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
hnnemati@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

*Work done while the author was at Microsoft Research, India.

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303952

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 $\mathcal{T}$. The remaining untrusted application, in contrast, is called $\mathcal{U}$. Code in $\mathcal{T}$ is not subject to any taint checks and can be compiled using a vanilla compiler. It can access all of $\mathcal{U}$'s memory, specifically, it can perform declassification by copying data from $\mathcal{U}$'s private region to $\mathcal{U}$'s public region (e.g. after encrypting the data). $\mathcal{T}$ has its own separate stack and heap, and we re-use the range checks on memory accesses in $\mathcal{U}$ to prevent them from reading or writing to $\mathcal{T}$'s memory. We give an example and general guidelines for refactoring $\mathcal{U}$ and $\mathcal{T}$ 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 $\mathcal{U}$, assuming that the $\mathcal{T}$ functions it calls are non-interfering. In other words, we prove that if two public-equivalent configurations of $\mathcal{U}$ 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 Evalution 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 implicit leaks too. We ran all our experiments (Section 7) in this stricter mode. Side channels (such as execution time and memory-access patterns) are outside the scope of this work.

Our scheme can also be used for integrity protection in a setting where an application computes over trusted and untrusted data [19]. Any data (explicitly) derived from untrusted inputs cannot be supplied to a sink that expects trusted data (Section 7.5 shows an example).

Example application. Consider the code for a web server in Figure 1. The server receives requests from the user (`main`:7), where the request contains the username and a file name (both in clear text), and the encrypted user password. The server decrypts the password and calls the `handleReq` helper routine that copies the (public) file contents into the `out` buffer. The server finally prepares the formatted response (`format`), and sends the response (`buf`), in clear, to the user.

The `handleReq` function allocates two local buffers, `passwd` and `fcontents` (`handleReq`:4). It reads the actual user password (e.g., from a database) into `passwd`, and authenticates the user. On successful authentication, it reads the file contents into `fcontents`, copies them to the `out` buffer, and appends a message to it signalling the completion of the request.

The code has several bugs that can cause it to leak the user password. First, at line 10, the code leaks the clear-text password to a log file by mistake. Note that memory-safety alone would not prevent this kind of bugs. Second, at line 14, `memcpy` reads `out_size` bytes from `fcontents` and copies them to `out`. If `out_size` is greater than `SIZE`, this can cause `passwd` to be copied to `out` because an overflow past `fcontents` would go into the `passwd` buffer. Third, if the format string `fmt` in the `sprintf` call (line 16) contains extra formatting directives, it can print stack contents into `out` ([56]). The situation is worse if `out_size` or `fmt` can be influenced by the attacker.

Our goal is to prevent such vulnerabilities from leaking out sensitive application data. Below we discuss the three main components of our approach.

Identifying trusted code. Figure 2 shows the workflow of our toolchain. The programmer starts by identifying code that must be *trusted*. This code, called $\mathcal{T}$ (for trusted), consists of functions that legitimately or intentionally declassify private data, or provide I/O. The remaining bulk of code, denoted $\mathcal{U}$, is *untrusted* and subjected to our compilation scheme. A good practice is to contain most of the application logic in $\mathcal{U}$ and limit $\mathcal{T}$ to a library of generic routines that can be hardened over time, possibly even formally verified [57]. For example, in the web server code from Figure 1, $\mathcal{T}$ would consist of: `recv`, `send`, `read_file` (network, I/O), `decrypt` (cryptographic primitive), and `read_passwd` (source of sensitive data). The remaining web server code (`parse`, `format`, and even `sprintf` and `memcpy`) would be in $\mathcal{U}$.

The programmer compiles $\mathcal{T}$ with any compiler (or even uses pre-compiled binaries), but $\mathcal{U}$ is compiled with our compiler ConfLLVM.

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 **`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 **`struct`** definitions. These latter annotations within $\mathcal{U}$ are *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:

```
int recv(int fd, char *buf, int buf_size);
int send(int fd, char *buf, int buf_size);
void decrypt(char *ciphertxt, private char *data);
void read_passwd(char *uname, private char *pass,
                 int size);
```

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

```
void handleReq(char *uname, private char *upasswd,
               char *fname, char *out, int out_sz);
int authenticate(char *uname, private char *upass,
                 private char *pass);
```

ConfLLVM automatically infers that, for example, `passwd` (line 4) is a **`private`** buffer. Based on this and `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 **`private 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 *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

```c
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");
}
```

```c
#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

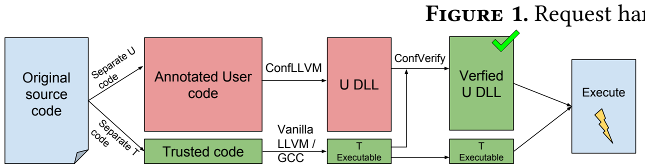


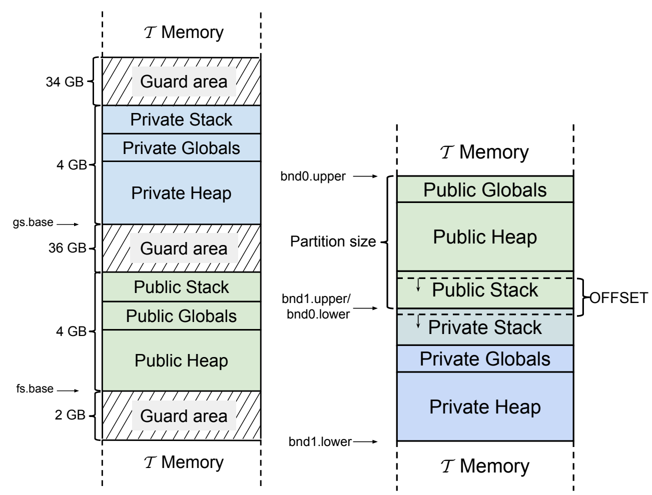
Figure 2. Workflow of our scheme and toolchain

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

Trusted Computing Base (TCB). We have also designed and implemented a static verifier, ConfVerify, to confirm that a binary output by ConfLLVM has enough checks in place to guarantee confidentiality (Section 5). 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 $\mathcal{U}$ or the compiler. We only trust the (small) library code $\mathcal{T}$ and the static verifier. We discuss more design considerations for $\mathcal{T}$ in Section 8.

3 Memory Partitioning Schemes

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 $\mathcal{U}$, 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, ConfLLVM proceeds to compile $\mathcal{U}$



(a) Segment scheme **(b)** MPX scheme

Figure 3. Memory layout of $\mathcal{U}$

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 (`bnd0` and `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 `OFFSET` at compile 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 `OFFSET` to each other. For each function call, the compiler arranges to generate a frame each on the public and the private stacks. Spilled **`private`** local variables and **`private`** arguments are stored on the private stack; 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. ConfLLVM automatically infers that `x` is a **`private int`** and places it on the private stack, whereas `y` is kept on the public stack. The stack pointer `rsp` points to the top of the public stack. Because the two stacks are kept at constant `OFFSET` to each other, `x` is accessed simply as `rsp+4+OFFSET`. The code also shows the instrumented MPX bound check instructions.

Segmentation scheme. x64 memory operands are in the form [*base* + *index* ∗ *scale* + *displacement*], where *base* and *index* are 64-bit unsigned registers, *scale* is a constant with maximum value of 8, and *displacement* is a 32-bit signed constant. The architecture also provides two segment registers `fs` and `gs` for the *base* address computation; conceptually, `fs`:*base* simply adds `fs` to *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 *base* is prefixed with `fs` cannot escape the public segment, and any memory operand prefixed with `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 *base* address stored in a 64-bit register, say a private value stored in `rax`, as `fs+eax`, where `eax` is the lower 32 bits of `rax`. Thus, in `fs+eax`, the lower 32 bits come from `eax` and the upper 32 bits come from `fs` (because `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 $\mathcal{U}$ 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 *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 $\mathcal{U}$ pointers when the control is passed to $\mathcal{T}$, thus avoiding the need to change or recompile $\mathcal{T}$. Generated code for our example under this scheme is shown in Figure 4b. The figure uses the convention that $\mathtt{e}_i$ (resp., `esp`) represents the lower 32 bits of the register $\mathtt{r}_i$ (resp., `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 `rsp+4+size`, where `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 (Section 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 registers. However, we do need additional support for thread-local storage (TLS). Typically, TLS is accessed via the segment register `gs`: the base of TLS is obtained at a constant offset from `gs`. The operating system takes care of setting `gs` on a per-thread basis. However, $\mathcal{U}$ and $\mathcal{T}$ operate in different trust domains, thus they cannot share the same TLS buffer.

We let $\mathcal{T}$ continue to use `gs` for accessing its own TLS. ConfLLVM changes the compilation of $\mathcal{U}$ to access TLS in a different way. The multiple (per-thread) stacks in $\mathcal{U}$ 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. $\mathcal{U}$ simply masks the lower 20-bits of `rsp` to zeros to obtain the base of the stack and access TLS.

The segment-register scheme further requires switching of the `gs` register as control transfers between $\mathcal{U}$ and $\mathcal{T}$. We use appropriate wrappers to achieve this switching, however $\mathcal{T}$ needs to reliably identify the current thread-id when called from $\mathcal{U}$ (so that $\mathcal{U}$ cannot force two different threads to use the same stack in $\mathcal{T}$). ConfLLVM achieves this by instrumenting an inlined-version of the `_chkstk` routine[1] to make sure that `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 $\mathcal{U}$ 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 canaries* 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) inputs [26, 35, 57]. While these techniques may prevent certain

[1] https://msdn.microsoft.com/en-us/library/ms648426.aspx

```
private int bar (private int *p, int *q)
{
  int x = *p;
  int y = *q;
  return x + y;
}
```

(a) A sample $\mathcal{U}$ procedure

```
;argument registers p = r1, p = r2
;private memory operands are gs prefixed
;public  memory operands are fs prefixed

sub rsp, 16            ;rsp = rsp - 16
r3 = load gs:[e1]      ;r3 = *p
store gs:[esp+4], r3   ;x = r3
r4 = load fs:[e2]      ;r4 = *q
store fs:[esp+8], r4   ;y = r4
r5 = load gs:[esp+4]   ;r5 = x
r6 = load fs:[esp+8]   ;r6 = y
r7 = r5 + r6
add rsp, 16            ;rsp = rsp + 16
ret r7
```

(b) Assembly code under segment scheme

```
;argument registers p = r1, q = r2
;stack offsets from rsp: x: 4, y: 8
sub rsp, 16                 ;rsp = rsp - 16
bndcu [r1], bnd1            ;MPX instructions to check that-
bndcl [r1], bnd1            ;-r1 points to private region
r3 = load [r1]              ;r3 = *p
bndcu [rsp+4+OFFSET], bnd1  ;check that rsp+4+OFFSET-
bndcl [rsp+4+OFFSET], bnd1  ;-points to private region
store [rsp+4+OFFSET], r3    ;x = r3
bndcu [r2], bnd0            ;check that r2 points to-
bndcl [r2], bnd0            ;-the public region
r4 = load [r2]              ;r4 = *q
bndcu [rsp+8], bnd0         ;check that rsp+8 points to-
bndcl [rsp+8], bnd0         ;-the public region
store [rsp+8], r4           ;y = r4
bndcu [rsp+4+OFFSET], bnd1
bndcl [rsp+4+OFFSET], bnd1
r5 = load [rsp+4+OFFSET]    ;r5 = x
bndcu [rsp+8], bnd0
bndcl [rsp+8], bnd0
r6 = load [rsp+8]           ;r6 = y
r7 = r5 + r6
add rsp, 16                 ;rsp = rsp + 16
ret r7
```

(c) Assembly code under MPX scheme

Figure 4. The (unoptimized) assembly generated by ConfLLVM for an example procedure.

attacks, our *only* goal is ensuring confidentiality. Thus, we designed a custom taint-aware CFI scheme.

Our CFI scheme ensures that for each indirect transfer of control: (a) the target address is *some* valid jump location, i.e., the target of an indirect call is some valid procedure entry, and the target of a return is some valid return site, and (b) 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.

CFI for function calls and returns. We use a *magic-sequence* based scheme to achieve this CFI. We follow the x64 calling convention for Windows that has 4 argument registers and one return register. Our scheme picks two bit sequences $\mathtt{M}_{\mathtt{Call}}$ and $\mathtt{M}_{\mathtt{Ret}}$ of length 59 bits each that appear nowhere else in $\mathcal{U}$'s binary. Each procedure in the binary is preceded with a string that consists of $\mathtt{M}_{\mathtt{Call}}$ followed by a 5-bit sequence encoding the expected taints of the 4 argument registers and the return register, as per the function signature. Similarly, each valid return site in the binary is preceded by $\mathtt{M}_{\mathtt{Ret}}$ followed by a 1-bit encoding of the taint of the return value register, again according to the callee's signature. To keep the length of the sequences uniform at 64 bits, the return site taint is padded with four zero bits. The 64-bit instrumented sequences are collectively referred to as magic sequences.

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 $\mathcal{U}$:

```
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:

```
#M_call#11111#
add:
```

```
  ...  ;assembly code for add
#M_call#01111#
incr:
  ... ;assembly code of incr
  call add
  #M_ret#00001# ;private-tainted ret with padded 0s
  ... ;assembly code for rest of incr
```

Our CFI scheme adds runtime checks using these sequences as follows. Each `ret` is replaced with instructions to: (a) fetch the return address, (b) confirm that the target location has $\mathsf{M}_{\mathsf{Ret}}$ followed by the taint-bit of the return register, and if so, (c) jump to the target location. For our example, the `ret` inside `add` is replaced as follows:

```
#M_call#11111#
add:
  ...
  r1 = pop ;fetch return address
  r2 = #M_ret_inverted#11110# ;bitwise negation
  r2 = not r2                 ;of M_ret
  cmp [r1], r2
  jne fail
  r1 = add r1, 8 ;skip magic sequence
  jmp r1 ;return
fail: call __debugbreak
```

We use the bitwise negation of $\mathsf{M}_{\mathsf{Ret}}$ in the code to maintain the invariant that the magic sequence does not appear in the binary at any place other than valid return sites. There is no requirement that the negated sequence not appear elsewhere in the binary.

For direct calls, ConfLLVM statically verifies that the register taints match between the call site and the call target. At indirect calls, the instrumentation is similar to that of a `ret`: check that the target location contains $\mathsf{M}_{\mathsf{Call}}$ followed by taint bits that match the register taints at the call site.

Indirect jumps. ConfLLVM does not generate indirect (non-call) jumps in $\mathcal{U}$. Indirect jumps are mostly required for jump-table optimizations, which we currently disable. We can conceptually support them as long as the jump tables are statically known and placed in read-only memory.

The insertion of magic sequences increases code size but it makes the CFI-checking more lightweight than the shadow stack schemes. The unique sequences $\mathsf{M}_{\mathsf{Call}}$ and $\mathsf{M}_{\mathsf{Ret}}$ are created post linking when the binaries are available (Section 6).

5 Implementation

We implemented ConfLLVM as part of the LLVM framework [36], targeting Windows and Linux x64 platforms. It is possible to implement all of ConfLLVM's instrumentation using simple branching instructions available on all platforms, 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 corresponding **`struct`**-typed 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 declared as `private st x;`, then `x.p` is a private pointer pointing to a private integer. We follow the same convention with **`union`**s as well: all fields of a union inherit their outermost annotation from the **`union`**-typed variable.

This convention ensures that despite the memory partitioning into public and private regions, each object is laid out contiguously 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 Representation (IR) instrumented with this additional metadata. Once the IR is generated, ConfLLVM runs standard LLVM IR optimizations that are part of the LLVM toolchain. Most of the optimizations work as-is and don't require any change. Optimizations 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 over the IR. This inference pass propagates the type qualifier 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 standard 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 determines the taint of each register at each instruction.

Register spilling and code generation. We made the register 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^{20}. 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 `rsp+OFFSET` 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 $\mathcal{U}$ is tainted **`private`**. In such a case, the job of the compiler is easy: it only needs to limit memory accesses in $\mathcal{U}$ 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 callsite (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 $\mathcal{U}$ using ConfLLVM. The only external functions for $\mathcal{U}$'s code are those exported by $\mathcal{T}$. $\mathcal{U}$'s code is compiled with an (auto-generated) stub file, that implements each of these $\mathcal{T}$ functions as an indirect jump from a table `externals`, located at a constant position in $\mathcal{U}$ (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 $\mathcal{U}$ files to produce a $\mathcal{U}$ dll.

The $\mathcal{U}$ 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 $\mathcal{T}$ functions. For each of the functions in $\mathcal{T}$'s interface exported to $\mathcal{U}$, 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 $\mathcal{U}$'s public region), (b) copies arguments to $\mathcal{T}$'s stack, (c) switches `gs`, (d) switches `rsp` to $\mathcal{T}$'s stack, and (e) calls the underlying $\mathcal{T}$ function (e.g. `send` in libc). On return, it the wrapper switches `gs` and `rsp` back and jumps to $\mathcal{U}$ in a similar manner to our CFI return instrumentation. Additionally, the wrappers include the magic sequences similar to those in $\mathcal{U}$ so that the CFI checks in $\mathcal{U}$ do not fail when calling $\mathcal{T}$. These wrappers are compiled with the $\mathcal{T}$ dll, and the output dll exports the interface functions.

Loading the $\mathcal{U}$ and $\mathcal{T}$ dlls. When loading the $\mathcal{U}$ and $\mathcal{T}$ dlls, the loader: (1) populates the `externals` table in U with addresses of the wrapper functions in $\mathcal{T}$, (2) relocates the globals in $\mathcal{U}$ 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 $\mathcal{U}$) and compile it with ConfLLVM. We use the system's native libc, which is treated as trusted (in $\mathcal{T}$). 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 $\mathcal{T}$ 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.[2]
- **Base**$_{\text{OA}}$: Benchmarks compiled with *vanilla* LLVM but running with our custom allocator.
- **Our**$_{\text{Bare}}$: Compiled with ConfLLVM, but without any runtime instrumentation. However, all optimizations unsupported by ConfLLVM are disabled, the memories of $\mathcal{T}$ and $\mathcal{U}$ are separated and, hence, stacks are switched in calling $\mathcal{T}$ functions from $\mathcal{U}$.
- **Our**$_{\text{CFI}}$: Like **Our**$_{\text{Bare}}$ but additionally with CFI instrumentation, but no memory bounds enforcement.
- **Our**$_{\text{MPX}}$: Full ConfLLVM, memory-bounds checks use MPX.
- **Our**$_{\text{Seg}}$: Full ConfLLVM, memory-bounds checks use segmentation.

Briefly, the difference between **Our**$_{\text{CFI}}$ and **Our**$_{\text{Bare}}$ is the cost of our CFI instrumentation. The difference between **Our**$_{\text{MPX}}$ (resp. **Our**$_{\text{Seg}}$) and **Our**$_{\text{CFI}}$ 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 (**Our**$_{\text{MPX}}$) is up to 74.03%, while that of ConfLLVM using segmentation (**Our**$_{\text{Seg}}$) 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 (**Our**$_{\text{CFI}}$−**Our**$_{\text{Bare}}$). The

[2] O2 is the standard optimization level for performance evaluation. Higher levels include "optimizations" that don't always speed up the program.

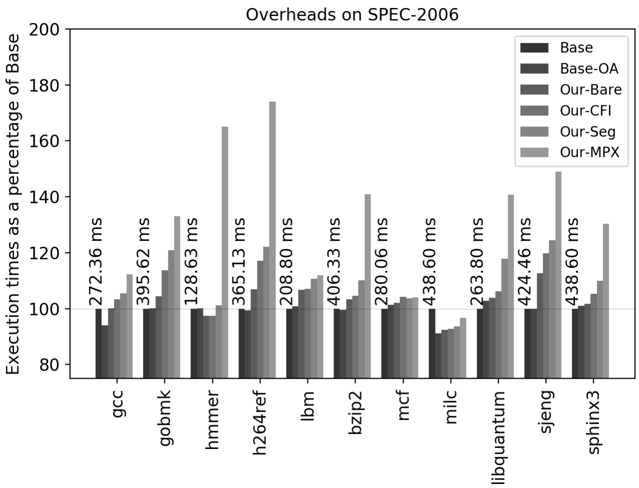


Figure 5. Execution time as a percentage of **Base** for SPEC CPU 2006 benchmarks. Numbers above the **Base** bars are absolute execution times of the baseline in ms. All bars are averages of 10 runs. Standard deviations are all below 3%.

average CFI overhead is 3.62%, competitive with best known techniques [26]. Stack switching and disabled optimizations (**Our**$_{\text{Bare}}$) account for the remaining overhead. The overhead due to our custom memory allocator (**Base**$_{\text{OA}}$) 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 (**Our**$_{\text{CFI}}$, 4.17%) seems to be higher than that of the full MPX-based instrumentation (**Our**$_{\text{MPX}}$, 4.02%). We verified that this is due to an outlier in the **Our**$_{\text{CFI}}$ experiment. On hmmer, the overhead of **Our**$_{\text{Bare}}$ is negative because the optimizations that ConfLLVM disables actually slow it down. Finally, on milc, the overhead of ConfLLVM is negative because this benchmark benefits significantly from the use of our custom memory allocator. Indeed, relative to **Base**$_{\text{OA}}$, 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 $\mathcal{T}$, and the rest of NGINX, including all its request parsing, processing, serving, and logging code in $\mathcal{U}$. The code in $\mathcal{U}$ is compiled with ConfLLVM (total 124,001 LoC). Within $\mathcal{U}$, 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 $\mathcal{T}$ that $\mathcal{U}$ invokes to encrypt the request URI before adding it to the log. This function encrypts using a key that is isolated in $\mathcal{T}$'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. The standard `SSL_recv` function in $\mathcal{T}$ decrypts the incoming payload with the session key, and provides it to $\mathcal{U}$ in a `private` buffer.

```
size_t SSL_recv(SSL *connection,
          private void *buffer, size_t length);
```

In total, we added or modified 160 LoC in NGINX (0.13% of NGINX's codebase) and added 138 LoC to $\mathcal{T}$.

Our goal is to measure the overhead of ConfLLVM on the maximum sustained throughput of NGINX. We run our experiments in 6 configurations: **Base**, **Our**$_{\text{1Mem}}$, **Our**$_{\text{Bare}}$, **Our**$_{\text{CFI}}$, and **Our**$_{\text{MPX-Sep}}$, **Our**$_{\text{MPX}}$. Of these, **Base**, **Our**$_{\text{Bare}}$, **Our**$_{\text{CFI}}$, and **Our**$_{\text{MPX}}$ are the same as those in Section 7.1. We use MPX for bounds checks (**Our**$_{\text{MPX}}$), as opposed to segmentation, since we know from Section 7.1 that MPX is worse for ConfLLVM. **Our**$_{\text{1Mem}}$ is like **Our**$_{\text{Bare}}$ (compiled with ConfLLVM without any instrumentation), but also does not separate memories for $\mathcal{T}$ and $\mathcal{U}$. **Our**$_{\text{MPX-Sep}}$ includes all instrumentation, but does not separate stacks for private and public data. Briefly, the difference between **Our**$_{\text{Bare}}$ and **Our**$_{\text{1Mem}}$ is the overhead of separating $\mathcal{T}$'s memory from $\mathcal{U}$'s and switching stacks on every call to $\mathcal{T}$, while the difference between **Our**$_{\text{MPX}}$ and **Our**$_{\text{MPX-Sep}}$ 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,

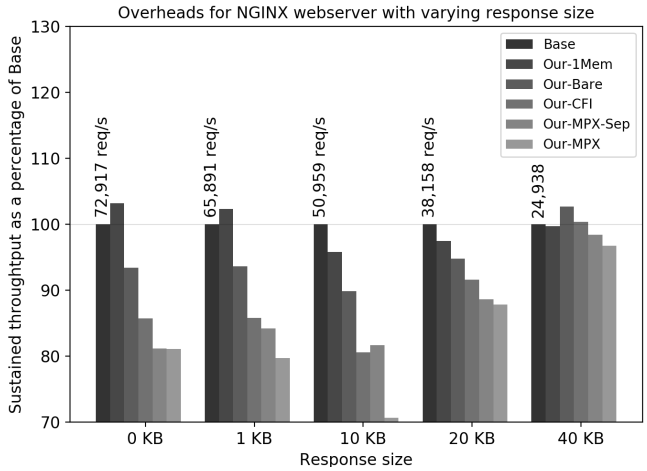


FIGURE 6. Maximum sustained throughput as a percentage of **Base** for the NGINX web server with increasing response sizes. Numbers above the **Base** bars are absolute throughputs in `req/s` for the baseline. All bars are averages of 10 runs. Standard deviations are all below 0.3%, except in **Base** for response sizes 0 KB and 1 KB, where they are below 2.2%.

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 $\textbf{Our}_{\text{MPX}}-\textbf{Our}_{\text{MPX-Sep}}$). This is unsurprising: As the file size increases, so does the cache footprint of $\mathcal{U}$. In contrast, the overheads due to CFI (difference $\textbf{Our}_{\text{CFI}}-\textbf{Our}_{\text{Bare}}$) and the separation of the memories of $\mathcal{T}$ and $\mathcal{U}$ (difference $\textbf{Our}_{\text{Bare}}-\textbf{Our}_{\text{1Mem}}$) 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 ($\textbf{Our}_{\text{MPX}}$). 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 $\textbf{Our}_{\text{MPX}}$, 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**$_{\mathrm{OA}}$, **Our**$_{\mathrm{Bare}}$, **Our**$_{\mathrm{CFI}}$ and **Our**$_{\mathrm{MPX}}$. 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**$_{\mathrm{MPX}}$ 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

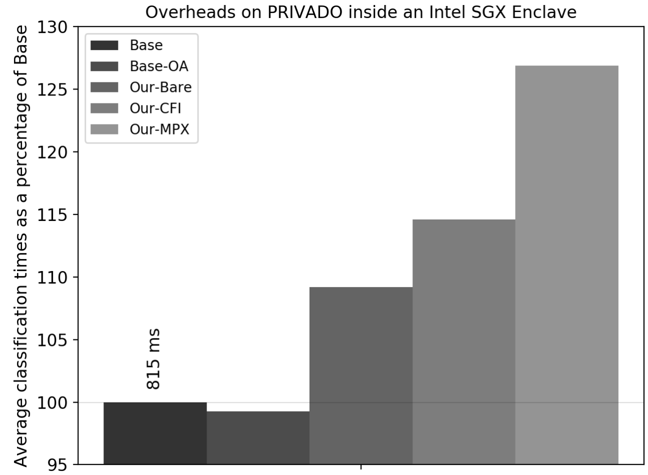


Figure 7. Average classification time for Privado inside an Intel SGX Enclave as a percentage of **Base**. The number above the **Base** bar is the absolute execution time of the baseline in `ms`. Every bar is the average of 10,000 trials. Standard deviations are all below 1%.

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 hyperthreaded 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**$_{\mathrm{Seg}}$ and **Our**$_{\mathrm{MPX}}$. 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**$_{\mathrm{Seg}}$ is below 10% and that of **Our**$_{\mathrm{MPX}}$ 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

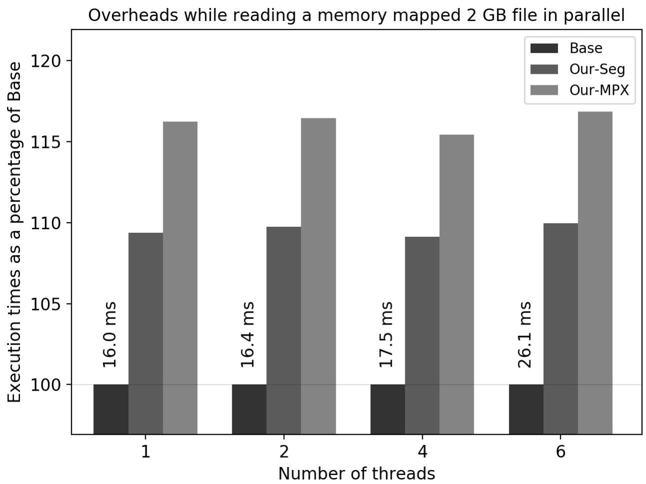


Figure 8. Total execution time as a percentage of **Base** for reading a memory-mapped 2 GB file in parallel, as a function of the number of reading threads. Numbers above the **Base** bars are absolute execution times of the baseline in `ms`. Each bar is an average of 5 runs. Standard deviations are all below 3%.

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 $\mathcal{U}$: 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, $\mathcal{T}$ is trusted and therefore must be implemented with care. ConfLLVM guarantees that $\mathcal{U}$ cannot access $\mathcal{T}$'s memory or jump to arbitrary points in $\mathcal{T}$, so the only remaining attack surface for $\mathcal{T}$ is the API that it exposes to $\mathcal{U}$. $\mathcal{T}$ must ensure that stringing together a sequence of these API calls cannot cause leaks. We recommend the following (standard) strategies. First, $\mathcal{T}$ 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 $\mathcal{U}$. Such a $\mathcal{T}$ can be re-used across applications and can be subject to careful audit/verification. Further, declassification routines in $\mathcal{T}$ must provide guarded access to $\mathcal{U}$. For instance, $\mathcal{T}$ 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 $\mathcal{T}$'s binary to prevent $\mathcal{U}$ from jumping inside $\mathcal{T}$. We ensure this by selecting the magic string when the entire code of $\mathcal{U}$ and $\mathcal{T}$ is available. While dynamic loading in $\mathcal{U}$ can simply be disallowed, any dynamic loading in $\mathcal{T}$ 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 $\mathcal{U}$ to remain inside $\mathcal{U}$'s own code.

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

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 $\mathcal{U}$ and $\mathcal{L}$. $\mathcal{U}$'s code is compiled via a special instrumenting compiler [23] while $\mathcal{L}$'s code is trusted and may be compiled using any compiler. This is similar in principle to our $\mathcal{U}$-$\mathcal{T}$ 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 $\mathcal{U}$ goes through $\mathcal{L}$, 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 $\mathcal{L}$ from $\mathcal{U}$). 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

[1] 2011 CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[2] Chrome owned by exploits in hacker contests, but Google's $1M purse still safe. https://www.wired.com/2012/03/pwnium-and-pwn2own/.

[3] Clang: A C language family frontend for LLVM. http://clang.llvm.org.

[4] Cve-2012-0769, the case of the perfect info leak. http://zhodiac.hispahack.com/my-stuff/security/Flash_ASLR_bypass.pdf.

[5] dlmalloc: A Memory Allocator. http://g.oswego.edu/dl/html/malloc.html.

[6] The heartbleed bug. http://heartbleed.com/.

[7] Intel Memory Protection Extensions (Intel MPX). https://software.intel.com/en-us/isa-extensions/intel-mpx.

[8] Minizip. https://github.com/nmoinvaz/minizip.

[9] Mongoose. https://github.com/cesanta/mongoose.

[10] NGINX web server. https://www.nginx.com/.

[11] Smashing the stack for fun and profit. insecure.org/stf/smashstack.html.

[12] SPEC CPU 2006. https://www.spec.org/cpu2006/.

[13] Torch TH library. https://github.com/torch/TH.

[14] Torch THNN library. https://github.com/torch/nn/tree/master/lib/THNN.

[15] wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2.

[16] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[17] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[18] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[19] Ken Biba. Integrity considerations for secure computer systems. Technical report, 1977.

[20] Priyam Biswas, Alessandro Di Federico, Scott A. Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. Venerable variadic vulnerabilities vanquished. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.

[21] Fraser Brown, Andres Nötzli, and Dawson Engler. How to build static checking systems using orders of magnitude less code. *SIGPLAN Not.*, 51(4):143–157, March 2016.

[22] Scott A. Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia*

Conference on Computer and Communications Security, ASIA CCS '17, pages 193–204, New York, NY, USA, 2017. ACM.

[23] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Symposium on Operating Systems Principles (SOSP)*, 2009.

[24] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, pages 520–535, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[26] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 555–566, New York, NY, USA, 2015. ACM.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[29] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A Theory of Type Qualifiers. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 192–203, Atlanta, Georgia, May 1999.

[30] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. ACM.

[31] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 517–528, New York, NY, USA, 2016. ACM.

[32] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM.

[33] A. Henderson, L. K. Yan, X. Hu, A. Prakash, H. Yin, and S. McCamant. Decaf: A platform-neutral whole-system dynamic binary analysis platform. *IEEE Transactions on Software Engineering*, 43(2):164–184, Feb 2017.

[34] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.

[35] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[36] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[37] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 321–334, 2009.

[38] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[39] Jiang Ming, Dinghao Wu, Jun Wang, Gaoyao Xiao, and Peng Liu. Straighttaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 308–319, New York, NY, USA, 2016. ACM.

[40] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[41] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 245–258, 2009.

[42] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[43] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[44] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, 2005.

[45] OpenLDAP Project. Openldap. http://www.openldap.org/.

[46] François Pottier and Vincent Simonet. Information flow inference for ML. In *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[47] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017.

[48] B. P. S. Rocha, M. Conti, S. Etalle, and B. Crispo. Hybrid static-runtime information flow and declassification enforcement. *IEEE Transactions on Information Forensics and Security*, 8(8):1294–1305, Aug 2013.

[49] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[50] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[51] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 15–30, March 2016.

[52] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 28–28, Berkeley, CA, USA, 2012. USENIX Association.

[53] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511, June 2006.

[54] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[55] Zhiyong Shan. Suspicious-taint-based access control for protecting OS from network attacks. *CoRR*, abs/1609.00100, 2016.

[56] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, SSYM'01, Berkeley, CA, USA, 2001. USENIX Association.

[57] Rohit Sinha, Manuel Costa, Akash Lal, Nuno P. Lopes, Sriram K. Rajamani, Sanjit A. Seshia, and Kapil Vaswani. A design and verification

Cmd	$:=$	$\mathbf{ldr}\,(reg, exp)$ | $\mathbf{str}\,(reg, exp)$ | $\mathbf{goto}\,(exp)$ | $\mathbf{ifthenelse}\,(exp, \mathbf{goto}\,(exp), \mathbf{goto}\,(exp))$ | $\mathbf{ret}$ | $\mathbf{call}_{\{\mathcal{U}\|\mathcal{T}\}}\ f\,(exp^*)$ | $\mathbf{icall}\ exp\,(exp^*)$ | $\mathbf{assert}\,(exp^*)$
exp	$:=$	$n \in Val$ | $reg \in Reg$ | $\Diamond_u\ exp$ | $exp\ \Diamond_b\ exp$ | $\&f$

Table 1. Command syntax

methodology for secure isolated regions. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 665–681, 2016.

[58] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, USA, 2004. ACM.

[59] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, February 1997.

[60] Shruti Tople, Karan Grover, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and secure DNN inference. *CoRR*, abs/1810.00602, 2018.

[61] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 29–40. ACM, 2011.

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 $\mathcal{T}$ and $\mathcal{U}$, 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 $G_f = \langle V_f, E\rangle$ of nodes of f and edges $E \subseteq V \times V$. The edges represent all direct control transfers. A node $\langle pc, C, \Gamma, \Gamma'\rangle \in V_f$ consists of a program counter pc, a single command (assembly instruction) C, and register taints Γ and Γ' before and after the execution of C. pc is a number, like a line number, that can be used in an indirect jump to this node. Γ and Γ' are maps from machine register ids to $\{\mathbf{H}, \mathbf{L}\}$, representing high (private) and low (public) data.

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 $\Diamond_b$ and $\Diamond_u$, respectively), and the special operation $\&f$ which returns the 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 $\mathbf{ldr}\,(reg, exp)$ and store $\mathbf{str}\,(reg, exp)$ where the input exp evaluates to a memory location, unconditional jumps $\mathbf{goto}\,(exp)$, conditionals **ifthenelse**, direct function calls $\mathbf{call}_{\{\mathcal{U}|\mathcal{T}\}}\ f\,(exp^*)$, indirect function calls **icall**, and return **ret**. The subscript for $\mathbf{call}_{\{\mathcal{U}|\mathcal{T}\}}$ 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() 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 pc which points to a node in the program's CFG. The memory μ is actually a union of two maps μ_{L} and μ_{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 $\langle \mu, \rho, [\sigma_{\mathrm{H}} : \sigma_{\mathrm{L}}], pc\rangle$ of the entire state—the memory, register state, stack, and the program counter. We make a distinction between high (σ_{H}) and low (σ_{L}) stacks. We write $\langle . \rangle.i$ to project a tuple to its ith component. Thus, if $s = \langle \mu, \rho, [\sigma_{\mathrm{H}} : \sigma_{\mathrm{L}}], pc\rangle$, then $s.pc = pc$. We call a configuration s initial (final) if $s.pc$ 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 $e \Downarrow n$ to mean evaluation of an expression to its corresponding (concrete) value. Moreover, we extend the configuration s of G with an additional component ν that represents the memory fragment available only to the trusted code and its protected against untrusted accesses, i.e., $\langle \nu, \mu, \rho, [\sigma_{\mathrm{H}} : \sigma_{\mathrm{L}}], pc\rangle$. Further, we introduce the record $\mathcal{F} = \{f_i \mapsto \langle n_i, M\_call_i\rangle \mid f_i \in \mathcal{P}\}$ and extend the typing judgment with this record $\mathcal{F}, G \vdash \Gamma\{pc\}\Gamma'$. $\mathcal{F}$ 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 $s \hookrightarrow_f s'$ that models the entire execution of the trusted function f and its effects on the configuration. The special transition $s \longrightarrow \bot$ means that $s.pc$ contains an **assert** command, one of whose arguments evaluates to false. $\bot$ does not transition any further. Furthermore, the transition $s \longrightarrow \lightning$ is used to model adversarial behavior, for example, when the target of **icall** is not in the CFG. $\lightning$ resembles a configuration where memory locations are loaded with random values.

Security Analysis. We formalize the checks that ConfVerify 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 Γ and Γ', 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 $pred(G, pc)$ returns the predecessors of the node labeled pc in G. For $\ell \in \{\mathbf{L}, \mathbf{H}\}$, $\mathbf{assert}(e \in Dom(\mu_\ell))$ represents the dynamic check that e evaluates to an address in the domain of μ_ℓ, and the auxiliary judgment $\Gamma \vdash e : \ell$ means that the expression e depends only on values with secrecy ℓ or lower.

The rules are mostly self-explanatory. The first rule, which is for the command $\mathbf{ldr}(reg, e)$, says that if reg has taint ℓ_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 μ_{ℓ_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 *Cons* be the bit-concatenation function, then: (i) for the function entry node v, $M\_call = \underset{i=0\ldots3}{Cons}(v.\Gamma(reg_i))$, and (ii) for a given return address adr and the node v such that $v \in pred(G, adr)$, $M\_ret = v.\Gamma'(reg_0)$. 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 ConfVerify 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 $\mathbf{assert}(\ell_{[1-4]} \sqsubseteq M\_call \ \mathbf{and}\ e_f \in G \ \mathbf{and}\ (e_f \mapsto M\_call) \in \mathcal{F})$ 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 $\mathcal{F}$. 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, ConfVerify confirms that the return address on the stack has a magic signature that has a taint bit of inferred return type ℓ for the function. In this rule we use the function *top* with the standard meaning to manipulate the stack content. Again since $\mathbf{assert}(\forall v' \in pred(G, top(\sigma_\mathbf{L})).\ \ell \sqsubseteq v'.\Gamma'(reg_0))$ 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 v in (untrusted functions in) G: 1) The node (locally) satisfies the type system. Formally, $G \vdash v.\Gamma\ \{v.pc\}\ v.\Gamma'$, and 2) The ending taint of the node is consistent with the beginning taints of all its successors. Formally, for all nodes $v' \in succ(G, v.pc)$, $v.\Gamma' \sqsubseteq v'.\Gamma$, where $succ(G, pc)$ 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 noninterference*. Roughly, this property requires a notion of low equivalence of program configurations (of the same CFG G), written $s =_\mathbf{L} s'$, which allows memories of s and s' to differ only in the private region. A program is noninterfering if it *preserves* $=_\mathbf{L}$ 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 $s =_\mathbf{L} s'$ 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, $s.\sigma_\mathbf{L} = s'.\sigma_\mathbf{L}$, (iii) for all low memory addresses $m \in \mu_\mathbf{L}$, $s.\mu(m) = s'.\mu(m)$, (iv) for all registers r such that $G(s.pc).\Gamma(r) = \mathbf{L}$, $s.\rho(r) = s'.\rho(r)$.

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* $s_0 =_\mathbf{L} s_1$, *if* $s_0 \hookrightarrow_f s_0'$ *then* $\exists s_1'.\ s_1 \hookrightarrow_f s_1'$ *and* $s_0' =_\mathbf{L} s_1'$.

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* $s \not\rightarrow^* \lightning$.

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* $s_0 =_\mathrm{L} s_1$ *and* $s_0 \longrightarrow^* s_0'$*, then either* $s_1 \longrightarrow^* \bot$ *or* $\exists s_1'.\ s_1 \longrightarrow^* s_1'$ *and* $s_0' =_\mathrm{L} s_1'$.

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).

$$\frac{C = \mathbf{ldr}(reg,\, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \in (Dom(\mu_\mathrm{L}) \cup Dom(\mu_\mathrm{H}))}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho[reg \mapsto \mu(v)], [\sigma], pc+1\rangle}$$

$$\frac{C = \mathbf{ldr}(reg,\, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \notin (Dom(\mu_\mathrm{L}) \cup Dom(\mu_\mathrm{H}))}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \lightning}$$

$$\frac{C = \mathbf{str}(reg,\, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \in (Dom(\mu_\mathrm{L}) \cup Dom(\mu_\mathrm{H}))}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu[v \mapsto \rho(reg)], \rho, [\sigma], pc+1\rangle}$$

$$\frac{C = \mathbf{str}(reg,\, e) \quad \mu, \rho \vdash e \Downarrow n \quad n \notin (Dom(\mu_\mathrm{L}) \cup Dom(\mu_\mathrm{H}))}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \lightning}$$

$$\frac{C = \mathbf{call}_\mathcal{U}\ f_u\,(e_1, \ldots, e_4) \quad (f_u \mapsto \langle pc_f, -\rangle) \in \mathcal{F}}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho, [(pc+1); \sigma_\mathrm{L} : \sigma_\mathrm{H}], pc_f\rangle}$$

$$\frac{C = \mathbf{call}_\mathcal{T}\ f_\tau\,(e_1, \ldots, e_4) \quad (f_\tau \mapsto \langle pc_f, -\rangle) \in \mathcal{F}}{\langle v, \mu, \rho, [\sigma], pc\rangle \hookrightarrow_{f_\tau} \langle v', \mu', \rho', [(pc+1); \sigma_\mathrm{L}' : \sigma_\mathrm{H}'], pc_f\rangle}$$

$$\frac{C = \mathbf{icall}\ e_f\,(e_1, \ldots, e_4) \quad \mu, \rho \vdash e_f \Downarrow pc_f \quad pc_f \in G}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho, [(pc+1); \sigma_\mathrm{L} : \sigma_\mathrm{H}], pc_f\rangle}$$

$$\frac{C = \mathbf{icall}\ e_f\,(e_1, \ldots, e_4) \quad \mu, \rho \vdash e_f \Downarrow pc_f \quad pc_f \notin G}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \lightning}$$

$$\frac{C = \mathbf{ret} \quad adr \in G}{\langle v, \mu, \rho, [adr; \sigma_\mathrm{L} : \sigma_\mathrm{H}], pc\rangle \rightarrow \langle v, \mu, \rho, [\sigma_\mathrm{L} : \sigma_\mathrm{H}], adr]\rangle}$$

$$\frac{C = \mathbf{ret} \quad adr \notin G}{\langle v, \mu, \rho, [adr; \sigma_\mathrm{L} : \sigma_\mathrm{H}], pc\rangle \rightarrow \lightning}$$

$$\frac{C = \mathbf{goto}\,(e) \quad \mu, \rho \vdash e \Downarrow pc}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho, [\sigma], pc]\rangle}$$

$$\frac{C = \mathbf{ifthenelse}\,(e, \mathbf{goto}\,(e_1), \mathbf{goto}\,(e_2)) \quad \mu, \rho \vdash e \Downarrow T \quad \mu, \rho \vdash e_1 \Downarrow n_1}{\langle v, \mu,, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho, [\sigma], n_1\rangle}$$

$$\frac{C = \mathbf{ifthenelse}\,(e, \mathbf{goto}\,(e_1), \mathbf{goto}\,(e_2)) \quad \mu, \rho \vdash e \Downarrow F \quad \mu, \rho \vdash e_2 \Downarrow n_2}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho, [\sigma], n_2\rangle}$$

$$\frac{C = \mathbf{assert}(e) \quad \mu, \rho \vdash e \Downarrow T}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \langle v, \mu, \rho, [\sigma], pc+1\rangle}$$

$$\frac{C = \mathbf{assert}(e) \quad \mu, \rho \vdash e \Downarrow F}{\langle v, \mu, \rho, [\sigma], pc\rangle \rightarrow \bot}$$

Figure 9. Operational semantics rules.

$$\frac{C = \mathbf{ldr}(reg,\, e) \qquad \forall\, v \in pred(G,\, pc).\ v.C = \mathbf{assert}(e \in Dom(\mu_{\ell_e}))}{\mathcal{F},\, G \vdash \Gamma\,\{pc\}\,\Gamma[reg \mapsto \ell_e]}$$

$$\frac{C = \mathbf{str}(reg,\, e) \quad \Gamma \vdash reg : \ell_r \quad \ell_r \sqsubseteq \ell_e \qquad \forall v \in pred(G,\, pc).\ v.C = \mathbf{assert}(e \in Dom(\mu_{\ell_e}))}{\mathcal{F},\, G \vdash \Gamma\,\{pc\}\,\Gamma}$$

$$\frac{C = \mathbf{call}_{\{\mathcal{U}|\mathcal{T}\}}\ f\,(e_1,\, \ldots,\, e_4) \quad (f \mapsto \langle -,\, M\_call\rangle) \in \mathcal{F} \qquad \Gamma \vdash e_i : \ell_i \quad \text{for} \quad i = 1 \ldots 4 \qquad \ell_{[1-4]} \sqsubseteq M\_call}{\mathcal{F},\, G \vdash \Gamma\,\{pc\}\,\Gamma[rcaller \mapsto \mathbf{H},\, rcallee \mapsto \mathbf{L}]}$$

$$\frac{C = \mathbf{icall}\ e_f(e_1, \ldots, e_4) \quad \Gamma \vdash e_f : \ell_f \sqsubseteq \mathbf{L} \quad \Gamma \vdash e_i : \ell_i \ \mathbf{for}\ i = 1 \ldots 4 \qquad \forall v \in pred(G,\, pc).\ v.C = \mathbf{assert}\begin{pmatrix} \ell_{[1-4]} \sqsubseteq M\_call & \mathbf{and} \\ e_f \in G & \mathbf{and} \\ (e_f \mapsto \langle -,\, M\_call\rangle) \in \mathcal{F} & \end{pmatrix}}{\mathcal{F},\, G \vdash \Gamma\,\{pc\}\,\Gamma[rcaller \mapsto \mathbf{H},\, rcallee \mapsto \mathbf{L}]}$$

$$\frac{C = \mathbf{ret} \quad \Gamma \vdash rcallee \sqsubseteq \mathbf{L} \quad \Gamma \vdash reg_0 : \ell \qquad \forall v \in pred(G,\, pc).\ v.C = \mathbf{assert}(\forall v' \in pred(G,\, top(\sigma_{\mathrm{L}})).\ \ell \sqsubseteq v'.\Gamma'(reg_0))}{\mathcal{F},\, G \vdash \Gamma\ \{pc\}\ \Gamma}$$

$$\frac{C = \mathbf{goto}\,(e) \quad \Gamma \vdash e : \ell_e \sqsubseteq \mathbf{L}}{\mathcal{F},\, G \vdash \Gamma\ \{pc\}\ \Gamma}$$

$$\frac{C = \mathbf{ifthenelse}\,(e,\, \mathbf{goto}\,(e_1),\, \mathbf{goto}\,(e_2)) \quad \Gamma \vdash e : \ell_e \sqsubseteq \mathbf{L}}{\mathcal{F},\, G \vdash \Gamma\ \{pc\}\ \Gamma}$$

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