacks (Appendix A).

**Implementation and Evaluation** We have implemented our compiler, ConfLLVM, as well as the complementary low-level verifier, ConfVerify, within the LLVM framework [36]. We evaluate our implementation on the standard SPEC-CPU benchmarks and three large applications—NGINX web server [10], OpenLDAP directory server [45], and a neural-network-based image classifier built on Torch [13, 14]. All three applications have private data—private user files in the case of NGINX, stored user passwords in OpenLDAP, and a model trained on private inputs in the classifier. In all cases, we are able to enforce confidentiality for the private data with a moderate overhead on performance and a small amount of programmer effort for adding annotations and declassification code.

## 2 Overview

**Threat model.** We consider C applications that work with both private and public data. Applications interact with the external world using the network, disk, and other channels. They communicate public data in clear, but want to protect the confidentiality of the private data by, for example, encrypting it before sending it out. However, the application could have logical or memory errors, or exploitable vulnerabilities that may cause private data to be leaked out in clear.

The attacker interacts with the application and may send carefully-crafted inputs that trigger bugs in the application. The attacker can also observe all the external communication of the application. Our goal is to prevent the private data of the application from leaking out in clear. Specifically, we address explicit information flow: any data directly derived from private data is also treated as private. While this addresses most commonly occurring exploits [44], optionally, our scheme can be used in a stricter mode where it disallows branching on private data, thereby preventing...
Partitioning \( \mathcal{U} \)'s memory. To enforce confidentiality in \( \mathcal{U} \), we minimally require the programmer to tell ConflLVM where private data enters and exits \( \mathcal{U} \). Since \( \mathcal{U} \) relies on \( \mathcal{T} \) for I/O and communication, the programmer does so by marking private data in the signatures of all functions exported from \( \mathcal{T} \) to \( \mathcal{U} \) with a new type qualifier \texttt{private} [29]. Additionally, to help ConflLVM’s analysis, the programmer must annotate private data in \( \mathcal{U} \)'s top-level definitions, i.e., globals, function signatures, and in \texttt{struct} definitions. These latter annotations within \( \mathcal{U} \) are \textit{not trusted}. Getting them wrong may cause a static error or runtime failure, but cannot leak private data. ConflLVM needs no other input from the programmer. Using a dataflow analysis (Section 5), it automatically infers which local variables carry private data. Based on this information, ConflLVM partitions \( \mathcal{U} \)'s memory into two regions, one for public and one for private data, with each region having its own stack and heap. A third region of memory, with its own heap and stack, is reserved for \( \mathcal{T} \)'s use.

In our example, the trusted annotated signatures of \( \mathcal{T} \) against which ConflLVM compiles \( \mathcal{U} \) are:

\[
\begin{align*}
\text{int} & \text{ recv} (\text{int} \ fd, \text{ char} *bbuf, \text{ int} \ buf\_size); \\
\text{int} & \text{ send} (\text{int} \ fd, \text{ char} *bbuf, \text{ int} \ buf\_size); \\
\text{void} & \text{ decrypt} (\text{char} *\text{ciphertext}, \text{private char} *\text{data}); \\
\text{void} & \text{ read\_passwd} (\text{char} *\text{uname}, \text{private char} *\text{pass}, \text{int} \ size);
\end{align*}
\]

while the untrusted annotations for \( \mathcal{U} \) are:

\[
\begin{align*}
\text{void} & \text{ handleReq} (\text{char} *\text{uname}, \text{private char} *\text{upasswd}, \text{char} *\text{name}, \text{char} *\text{out}, \text{int} \ out\_sz); \\
\text{int} & \text{ authenticate} (\text{char} *\text{uname}, \text{private char} *\text{upass}, \text{private char} *\text{pass});
\end{align*}
\]

ConflLVM automatically infers that, for example, \texttt{passwd} (line 4) is a \texttt{private} buffer. Based on this and \texttt{send}'s prototype, ConflLVM raises a compile-time error flagging the bug at line 10. Once the bug is fixed by the programmer (e.g., by removing the line), ConflLVM compiles the program and lays out the stack and heap data in their corresponding regions (Section 3). The remaining two bugs are prevented by runtime checks that we briefly describe next.

Runtime checks. ConflLVM inserts runtime checks to ensure that, (a) at runtime, the pointers belong to their annotated or inferred regions (e.g. a \texttt{private} \texttt{char} * actually points to the private region in \( \mathcal{U} \)), (b) \( \mathcal{U} \) does not read or write beyond its own memory (i.e. it does not read or write to \( \mathcal{T} \)'s memory), and (c) \( \mathcal{U} \) follows a \textsl{taint-aware} form of CFI that prevents circumvention of these checks and prevents data leaks due to control-flow attacks. In particular, the bugs on lines 14 and 16 in our example cannot be exploited due to check (a). We describe the details of these checks in Sections 3 and 4.

\( \mathcal{T} \) code is allowed to access all memory. However, \( \mathcal{T} \) functions must check their arguments to ensure that the data
void handleReq (char *uname, char *upasswd, char *fname, char *out, int out_size)
{
    char passwd[SIZE], fcontents[SIZE];
    read_password (uname, passwd, SIZE);
    if(!(authenticate (uname, upasswd, passwd))) {
        return;
    }
    // inadvertently copying the password to the log file
    send (log_file, passwd, SIZE);
    read_file (fname, fcontents, SIZE);
    // (out_size > SIZE) can leak passwd to out
    memcpy (out, fcontents, out_size);
    // a bug in the fmt string can print stack contents
    sprintf (out + SIZE, fmt, "Request complete");
}

#define SIZE 512

int main (int argc, char **argv)
{
    ... // variable declarations
    while (1) {
        n = recv(fd, buf, buf_size);
        parse(buf, uname, upasswd_enc, fname);
        decrypt (upasswd_enc, upasswd);
        handleReq (uname, upasswd, fname, out, size);
        format (out, size, buf, buf_size);
        send (fd, buf, buf_size);
    }
}

Figure 1. Request handling code for a web server

Figure 2. Workflow of our scheme and toolchain

generated by \( U \) has the correct sensitivity label. For example, the \texttt{read\_passwd} function would check that the range \([\text{pass}, \text{pass+size-1}]\) falls inside the private memory segment of \( U \). (Note that this is different from complete memory safety; \( T \) need not know the size of the \texttt{passwd} buffer.)

**Trusted Computing Base (TCB).** We have also designed and implemented a static verifier, \texttt{ConfVerify}, to confirm that a binary output by \texttt{ConfLLVM} has enough checks in place to guarantee confidentiality (Section 5). \texttt{ConfVerify} guards against bugs in the compiler. It allows us to extend the threat model to one where the adversary has full control of the compiler that was used to generate the binary of the untrusted code. To summarize, our TCB, and thus the security of our scheme, does not depend on the (large) untrusted application code \( U \) or the compiler. We only trust the (small) library code \( T \) and the static verifier. We discuss more design considerations for \( T \) in Section 8.

### 3 Memory Partitioning Schemes

\texttt{ConfLLVM} uses the programmer-supplied annotations, and with the help of type inference, statically determines the taint of each memory access (Section 5), i.e., for every memory load and store in \( U \), it infers if the address contains private or public data. It is possible for the type-inference to detect a problem (for instance, when a variable holding private data is passed to a method expecting a public argument), in which case, a type error is reported back to the programmer. On successful inference, \texttt{ConfLLVM} proceeds to compile \( U \) with a custom memory layout and runtime instrumentation. We have developed two different memory layouts as well as runtime instrumentation schemes. The schemes have different trade-offs, but they share the common idea – all private and all public data are stored in their own respective contiguous regions of memory and the instrumentation ensures that at runtime each pointer respects its statically inferred taint. We describe these schemes next.

**MPX scheme.** This scheme relies on the Intel MPX ISA extension [7] and uses the memory layout shown in Figure 3b. The memory is partitioned into a public region and a private region, each with its own heap, stack and global segments. The two regions are laid out contiguously in memory. Their ranges are stored in MPX bounds registers (\texttt{bnd0} and \texttt{bnd1}). Each memory access is preceded with MPX instructions...
(bndcu and bndcl) that check their first argument against the
(upper and lower) bounds of their second argument. The
concrete values to be stored in the bounds registers are
determined at load time (Section 6).

The user selects the maximum stack size \texttt{offset} at com-
pile time (at most \(2^{31} - 1\)). The scheme maintains the public and
private stacks in lock-step: their respective top-of-stack are
always at offset \texttt{offset} to each other. For each function

call, the compiler arranges to generate a frame each on the
public and the private stacks. Spilled \texttt{private} local variables
and \texttt{private} arguments are stored on the private stack; ev-

everything else is on the public stack. Consider the procedure
in Figure 4a. The generated (unoptimized) assembly under the
MPX scheme (using virtual registers for simplicity) is
shown in Figure 4c. ConvrLLVM automatically infers that \(x\)
is a \texttt{private int} and places it on the private stack, whereas
\(y\) is kept on the public stack. The stack pointer \texttt{rsp}
points to the top of the public stack. Because the two stacks are
kept at constant \texttt{offset} to each other, \(x\) is accessed simply
as \texttt{rsp+4+offset}. The code also shows the instrumented MPX
bound check instructions.

**Segmentation scheme.** x64 memory operands are in the
form \([\text{base} + \text{index} * \text{scale} + \text{displacement}]\), where \text{base}
and \text{index} are 64-bit unsigned registers, \text{scale} is a constant with
maximum value of 8, and \text{displacement} is a 32-bit signed con-
stant. The architecture also provides two segment registers
\text{fs} and \text{gs} for the \text{base} address computation; conceptually,
\text{fs:base} simply adds \text{fs} to \text{base}.

We use these segment registers to store the lower bounds
of the public and private memory regions, respectively, and
prefix the base of memory operands with these registers. The
public and private regions are separated by (at least) 36GB
of guard space (unmapped pages that cause a fault when
accessed). The guard sizes are chosen so that any memory
operand whose \text{base} is prefixed with \text{fs} cannot escape the
public segment, and any memory operand prefixed with \text{gs}
cannot escape the private segment (Figure 3a).

The segments are each aligned to a 4GB boundary. The
usable space within each segment is also 4GB. We access the
\text{base} address stored in a 64-bit register, say a private
value stored in \text{rax}, as \text{fs+eax}, where \text{eax} is the lower 32 bits
of \text{rax}. Thus, in \text{fs+eax}, the lower 32 bits come from \text{eax}
and the upper 32 bits come from \text{fs} (because \text{fs} is 4GB aligned).
Further, the index register is also constrained to use lower
32 bits only. This implies that the maximum offset within
a segment that \text{U} can access is 38GB \((4 + 4 * 8 + 2)\). This is
rounded up to 40GB for 4GB alignment, with 4GB of usable
space and 36GB of guard space. Since the \text{displacement} value
can be negative, the maximum negative offset is 2GB, for
which we have the guard space below the public segment.

The usable parts of the segments are restricted to 4GB
because it is the maximum addressable size using a single 32
bit register. This restriction also ensures that we don’t have
to translate \text{U} pointers when the control is passed to \text{T}, thus
avoiding the need to change or recompile \text{T}. Generated code
for our example under this scheme is shown in Figure 4b.

The figure uses the convention that \text{e}_1 \text{(resp., \text{esp})}
represents the lower 32 bits of the register \text{r}_1 \text{(resp., \text{rsp})}. The public
and private stacks are still maintained in lock-step. Taking
the address of a private stack variable requires extra support:
the address of variable \(x\) in our example is \text{rsp+*size}, where
\text{size} is the total segment size (40GB).

The segmentation scheme has a lower runtime overhead
than the MPX scheme as it avoids doing bound-checks (Sec-

tion 7.1). However, it restricts the segment size to 4GB.

**Multi-threading support.** Both our schemes support multi-
threading. All inserted runtime checks (including those in
Section 4) are thread-safe because they check values of regis-
ters. However, we do need additional support for thread-local
storage (TLS). Typically, TLS is accessed via the segment reg-


isters: the base of TLS is obtained at a constant offset from
\text{gs}. The operating system takes care of setting \text{gs} on a per-
thread basis. However, \text{U} and \text{T} operate in different trust
domains, thus they cannot share the same TLS buffer.

We let \text{T} continue to use \text{gs} for accessing its own TLS.
ConvrLLVM changes the compilation of \text{U} to access TLS in a
different way. The multiple (per-thread) stacks in \text{U} are
all allocated inside the stack regions; the public and private
stacks for each thread are still at a constant offset to each
other. Each thread stack is, by default, of maximum size 1MB
and its start is aligned to a 1MB boundary (configurable at
compile time). We keep the per-thread TLS buffer at the
beginning of the stack. \text{U} simply masks the lower 20-bits of
\text{rsp} to zeros to obtain the base of the stack and access TLS.

The segment-register scheme further requires switching
of the \text{gs} register as control transfers between \text{U} and \text{T}. We
use appropriate wrappers to achieve this switching, how-
ever \text{T} needs to reliably identify the current thread-id when
called from \text{U} (so that \text{U} cannot force two different threads
to use the same stack in \text{T}). ConvrLLVM achieves this by
instrumenting an inlined-version of the \_chkstk routine\footnote{https://msdn.microsoft.com/en-us/library/ms648426.aspx}
to make sure that \text{rsp} does not escape its stack boundaries.

### 4 Taint-aware CFI

We design a custom, taint-aware CFI scheme to ensure that
an attacker cannot alter the control flow of \text{U} to circumvent
the instrumented checks and leak sensitive data.

Typical low-level attacks that can hijack the control flow
of a program include overwriting the return address, or the
targets of function pointers and indirect jumps. Existing
approaches use a combination of shadow stacks or stack can-
aries to prevent overwriting the return address, or use fine-
grained taint tracking to ensure that the value of a function
pointer is not derived from user (i.e. attacker-controlled) in-

inputs [26, 35, 57]. While these techniques may prevent certain
The x64 calling convention for Windows that has 4 magic sequences.

bits, the return site taint is padded with four zero bits. The signature. To keep the length of the sequences uniform at 59 bits, the register taints expected at the target address match the current register taints (e.g., when the rax register holds a private value then a ret can only go to a site that expects a private return value). These suffice for our goal of data confidentiality. Our scheme does not need to ensure, for instance, that a return matches the previous call.

Callee-save registers are also live at function entry and exit and their taints cannot be determined statically by the compiler. ConfLLVM forces their taint to be public by making the caller save and clear all the private-tainted callee.saved registers before making a call. All dead registers (e.g. unused argument registers and caller-saved registers at the beginning of a function) are conservatively marked private to avoid accidental leaks. We note that our scheme can be extended easily to support other calling conventions.

Consider the following U:

```c
private int add (private int x) { return x + 1; }
private int incr (private int *p, private int x) { int y = add (x); *p = y; return *p; }
```

The compiled code for these functions is instrumented with magic sequences as follows. The 5 taint bits for the add procedure are 11111 as its argument x is private, unused argument registers are conservatively treated as private, and its return type is also private. On the other hand, the taint bits for incr are 01111 because its first argument is a public pointer (note that the private annotation on the argument is on the int, not the pointer), second argument is private, unused argument registers are private, and the return value is also private. For the return site in incr after the call to add, the taint bits are 00001 to indicate the private return value register (with 4 bits of padding). The sample instrumentation is as shown below:

```c
#define #M_call #111111
add:
```
We implemented

We can conceptually support them as long as the jump tables

We use the bitwise negation of

At indirect calls, the instrumentation is similar to that of a

U

Inside

add

work [36], targeting Windows and Linux x64 platforms. It is

possible to implement all of ConfLLVM’s instrumentation

using simple branching instructions available on all plat-

forms, but we rely on x64’s MPX support or x64’s segment

registers to optimize runtime performance. We leave the

optimizations on other architectures as future work.

5 Implementation

We implemented ConfLLVM as part of the LLVM frame-

work [36], targeting Windows and Linux x64 platforms. It is

possible to implement all of ConfLLVM’s instrumentation

using simple branching instructions available on all plat-

forms, but we rely on x64’s MPX support or x64’s segment

registers to optimize runtime performance. We leave the

optimizations on other architectures as future work.

5.1 ConfLLVM

Compiler front-end. We introduce a new type qualifier,

private, in the language that the programmers can use to

annotate sensitive data. For example, a private integer-typed

variable can be declared as private int x, and a (public)

pointer pointing to a private integer as private int *p. The

struct fields inherit their outermost annotation from the cor-

responding struct-type variable. For example, consider a

declaration struct st { private int *p; }, and a variable x of

type struct st. Then x.p inherits its qualifier from x: if x is de-

clared as private st x, then x.p is a private pointer pointing
to a private integer. We follow the same convention with

unions as well: all fields of a union inherit their outermost

annotation from the union-typed variable.

This convention ensures that despite the memory parti-
tioning into public and private regions, each object is laid out
together in memory in only one of the regions. It does,

however, carry the limitation that one cannot have structures

or unions whose fields have mixed outermost annotations,

e.g., a struct with a public int field as well as a private int

field, or a union over two structures, one with a public int

field and another with a private int field. In all such cases,

the programmer must restructure their code, often by simply

introducing one level of indirection in the field (because the

type constraints only apply to the outermost level).

We modified the Clang [3] frontend to parse the private

type qualifier and generate the LLVM Intermediate Represen-
tation (IR) instrumented with this additional metadata. Once

the IR is generated, ConfLLVM runs standard LLVM IR op-
timizations that are part of the LLVM toolchain. Most of the

optimizations work as-is and don’t require any change. Op-
timizations that change the metadata (e.g. remove-dead-args
changes the function signatures), need to be modified. While

we found that it is not much effort to modify an optimization,

we chose to modify only the most important ones in order

to bound our effort. We disable the remaining optimizations

in our prototype.

LLVM IR and type inference. After all the optimizations are

run, our compiler runs a type qualifier inference [29] pass

ever the IR. This inference pass propagates the type quali-

fier annotations to local variables, and outputs an IR where

all the intermediate variables are optionally annotated with

private qualifiers. The inference is implemented using a stan-
dard algorithm based on generating subtyping constraints on

dataflows, which are then solved using an SMT solver [27].

If the constraints are unsatisfiable, an error is reported to the

user. We refer the reader to [29] for details of the algorithm.

After type qualifier inference, ConfLLVM knows the taint of
each memory operand for load and store instructions. With

a simple dataflow analysis [17], the compiler statically deter-
mines the taint of each register at each instruction.

Register spilling and code generation. We made the reg-

ister allocator taint-aware: when a register is to be spilled on
the stack, the compiler appropriately chooses the private or the public stack depending on the taint of the register. Once the LLVM IR is lowered to machine IR, ConfLLVM emits the assembly code inserting all the checks for memory bounds and taint-aware CFI.

**MPX Optimizations.** ConfLLVM optimizes the bounds-checking in the MPX scheme. MPX instruction operands are identical to x64 memory operands, therefore one can check bounds of a complex operand using a single instruction. However, we found that bounds-checking a register is faster than bounds-checking a memory operand (perhaps because using a memory operand requires an implicit lea). Consequently, ConfLLVM uses instructions on a register (as opposed to a complex memory operand) as much as possible for bounds checking. It reserves 1MB of space around the public and private regions as guard regions and eliminates the *displacement* from the memory operand of a check if its absolute value is smaller than $2^{30}$. Usually the displacement value is a small constant (for accessing structure fields or doing stack accesses) and this optimization applies to a large degree. Further, by enabling the _chkstk enforcement for the MPX scheme also (Section 3), ConfLLVM eliminates checks on stack accesses altogether because the rsp value is bound to be within the public region (and rspOFFSET is bound to be within the private region).

ConfLLVM further coalesces MPX checks within a basic block. Before adding a check, it confirms if the same check was already added previously in the same block, and there are no intervening call instructions or subsequent modifications to the base or index registers.

**Implicit flows.** By default, ConfLLVM tracks explicit flows. It additionally produces a warning when the program branches on private data, indicating the presence of a possible implicit flow. Such a branch is easy to detect: it is a conditional jump on a private-tainted register. Thus, if no such warning is produced, then the application indeed lacks both explicit and implicit flows. Additionally, we also allow the compiler to be used in an all-private scenario where all data manipulated by $U$ is tainted private. In such a case, the job of the compiler is easy: it only needs to limit memory accesses in $U$ to its own region of memory. Implicit flows are not possible in this mode. None of our applications (Section 7) have implicit flows.

### 5.2 ConfVerify

ConfVerify checks that a binary produced by ConfLLVM has the required instrumentation in place to guarantee that there are no (explicit) private data leaks. The design goal of ConfVerify is to guard against bugs in ConfLLVM; it is not meant for general-purpose verification of arbitrary binaries. ConfVerify actually helped us catch bugs in ConfLLVM during its development.

ConfVerify is only 1500 LOC in addition to an off-the-shelf disassembler that it uses for CFG construction. (Our current implementation uses the LLVM disassembler.) This is three orders of magnitude smaller than ConfLLVM’s 5MLOC. Moreover, ConfVerify is much simpler than ConfLLVM; ConfVerify does not include register allocation, optimizations, etc. and uses a simple dataflow analysis to check all the flows. Consequently, it provides a higher degree of assurance for the security of our scheme.

**Disassembly.** ConfVerify requires the unique prefixes of magic sequences (Section 4) as input and uses them to identify procedure entries in the binary. It starts disassembling the procedures and constructs their control-flow graphs (CFG). ConfVerify assumes that the binary satisfies CFI, which makes it possible to reliably identify all instructions in a procedure. If the disassembly fails, the binary is rejected. Otherwise, ConfVerify checks its assumptions: that the magic sequences were indeed unique in the procedures identified and that they have enough CFI checks.

**Data flow analysis and checks.** Next, ConfVerify performs a separate dataflow analysis on every procedure’s CFG to determine the taints of all the registers at each instruction. It starts from the taint bits of the magic sequence preceding the procedure. It looks for MPX checks or the use of segment registers to identify the taints of memory operands; if it cannot find a check in the same basic block, the verification fails. For each store instruction, it checks that the taint of the destination operand matches the taint of the source register. For direct calls, it checks that the expected taints of the arguments, as encoded in the magic sequence at the callee, matches the taints of the argument registers at the call site (this differs from ConfLLVM which uses functions signatures). For indirect control transfers (indirect calls and ret), ConfVerify confirms that there is a check for the magic sequence at the target site and that its taint bits match the inferred taints for registers. After a call instruction, ConfVerify picks up taint of the return register from the magic sequence (there should be one), marks all caller-save registers as private, and callee-save registers as public (following ConfLLVM’s convention).

**Additional checks.** ConfVerify additionally makes sure that a direct or a conditional jump can only go a location in the same procedure. ConfVerify rejects a binary that has an indirect jump, a system call, or if it modifies a segment register. ConfVerify also confirms correct usage of _chkstk to ensure that rsp is kept within stack bounds. For the segment-register scheme, ConfVerify additionally checks that each memory operand uses only the lower 32-bits of registers.

**Formal analysis.** Although ConfVerify is fairly simple, to improve our confidence in its design, we built a formal model consisting of an abstract assembly language with essential features like indirect calls, returns, as well as ConfVerify’s checks. We prove formally that any program that
passes ConfVerify’s checks satisfies the standard information flow security property of termination-insensitive noninterference [50]. In words, this property states that data leaks are impossible. We defer the details of the formalization to Appendix A.

6 Toolchain

This section describes the overall flow to launch an application using our toolchain.

Compiling $U$ using ConfLLVM. The only external functions for $U$’s code are those exported by $T$. $U$’s code is compiled with an (auto-generated) stub file, that implements each of these $T$ functions as an indirect jump from a table externals, located at a constant position in $U$ (e.g., jmp (externals + offset)$_i$ for the $i$-th function). The table externals is initialized with zeroes at this point, and ConfLLVM links all the $U$ files to produce a $U$ dll.

The $U$ dll is then post-processed to patch all the references to globals, so that they correspond to the correct (private or public) region. The globals themselves are relocated by the loader. The post-processing pass also sets the 59-bit prefix for the magic sequences (used for CFI, Section 4). We find these sequences by generating random bit sequences and checking for uniqueness; usually a small number of iterations are sufficient.

Wrappers for $T$ functions. For each of the functions in $T$’s interface exported to $U$, we write a small wrapper that: (a) performs the necessary checks for the arguments (e.g. the send wrapper would check that its argument buffer is contained in $U$’s public region), (b) copies arguments to $T$’s stack, (c) switches gs, (d) switches rsp to $T$’s stack, and (e) calls the underlying $T$ function (e.g. send in libc). On return, it the wrapper switches gs and rsp back and jumps to $U$ in a similar manner to our CFI return instrumentation. Additionally, the wrappers include the magic sequences similar to those in $U$ so that the CFI checks in $U$ do not fail when calling $T$. These wrappers are compiled with the $U$ dll, and the output dll exports the interface functions.

Loading the $U$ and $T$ dlls. When loading the $U$ and $T$ dlls, the loader: (1) populates the externals table in $U$ with addresses of the wrapper functions in $T$, (2) relocates the globals in $U$ to their respective, private or public regions, (3) sets the MPX bound registers for the MPX scheme or the segment registers for the segment-register scheme, and (4) initializes the heaps and stacks in all the regions, marks them non-executable, and jumps to the main routine.

Memory allocator. ConfLLVM uses a customized memory allocator to enclose the private and public allocations in their respective sections. We modified dlmalloc [5] to achieve this.

7 Evaluation

The goal of our evaluation is three-fold: (a) Quantify the performance overheads of ConfLLVM’s instrumentation, both for enforcing bounds and for enforcing CFI; (b) Demonstrate that ConfLLVM scales to large, existing applications and quantify changes to existing applications to make them ConfLLVM-compatible; (c) Check that our scheme actually stops confidentiality exploits in applications.

7.1 CPU benchmarks

We measured the overheads of ConfLLVM’s instrumentation on the standard SPEC CPU 2006 benchmarks [12]. We treat the code of the benchmarks as untrusted (in $U$) and compile it with ConfLLVM. We use the system’s native libc, which is treated as trusted (in $T$). These benchmarks use no private data, so we added no annotations to the benchmarks, which makes all data public by default. Nonetheless, the code emitted by ConfLLVM ensures that all memory accesses are actually in the public region, it enforces CFI, and switches stacks when calling $T$ functions, so this experiment accurately captures ConfLLVM’s overheads. We ran the benchmarks in the following configurations.

- **Base**: Benchmarks compiled with vanilla LLVM, with O2 optimizations. This is the baseline for evaluation.
- **BaseOA**: Benchmarks compiled with vanilla LLVM but running with our custom allocator.
- **Ourbare**: Compiled with ConfLLVM, but without any runtime instrumentation. However, all optimizations unsupported by ConfLLVM are disabled, the memories of $T$ and $U$ are separated and, hence, stacks are switched in calling $T$ functions from $U$.
- **OurCFI**: Like Ourbare but additionally with CFI instrumentation, but no memory bounds enforcement.
- **OurMPX**: Full ConfLLVM, memory-bounds checks use MPX.
- **OurSEG**: Full ConfLLVM, memory-bounds checks use segmentation.

Briefly, the difference between OurCFI and Ourbare is the cost of our CFI instrumentation. The difference between OurMPX (resp. OurSEG) and OurCFI is the cost of enforcing bounds using MPX (resp. segment registers).

We ran all of the C benchmarks of SPEC CPU 2006, except perlBench, which uses fork that we currently do not support. All benchmarks were run on a Microsoft Surface Pro-4 Windows 10 machine with an Intel Core i7-6650U 2.20 GHz 64-bit processor with 2 cores (4 logical cores) and 8 GB RAM.

Figure 5 shows the results of our experiment. The overhead of ConfLLVM using MPX (OurMPX) is up to 74.03%, while that of ConfLLVM using segmentation (OurSEG) is up to 24.5%. As expected, the overheads are almost consistently significantly lower when using segmentation than when using MPX. Looking further, some of the overhead (up to 10.2%) comes from CFI enforcement (OurCFI−Ourbare).
average CFI overhead is 3.62%, competitive with best known techniques [26]. Stack switching and disabled optimizations (Ourbare) account for the remaining overhead. The overhead due to our custom memory allocator (BaseOA) is negligible and, in many benchmarks, the custom allocator improves performance.

We further comment on some seemingly odd results. On mcf, the cost of CFI alone (OurCFI, 4.17%) seems to be higher than that of the full MPX-based instrumentation (OurMPX, 4.02%). We verified that this is due to an outlier in the OurCFI experiment. On hmmer, the overhead of Ourbare is negative because the optimizations that ConfLLVM disables actually slow it down. Finally, on mile, the overhead of ConfLLVM is negative because this benchmark benefits significantly from the use of our custom memory allocator. Indeed, relative to BaseOA, the remaining overheads follow expected trends.

7.2 Web server: NGINX

Next, we demonstrate that our method and ConfLLVM scale to large applications. We cover three applications in this and the next two sections. We first use ConfLLVM to protect logs in NGINX, the most popular web server among high-traffic websites [10]. NGINX has a logging module that logs time stamps, processing times, etc. for each request along with request metadata such as the client address and the request URI. An obvious confidentiality concern is that sensitive content from files being served may leak into the logs due to bugs. We use ConfLLVM to prevent such leaks.

We annotate NGINX’s codebase to place OpenSSL in T, and the rest of NGINX, including all its request parsing, processing, serving, and logging code in U. The code in U is compiled with ConfLLVM (total 124,001 LoC). Within U, we mark everything as private, except for the buffers in the logging module that are marked as public. To log request URIs, which are actually private, we add a new encrypt_log function to T that U invokes to encrypt the request URI before adding it to the log. This function encrypts using a key that is isolated in T’s own region. The key is pre-shared with the administrator who is authorized to read the logs. The encrypted result of encrypt_log is placed in a public buffer.

In total, we added or modified 160 LoC in NGINX (0.13% of NGINX’s codebase) and added 138 LoC to T.

Our goal is to measure the overhead of ConfLLVM on the maximum sustained throughput of NGINX. We run our experiments in 6 configurations: Base, Ourbare, OurMPX-Sep. Of these, Base, Ourbare, and OurMPX-Sep are the same as those in Section 7.1. We use MPX for bounds checks (OurMPX), as opposed to segmentation, since we know from Section 7.1 that MPX is worse for ConfLLVM. Ourbare is like Ourbare (compiled with ConfLLVM without any instrumentation), but also does not separate memories for T and U. OurMPX-Sep includes all instrumentation, but does not separate stacks for private and public data. Briefly, the difference between Ourbare and OurMPX-Sep is the overhead of separating T’s memory from U’s and switching stacks on every call to T, while the difference between OurMPX and OurMPX-Sep is the overhead of increased cache pressure from having separate stacks for private and public data.

We host NGINX version 1.13.12 on an Intel Core i7-6700 3.40GHz 64-bit processor with 4 cores (8 logical cores), 32 GB RAM, and a 10 Gbps Ethernet card, running Ubuntu v16.04.1 with Linux kernel version 4.13.0-38-generic. Hyperthreading was disabled in order to reduce experimental noise. NGINX is configured to run a single worker process pinned to a core. We connect to the server from two client machines using the wrk2 tool [15], simulating a total of 32 concurrent clients. Each client makes 10,000 sequential requests, randomly chosen from a corpus of 1,000 files of the same size (we vary the file size across experiments). The files are served from a RAM disk. This saturates the CPU core hosting NGINX in all setups.

Figure 6 shows the steady-state throughputs in the six configurations for file sizes ranging from 0 to 40 KB. For file sizes beyond 40 KB, the 10 Gbps network card saturates in the base line before the CPU, and the excess CPU cycles absorb our overheads.

Overall, ConfLLVM’s overhead on sustained throughput ranges from 3.25% to 29.32%. The overhead is not monotonic in file size or base line throughput, indicating that there are compensating effects at play. For large file sizes,
the relative amount of time spent outside $\mathcal{U}$, e.g., in the kernel in copying data, is substantial. Since code outside $\mathcal{U}$ is not subject to our instrumentation, our relative overhead falls for large file sizes (>10 KB here) and eventually tends to zero. The initial increase in overhead up to file size 10 KB comes mostly from the increased cache pressure due to the separation of stacks for public and private data (the difference $\text{Our}_{\text{MPX}} - \text{Our}_{\text{MPX-Sep}}$). This is unsurprising: As the file size increases, so does the cache footprint of $\mathcal{U}$. In contrast, the overheads due to CFI (difference $\text{Our}_{\text{CFI}} - \text{Our}_{\text{bare}}$) and the separation of the memories of $\mathcal{T}$ and $\mathcal{U}$ (difference $\text{Our}_{\text{bare}} - \text{Our}_{\text{IMem}}$) are relatively constant for small file sizes.

Note that these overheads are moderate and we expect that they can be reduced further by using segmentation in place of MPX for bounds checks.

### 7.3 OpenLDAP

Next, we apply ConflLVM to OpenLDAP [45], an implementation of the Lightweight Directory Access Protocol (LDAP) [53]. LDAP is a standard for organizing and accessing hierarchical information. Here, we use ConflLVM to protect root and user passwords stored in OpenLDAP version 2.4.45. By default, the root password (which authorizes access to OpenLDAP’s entire store) and user passwords are all stored unencrypted. We added new functions to encrypt and decrypt these passwords, and modified OpenLDAP to use these functions prior to storing and loading passwords, respectively. The concern still is that OpenLDAP might leak in-memory passwords without encrypting them. To prevent this, we treat all of OpenLDAP as untrusted ($\mathcal{U}$), and protect it by compiling it with ConflLVM. The new cryptographic functions are in $\mathcal{T}$. Specifically, decryption returns its output in a private buffer, so ConflLVM prevents $\mathcal{U}$ from leaking it. The part we compile with ConflLVM is 300,000 lines of C code, spread across 728 source files. Our modifications amount to 52 new LoC for $\mathcal{T}$ and 100 edited LoC in $\mathcal{U}$. Together, these constitute about 0.5% of the original codebase.

We configure OpenLDAP as a multi-threaded server (the default) with a memory-mapped backing store (also the default), and simple username/password authentication. We use the same machine as for the SPEC CPU benchmarks (Section 7.1) to host an OpenLDAP server configured to run 6 concurrent threads. The server is pre-populated with 10,000 random directory entries. All memory-mapped files are cached in memory before the experiment starts.

In our first experiment, 80 concurrent clients connected to the server from another machine over a 100Mbps direct Ethernet link issue concurrent requests for directory entries that do not exist. Across three trials, the server handles on average 26,254 and 22,908 requests per second in the baseline (Base) and ConflLVM using MPX (Our$_{\text{MPX}}$). This corresponds to a throughput degradation of 12.74%. The server CPU remains nearly saturated throughout the experiment. The standard deviations are very small (1.7% and 0.2% in Base and Our$_{\text{MPX}}$, respectively).

Our second experiment is identical to the first, except that 60 concurrent clients issue small requests for entries that exist on the server. Now, the baseline and ConflLVM handle 29,698 and 26,895 queries per second, respectively. This is a throughput degradation of 9.44%. The standard deviations are small (less than 0.2%).

The reason for the difference in overheads in these two experiments is that OpenLDAP does less work in $\mathcal{U}$ looking for directory entries that exist than it does looking for directory entries that don’t exist. Again, the overheads of ConflLVM’s instrumentation are only moderate and can be reduced further by using segmentation in place of MPX.

### 7.4 ConflLVM with Intel SGX

Hardware features, such as the Intel Software Guard Extensions (SGX) [25], allow isolating sensitive code and data into a separate enclave, which cannot be read or written directly from outside. Although this affords strong protection (even against a malicious operating system), bugs within the isolated code can still leak sensitive data out from the enclave. This risk can be mitigated by using ConflLVM to compile the code that runs within the enclave.

We experimented with Privado [60], a system that performs image classification by running a pre-trained model on a user-provided image. Privado treats both the model (i.e., its parameters) and the user input as sensitive information. These appear unencrypted only inside an enclave. Privado contains a port of Torch [13, 14] made compatible with Intel SGX SDK. The image classifier is an eleven-layer neural network (NN) that categorizes images into ten classes.
We treat both Torch and the NN as untrusted code $\mathcal{U}$ and compile them using ConflLVM. We mark all data in the enclave private, and map $\mathcal{U}$’s private region to a block of memory within the enclave. $\mathcal{U}$ contains roughly 36K Loc. The trusted code ($\mathcal{T}$) running inside the enclave only consists of generic application-independent logic: the enclave SDK, our memory allocator, stubs to switch between $\mathcal{T}$ and $\mathcal{U}$, and a declassifier that communicates the result of image classification (which is a private value) back to the host running outside the enclave. Outside the enclave, we deploy a web server that takes encrypted images from clients and classifies them inside the enclave. The use of ConflLVM crucially reduces trust on the application-sensitive logic that is all contained in $\mathcal{U}$: ConflLVM enforces that the declassifier is the only way to communicate private information outside the enclave.

We ran this setup on an Intel Core i7-6700 3.40GHz 64-bit processor with an Ubuntu 16.04 OS (Linux 4.15.0-34-generic kernel). We connect to our application from a client that issues 10,000 sequential requests to classify small (3 KB) files, and measure the response time per image within the enclave (thus excluding latencies outside the enclave, which are common to the baseline and our enforcement). We do this in 5 configurations of Section 7.1: Base, Base-OA, Ourbare, Our-CFI, and Our-MPX. Figure 7 shows the average response time for classifying an image in each of these five configurations. For Base, we use the optimization level O2, while the remaining configurations use O2 except for two source files (out of 13) where a bug in ConflLVM (an optimization pass of O2 crashes) forces us to use O0. This means that the overheads reported in Figure 7 are higher than actual and, hence, conservative.

The overhead of Our-MPX is 26.87%. This is much lower than many of the latency experiments of Section 7.1. This is because, in the classifier, a significant amount of time (almost 70%) is spent in a tight loop, which contains mostly only floating point instructions and our instrumentation’s MPX bound-check instructions. These two classes of instructions can execute in parallel on the CPU, so the overhead of our instrumentation is masked within the loop.

7.5 Data integrity and scaling with parallelism

The goal of our next experiment is two-fold: to verify that ConflLVM’s instrumentation scales well with thread parallelism, and to test that ConflLVM can be used to protect data integrity, not just confidentiality. We implemented a simple multi-threaded userspace library that offers standard file read and write functionality, but additionally provides data integrity by maintaining a Merkle hash tree of file system contents. A security concern is that a bug in the application or the library may clobber the hash tree to nullify integrity guarantees. To prevent this, we compile both the library and its clients using ConflLVM (i.e. as part of $\mathcal{U}$). All data within the client and the library is marked private. The only exception is the hash tree, which is marked public. As usual, ConflLVM prevents the private data from being written to public data structures accidentally, thus providing integrity for the hash tree. To write hashes to the tree intentionally, we place the hashing function in $\mathcal{T}$, allowing it to "declassify" data hashes as public.

We experiment with this library on a Windows 10 machine with an Intel i7-6700 CPU (4 cores, 8 hyper-threaded cores) and 32 GB RAM. Our client program creates between 1 and 6 parallel threads, all of which read a 2 GB file concurrently. The file is memory-mapped within the library and cached previously. This gives us a CPU-bound workload. We measure the total time taken to perform the reads in three configurations: Base, Our- and Our-MPX. Figure 8 shows the total runtime as a function of the number of threads and the configuration. Until the number of threads exceeds the number of cores (4), the absolute execution time (written above the Base bars) and relative overhead of both the MPX and segmentation schemes remains nearly constant, establishing linear scaling with the number of threads. The actual overhead of Our- is below 10% and that of Our-MPX is below 17% in all configurations.

7.6 Vulnerability-injection experiments

To test that ConflLVM stops data extraction vulnerabilities from being exploited, we hand-crafted three vulnerabilities. In all cases, compiling the vulnerable applications with ConflLVM (after adding suitable private annotations) prevented the vulnerabilities from being exploited.

First, we introduced a buffer-bounds vulnerability in the popular Mongoose web server [9] in the code path for serving an unencrypted file over http. The vulnerability transmits any amount of stale data from the stack, unencrypted. We
wrote a client that exploits the vulnerability by first requesting a private file, which causes some contents of the private file to be written to the stack in clear text, and then requesting a public file with the exploit. This leaks stale private data from the first request. Using ConflLVM on Mongoose with proper annotations stops the exploit since ConflLVM separates the private and public stacks. The contents of the private file are written to the private stack but the exploit reads from the public stack.

Second, we modified Minizip [8], a file compression tool, to explicitly leak the file encryption password to a log file. ConflLVM’s type inference detects this leak once we annotate the password as private. To make it harder for ConflLVM, we added several pointer type casts on the password, which make it impossible to detect the leak statically. But, then, the dynamic checks inserted by ConflLVM prevent the leak.

Third, we wrote a simple function with a format string vulnerability in its use of printf. printf is a vararg function, whose first argument, the format string, determines how many subsequent arguments the function tries to print. If the format string has more directives than the number of arguments, potentially due to adversary provided input, printf ends up reading other data from either the argument registers or the stack. If any of this data is private, it results in a data leak. ConflLVM prevents this vulnerability from being exploited if we include printf’s code in U: since printf tries to read all arguments into buffers marked public, the bounds enforcement of ConflLVM prevents it from reading any private data.

8 Discussion and Future Work

Currently, our scheme allows classifying data into two levels—public and private. It cannot be used for finer classification, e.g., to separate the private data of Alice, the private data of Bob and public data at the same time. We also do not support label polymorphism at present, although that can potentially be implemented using C++-like templates.

In our scheme, T is trusted and therefore must be implemented with care. ConflLVM guarantees that U cannot access T’s memory or jump to arbitrary points in T, so the only remaining attack surface for T is the API that it exposes to U. T must ensure that stringing together a sequence of these API calls cannot cause leaks. We recommend the following (standard) strategies. First, T should be kept small, mostly containing application-independent functionality, e.g., communication interfaces, cryptographic routines, and optionally a small number of libc routines (mainly for performance reasons), moving the rest of the code to U. Such a T can be re-used across applications and can be subject to careful audit/verification. Further, declassification routines in T must provide guarded access to U. For instance, T should disallow an arbitrary number of calls to a password checking routine to prevent probing attacks.

We rely on the absence of the magic sequence in T’s binary to prevent U from jumping inside T. We ensure this by selecting the magic string when the entire code of U and T is available. While dynamic loading in U can simply be disallowed, any dynamic loading in T must ensure that the loaded library does not contain the magic sequence. Since the (59-bits) magic sequence is generated at random, the chances of it appearing in the loaded library is minimal. A stronger defense is to instrument indirect control transfers in U to remain inside U’s own code.

ConflLVM supports callbacks from T to U with the help of trusted wrappers in U that return to a fixed location in T, where T can restore its stack and start execution from where it left off (or fail if T never called into U).

At present, ConflLVM only supports the C language. Implementing support for C++ remains as interesting future work. It requires integrating the private type qualifier with the C++ type system and ensuring that the C++ runtime and object system respects the user-intended taint flow.

9 Related Work

Our work bears similarities to Sinha et al. [57] who proposed a design methodology for programming secure enclaves (e.g., those that use Intel SGX instructions for memory isolation). The code inside an enclave is divided into U and L. U’s code is compiled via a special instrumenting compiler [23] while L’s code is trusted and may be compiled using any compiler. This is similar in principle to our U-T division. However, there are several differences. First, even the goals
are different: their scheme does not track taints; it only ensures that all unencrypted I/O done by \( U \) goes through \( L \), which encrypts all outgoing data uniformly. Thus, the application cannot carry out plain-text communication even on public data without losing the security guarantee. Second, their implementation does not support multi-threading (it relies on page protection to isolate \( L \) from \( U \)). Third, they maintain a bitmap of writeable memory locations for enforcing CFI, resulting in time and memory overheads. Our CFI is taint-aware and without these overheads. Finally, their verifier does not scale to SPEC benchmarks, whereas our verifier is faster and scales to all binaries that we have tried.

In an effort parallel to ours, Carr et al. [22] present DataShield, whose goal, like ConfLLVM’s, is information flow control in low-level code. However, there are several differences between DataShield and our work. First and foremost, DataShield itself only prevents non-control data flow attacks in which data is leaked or corrupted without relying on a control flow hijack. A separate CFI solution is needed to prevent leaks of information in the face of control flow hijacks. In contrast, our scheme incorporates a customized CFI that provides only the minimum necessary for information flow control. One of the key insights of our work is that (standard) CFI is neither necessary nor sufficient to prevent information flow violations due to control flow hijacks. Second, DataShield places blind trust in its compiler. In contrast, in our work, the verifier ConfVERIFY eliminates the need to trust the compiler ConfLLVM. Third, DataShield enforces memory safety at object-granularity on sensitive objects. This allows DataShield to enforce integrity for data invariants, which is mostly outside the scope of our work. However, as we show in Section 7.5, our work can be used to prevent untrusted data from flowing into sensitive locations, which is a different form of integrity.

Rocha et al. [48] and Banerjee et al. [18] use combination of hybrid and static methods for information flow control, but in memory- and type-safe languages like Java. Cimplifier by Rastogi et al. [47] tracks information flow only at the level of process binaries by using separate docker containers.

Region-based memory partitioning has been explored before in the context of safe and efficient memory management [30, 59], but not for information flow. In ConfLLVM, regions obviate the need for dynamic taint tracking [39, 51, 55]. TaintCheck [44] first proposed the idea of dynamic taint tracking, and forms the basis of Valgrind [43]. DECAF [33] is a whole system binary analysis framework including a taint-tracking mechanism. However, such dynamic taint trackers incur heavy performance overhead. For example, DECAF has an overhead of 600%. Similarly, TaintCheck can impose a 37x performance overhead for CPU-bound applications. Suh et al. [58] report less than 1% overheads for their dynamic taint-tracking scheme, but they rely on custom hardware.

Static analyses [20, 21, 31, 52] of source code can prove security-relevant criteria such as safe downcasts in C++, or the correct use of variadic arguments. When proofs cannot be constructed, runtime checks are inserted to enforce relevant policy at runtime. This is similar to our use of runtime checks, but the purposes are different.

Memory-safety techniques for C such as CCured [42] and SoftBound [41] do not provide confidentiality in all cases and already have overheads higher than those of ConfLLVM (see [22, Section 2.2] for a summary of the overheads). Techniques such as control flow integrity (CFI) [16] and code-pointer integrity (CPI) [35] prevent control flow hijacks but not all data leaks. While our new taint-aware CFI is an integral component of our enforcement, our goal of preventing data leaks goes beyond CFI and CPI. Our CFI mechanism is similar to Abadi et al. [16] and Zeng et al. [61] in its use of magic sequences, but our magic sequences are taint-aware.

References

whether the function is part of the trusted code or the untrusted function whose code describing the code of the function. The block-graph for an untrusted function \( f \) and its return value (see Section 4). For trusted functions additionally, there is a 64-bit magic sequence for each function.

Table 1. Command syntax

<table>
<thead>
<tr>
<th>Command Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>`cmd := ldr (reg, exp)</td>
</tr>
<tr>
<td>`if then else (exp, goto (exp), goto (exp))</td>
</tr>
<tr>
<td>`call_{u/r} f (exp'')</td>
</tr>
</tbody>
</table>

\[
\text{exp} := n \in Val | \text{reg} \in \text{Reg} | \text{reg} \oplus \text{exp} | \text{exp} \ominus b \text{exp} | \& f
\]


A Formal model of verifier

Conceptually, ConfVerify operates in two stages. First, it disassembles a binary, constructs its control flow graph (CFG) and re-infers the taints of registers at all program points. Second, it checks that the taints at the beginning and end of every instruction are consistent with the instruction’s semantics, and that other dynamic checks described in Section 5.2 are correctly inserted. Here, we formalize the second stage and show if a program passes those checks, then the program is secure in a formal sense. We assume that the disassembly and the reconstruction of the CFG from the first stage, both of which use an existing tool, are correct.

We model the disassembled CFG abstractly. For each function, the CFG has one bit of metadata, which indicates whether the function is part of the trusted code or the untrusted code (abstractly written \( T \) and \( U \), respectively). Additionally, there is a 64-bit magic sequence for each function, which encodes the taints of the function’s arguments and its return value (see Section 4). For trusted functions (whose code ConfVerify does not analyze), this is all the CFG contains. For untrusted functions (whose code ConfVerify does analyze), there is an additional block-graph describing the code of the function. The block-graph for an untrusted function \( f \) is a tuple \( G_f = (V_f, E) \) of nodes of \( f \) and edges \( E \subseteq V \times V \). The edges represent all direct control transfers. A node \( (\text{pc}, C, \Gamma, \Gamma') \in V_f \) consists of a program counter \( \text{pc} \), a single command (assembly instruction) \( C \), and register taints \( \Gamma \) and \( \Gamma' \) before and after the execution of \( C \).

Commands \( C \) are represented in an abstract syntax shown in Table 1. The auxiliary syntax of expressions consists of constants, which here can only be integers that may represent ordinary data or line numbers, standard binary and unary operators (ranged over by \( \ominus b \) and \( \oplus u \), respectively), and the special operation \& \( f \) which returns the \( \text{pc} \) of the starting instruction of function \( f \). We assume that operations used in expressions are total, i.e., expressions never get stuck.

Commands include register load \( ldr (\text{reg}, \text{exp}) \) and store \( str (\text{reg}, \text{exp}) \) where the input \( \text{exp} \) evaluates to a memory location, unconditional jumps \( goto (\text{exp}) \), conditionals ifthenelse, direct function calls \( call_{u/r} f (\text{exp}'') \), indirect function calls \( ical \), and return \( ret \). The subscript for \( call_{u/r} \) denotes if the command is used to invoke an untrusted functionality or a trusted one. An additional command \( assert \) models checks inserted by the compiler. The asserted expressions must be true, else the program halts in a special state \( \bot \). Figure 10 presents the operational semantic of the commands in Table 1.

Every CFG \( G \) has a designated entry function, similar to main(\( G \)) in C programs.

Dynamic semantics. A program/CFG \( G \) can be evaluated in a context of a data memory \( \mu : Val \rightarrow Val \), a register state \( \rho : Reg \rightarrow Val \) and a program counter \( \text{pc} \) which points to a node in the program’s CFG. The memory \( \mu \) is actually a union of two maps \( \mu_L \) and \( \mu_H \) over disjoint domains, representing the low and the high regions of memory of our scheme. Given a fixed \( G \), a configuration \( s \) of \( G \) is a record \( (\mu, \rho, (\sigma_H : \sigma_L), \text{pc}) \) of the entire state—the memory, register state, stack, and the program counter. We make a distinction between high \( (\sigma_H) \) and low \( (\sigma_L) \) stacks. We write \( (\cdot, i) \) to project a tuple to its \( i \)th component. Thus, if \( s = (\mu, \rho, (\sigma_H : \sigma_L), \text{pc}) \), then \( s.\text{pc} = \text{pc} \). We call a configuration \( s \) initial (final) if \( s.\text{pc} \) is the first (last) node of the entry function of \( G \).

Next, we define the dynamic semantics of a program as a transition relation \( s \rightarrow s' \) (Figure 9). We use \( e \models n \) to mean evaluation of an expression to its corresponding (concrete) value. Moreover, we extend the configuration \( s \) of \( G \) with an additional component \( v \) that represents the memory fragment available only to the trusted code and its protected against untrusted accesses, i.e., \( (v, \mu, \rho, (\sigma_H : \sigma_L), \text{pc}) \). Further, we introduce the record \( F = \{ f_i \rightarrow (n, M, \text{call}), i \in \mathcal{P} \} \) and extend the typing judgment with this record \( \overline{F} \), \( G + \Gamma(\text{pc}) \Gamma' \). \( \overline{F} \) keeps for each function its starting node in the CFG and the magic sequence associated with that function. We only note that calls to trusted functions, whose code is not modeled in the CFG, are represented via an external relation \( s \xrightarrow{f} s' \) that models the entire execution of the trusted
function $f$ and its effects on the configuration. The special transition $s \rightarrow \bot$ means that $s.pc$ contains an `assert` command, one of whose arguments evaluates to false. $\bot$ does not transition any further. Furthermore, the transition $s \rightarrow \xi$ is used to model adversarial behavior, for example, when the target of `icall` is not in the CFG. $\xi$ resembles a configuration where memory locations are loaded with random values.

**Security Analysis.** We formalize the checks that ConFFVerify makes via the judgment $G \vdash \Gamma(pc) \Gamma'$, which means that the command in the node labeled $pc$ in the CFG $G$ is consistent with the beginning and ending taints $\Gamma$ and $\Gamma'$, and that the checks from Section 5.2 corresponding to this command are satisfied. Rules for this judgment are shown in Figure 10. The function $\text{pred}(G, pc)$ returns the predecessors of the node labeled $pc$ in $G$. For $\ell \in \{L, H\}$, $\text{assert}(e \in \text{Dom}(\mu_\ell))$ represents the dynamic check that $e$ evaluates to an address in the domain of $\mu_\ell$, and the auxiliary judgment $\Gamma \vdash e : \ell$ means that the expression $e$ depends only on values with secrecy $\ell$ or lower.

The rules are mostly self-explanatory. The first rule, which is for the command `ldr(reg, e)`, says that if `reg` has taint $\ell_e$ after the command, then on all paths leading to this command, there must be a check that whatever $e$ evaluates to is actually pointing into $\mu_{\ell_e}$. The rule for `str` is similar. In the rules for indirect branching, we insist that the addresses of the branch targets have the taint $L$. Overall, our type system’s design is inspired by the flow-sensitive information flow type system of Hunt and Sands [34]. However, the runtime checks are new to our system.

We formally define call- and return-sites magic sequences based on the CFG structure. Let $\text{Cons}$ be the bit-concatenation function, then: (i) for the function entry node $\nu, M_{\text{call}} = \text{Cons}(\nu.\Gamma(\text{reg}_\nu))$, and (ii) for a given return address $\text{adr}$ and the node $\nu$ such that $\nu \in \text{pred}(G, \text{adr}), M_{\text{ret}} = \nu.\Gamma'(\text{reg}_\nu)$. We now turn to explain the meaning of rules in Figure 10.

**call** For the `call` statement we check that the expected taints of the arguments, as encoded in the magic sequence at the callee, matches the taints of the argument registers at the callsite. It is also worth noting that at runtime all return addresses are stored in the stack allocated for the low-security context.

**icall** For the indirect calls ConFFVerify confirms that the function pointer is low and that there is a check for the magic sequence at the target site and its taint bits match the inferred taints for registers. Note that since the check $\text{assert}(\ell_{[1-4]} \subseteq M_{\text{call}} \text{ and } e_f \in G \text{ and } (e_f \mapsto M_{\text{call}}) \in F)$ is a runtime condition, the pointer $e_f$ will be evaluated to the corresponding function name at the execution time and we can retrieve the magic string directly from $F$. The condition $e_f \in G$ ensures that the target of the call statement is a valid node in the CFG of the program.

**ret** Similar to indirect function call, for the `ret` command we check that there is a check for the magic sequence at the target site and that its taint bits match the inferred taints for registers. Additionally, ConFFVerify confirms that the return address on the stack has a magic signature that has a taint bit of inferred return type $\ell$ for the function. In this rule we use the function `top` with the standard meaning to manipulate the stack content. Again since $\text{assert}(\forall \nu' \in \text{pred}(G, \text{top}(\sigma_L)), \ell \subseteq \nu'.\Gamma'(\text{reg}_\nu))$ is checked at runtime, we will have access to the stack.

**ifthenelse** For this rule we require that the security level of conditional expression is not $H$. Checks on the `goto` and `ifthenelse` guarantee that the program flow is secret independent.

We say that a CFG $G$ is well-typed (passes ConfVerify’s checks), written $\vdash G$, when two conditions hold for all nodes $\nu$ in (untrusted functions in) $G$: 1) The node (locally) satisfies the type system. Formally, $G \vdash v \Gamma \{v.pc\} \nu.\Gamma'$, and 2) The ending taint of the node is consistent with the beginning taints of all its successors. Formally, for all nodes $\nu' \in \text{succ}(G, v.pc), v.\Gamma' \subseteq \nu'.\Gamma$, where $\text{succ}(G, pc)$ returns the successors of the node labeled $pc$ in $G$. We emphasize again that only untrusted functions are checked.

**Security theorem.** The above checks are sufficient to prove our key theorem: If a program passes ConfVerify, then assuming that its trusted functions don’t leak private data, the whole program does not leak private data, end-to-end. We formalize non-leakage of private data as the standard information flow property called termination-insensitive non-interference. Roughly, this property requires a notion of low equivalence of program configurations (of the same CFG $G$), written $s =_L s'$, which allows memories of $s$ and $s'$ to differ only in the private region. A program is noninterfering if it preserves $<_L$ for any two runs, except when one program halts safely (e.g., on a failed assertion). Intuitively, noninterference means that no information from the private part of the initial memory can leak into the public part of the final memory.

For our model, we define $s =_L s'$ to hold if: (i) $s$ and $s'$ point to the same command, i.e., $s.pc = s'.pc$, (ii) the contents of their low-stacks are equal, $s.\sigma_L = s'.\sigma_L$, (iii) for all low memory addresses $m \in \mu_L, s.\mu(m) = s'.\mu(m)$, (iv) for all registers $r$ such that $G(s.pc).\Gamma(r) = L, s.\rho(r) = s'.\rho(r)$.

The assumption that trusted code does not leak private data is formalized as follows.

**Assumption 1.** For all $s_0, s_1, s'_0$ such that $s_0 =_L s_1$, if $s_0 \leftrightarrow_f s'_0$ then $\exists s'_1, s_1 \leftrightarrow_f s'_1$ and $s'_0 =_L s'_1$.

Under this assumption on the trusted code, we can show the noninterference or security theorem. A necessary condition to show noninterference, however, is to ensure that no well-typed program can reach an ill-formed configuration.
Lemma 1. Suppose $\vdash G$, for all configurations $s$ of $G$ it holds that $s \rightarrow^* L$.

Lemma 1 rules out possible nondeterminism caused by adversarial behavior and allows to formalize the security theorem as follows.

**Theorem 1** (Termination-insensitive noninterference). Suppose $\vdash G$. Then, for all configurations $s_0, s'_0$ and $s_1$ of $G$ such that $s_0 \stackrel{L}{=} L$, $s_1$ and $s_0 \rightarrow^* s'_0$, then either $s_1 \rightarrow^* \bot$ or $\exists s'_1$. $s_1 \rightarrow^* s'_1$ and $s'_0 \stackrel{L}{=} L$.

When $s_0$ and $s'_0$ are initial and final configurations, respectively, then $s_1$ and $s'_1$ must also be initial and final configurations, so the theorem guarantees freedom from data leaks, end-to-end (modulo assertion check failures).

$C = \text{ldr}(\text{reg}, e)$ $\mu, \rho \vdash e \parallel n \ n \in (\text{Dom}(\mu_1) \cup \text{Dom}(\mu_2))$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho[\text{reg} \mapsto \mu(v)], [\sigma], pc + 1 \rangle$

$C = \text{ldr}(\text{reg}, e)$ $\mu, \rho \vdash e \parallel n \ n \notin (\text{Dom}(\mu_1) \cup \text{Dom}(\mu_2))$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \bot$

$C = \text{str}(\text{reg}, e)$ $\mu, \rho \vdash e \parallel n \ n \in (\text{Dom}(\mu_1) \cup \text{Dom}(\mu_2))$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [pc + 1]; \sigma_1 : \sigma_1 \mu, pc_f \rangle$

$C = \text{call}_f (e_1, \ldots, e_l)$ $\langle f_e \mapsto (pc_{cf}, \bot) \rangle \in F$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [pc + 1]; \sigma_1 : \sigma_1 \mu, pc_f \rangle$

$C = \text{icall}_f (e_1, \ldots, e_l)$ $\mu, \rho \vdash e \parallel pc_f \ \text{pc}_{cf} \in G$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [pc + 1]; \sigma_1 : \sigma_1 \mu, pc_f \rangle$

$C = \text{call}_f (e_1, \ldots, e_l)$ $\mu, \rho \vdash e \parallel pc_f \ \text{pc}_{cf} \notin G$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [pc + 1]; \sigma_1 : \sigma_1 \mu, pc_f \rangle$

$C = \text{ret} \ \text{adr} \notin G$

$\langle v, \mu, \rho, [\text{adr} : \sigma_1 \mu], pc \rangle \rightarrow \langle v, \mu, \rho, [\sigma_1 : \sigma_1 \mu, \text{adr}] \rangle$

$C = \text{ret} \ \text{adr} \in G$

$\langle v, \mu, \rho, [\text{adr} : \sigma_1 \mu], pc \rangle \rightarrow \bot$

$C = \text{goto}(e)$ $\mu, \rho \vdash e \parallel pc$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [\sigma], pc \rangle$

$C = \text{ifthenelse}(e_1, \text{goto}(e_1), \text{goto}(e_2))$ $\mu, \rho \vdash e \parallel T$ $\mu, \rho \vdash e_1 \parallel n_1$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [\sigma], n_1 \rangle$

$C = \text{ifthenelse}(e_1, \text{goto}(e_1), \text{goto}(e_2))$ $\mu, \rho \vdash e \parallel F$ $\mu, \rho \vdash e_2 \parallel n_2$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [\sigma], n_2 \rangle$

$C = \text{assert}(e)$ $\mu, \rho \vdash e \parallel T$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \langle v, \mu, \rho, [\sigma], pc + 1 \rangle$

$C = \text{assert}(e)$ $\mu, \rho \vdash e \parallel F$

$\langle v, \mu, \rho, [\sigma], pc \rangle \rightarrow \bot$

**Figure 9.** Operational semantics rules.
\[
\begin{align*}
C = \text{idr}(\text{reg}, e) & \quad \forall v \in \text{pred}(G, pc). \quad v.C = \text{assert}(e \in \text{Dom}(\mu_e)) \\
\Gamma, G + \Gamma \{ pc \} \Gamma[\text{reg} \mapsto \ell_e] & \\
C = \text{str}(\text{reg}, e) & \quad \Gamma \vdash \text{reg} : \ell_e \quad \ell_e \subseteq \ell_e \\
\forall v \in \text{pred}(G, pc), \quad v.C = \text{assert}(e \in \text{Dom}(\mu_e)) & \\
\Gamma, G + \Gamma \{ pc \} \Gamma & \\
C = \text{call}_\{ (e_1, \ldots, e_4) \} \ (f \mapsto (\sim, M_{\text{call}})) \in \mathcal{F} & \\
\Gamma + e_i : \ell_i \quad \text{for} \quad i = 1 \ldots 4 & \quad \ell_i[1-4] \subseteq M_{\text{call}} \\
\Gamma, G + \Gamma \{ pc \} \Gamma[\text{callee} \mapsto H, \text{callee} \mapsto L] & \\
C = \text{icall} e_f(e_1, \ldots, e_4) & \quad \Gamma + e_f : \ell_f \subseteq \ell \quad \Gamma + e_i : \ell_i \quad \text{for} \quad i = 1 \ldots 4 \\
\forall v \in \text{pred}(G, pc), \quad v.C = \text{assert}(e_1 \subseteq \ell_1 \\
& \quad \ell_1[1-4] \subseteq M_{\text{call}} \quad \text{and} \quad e_f \in G \\
& \quad (e_f \mapsto (\sim, M_{\text{call}})) \in \mathcal{F}) \\
\Gamma, G + \Gamma \{ pc \} \Gamma[\text{callee} \mapsto H, \text{callee} \mapsto L] & \\
C = \text{ret} & \quad \Gamma + \text{callee} \subseteq \ell \quad \Gamma + \text{reg}_0 : \ell \\
\forall v \in \text{pred}(G, pc), \quad v.C = \text{assert}(v' \in \text{pred}(G, \text{top}(\sigma_1)). \quad \ell \subseteq v'. \Gamma'(\text{reg}_0)) & \\
\Gamma, G + \Gamma \{ pc \} \Gamma & \\
C = \text{goto}(e) & \quad \Gamma + e : \ell_e \subseteq \ell \\
\Gamma, G + \Gamma \{ pc \} \Gamma & \\
C = \text{ifthenelse}(e, \text{goto}(e_1), \text{goto}(e_2)) & \quad \Gamma + e : \ell_e \subseteq \ell \\
\Gamma, G + \Gamma \{ pc \} \Gamma & \\
\end{align*}
\]

**Figure 10.** Complete list of type rules. \(C\) is a command from the CFG node pointed to by \(pc\) and callee and caller are callee- and caller-save registers.