Sound Modular Verification of C Code Executing in an Unverified Context

Pieter Agten Bart Jacobs Frank Piessens

iMinds-DistriNet, KU Leuven

{firstname}.{lastname}@cs.kuleuven.be

Abstract

Over the past decade, great progress has been made in the static modular verification of C code by means of separation logic-based program logics. However, the runtime guarantees offered by such verification are relatively limited when the verified modules are part of a whole program that also contains unverified modules. In particular, a memory safety error in an unverified module can corrupt the runtime state, leading to assertion failures or invalid memory accesses in the verified modules. This paper develops runtime checks to be inserted at the boundary between the verified and the unverified part of a program, to guarantee that no assertion failures or invalid memory accesses can occur at runtime in any verified module. One of the key challenges is enforcing the separation logic frame rule, which we achieve by checking the integrity of the footprint of the verified part of the program on each control flow transition from the unverified to the verified part. This in turn requires the presence of some support for module-private memory at runtime. We formalize our approach and prove soundness. We implement the necessary runtime checks by means of a program transformation that translates C code with separation logic annotations into plain C, and that relies on a protected module architecture for providing module-private memory and restricted module entry points. Benchmarks show the performance impact of this transformation depends on the choice of boundary between the verified and unverified parts of the program, but is below 4% for real-world applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords separation logic; verification; runtime checking

1. Introduction

The construction of reliable software in unsafe languages like C or C++ is known to be challenging. Yet, because of the excellent performance of these languages, and because they can give the programmer access to low-level details of the machine on which the program is executing, they are often the languages of choice for infrastructural software such as hypervisors, operating systems and servers, and for embedded software.

POPL '15, January 15–17 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676972

One way of significantly increasing assurance in the reliability of software is the use of program verification. Over the past decade, program verification technology for C-like programming languages has reached a level of maturity that makes it possible to verify real-world software, albeit with considerable effort. Two high-profile examples are the verification of Microsoft's hypervisor Hyper-V using the VCC verifier [11], and the verification of seL4 [24], a microkernel of the L4 family. But there are many other examples of projects that formally verify the safety and/or functional correctness of (parts of) C programs.

For large systems, it is essential that the verification technique used is *modular*. Each module (for instance each function) is verified to comply with its specification, relying only on the *specification* of the other modules that the verified module is interacting with. Several sound modular verification tools for C have been proposed [10, 22]. However, the soundness properties of these verifiers have an important limitation. To the best of our knowledge, all soundness results for modular C verifiers have the form: under the condition that *all* modules of a program have been verified, any execution of that program will comply with the specification. In other words, as soon as there is one unverified module, all bets are off. The implementations of modules that are not verified are part of the *trusted computing base*; it is *assumed* that they comply with the specifications for these modules that were used to verify the verified part of the program. Such assumptions are particularly troublesome for memory-unsafe languages such as C, as a single memory-safety error (such as a buffer overflow) in one unverified module can in principle mess up the runtime state of *all* modules. This has several undesirable consequences:

- While testing a partially verified program, failures may still occur in the verified part of the program, and the root cause for such failures may be hard to track down. This includes both memory safety failures (e.g. dereferencing invalid memory addresses) as well as failures of assertions that were statically verified to hold.
- Security properties verified to hold in a module are not guaranteed to hold when that module is used as part of a larger, unverified program. Hence, the benefits of partial verification for security purposes are limited. In particular, in a security setting, one must consider that memory safety errors may be *exploited* by means of code injection attacks [14]. Maintaining the integrity of a verified module in such a setting is very challenging.

What is needed is a technique for ensuring that failures cannot occur in the verified part of the program. Any runtime error should either (1) lead to a failure in the unverified part, or (2) be detected on entry to the verified part. This will entail that the state of the verified module is always valid, and that no properties that were verified to hold for this module state will ever be violated.

The approach we suggest is based on performing *runtime checks* at the *boundary* between the verified and unverified part of the program. While sound approaches for such dynamic contract checking exist for safe languages [15, 16], to the best of our knowledge there is no system that achieves such sound contract checking for unsafe languages such as C. Furthermore, existing approaches instrument *each* memory access in the unverified part [29] or verified part [34], entailing a large performance cost, while we will perform our checks only when crossing the verified-unverified boundary.

The main contribution of this paper is the development of a program transformation that, given a C program partitioned into an unverified module and a module verified by means of separation logic, transforms the verified module into a *hardened* module that includes sound and complete runtime checks at the boundaries of the module. A key problem that needs to be solved is how to make sure that memory errors (or alternatively, malicious code) in the unverified part cannot corrupt the state of the verified module, while still only performing explicit checks at the boundary. We solve this problem in two steps. First, the boundary checks perform integrity checks on the heap footprint of the verified module: on re-entry to the verified module, we check that the heap memory "owned" by the verified part has not been changed by the unverified part of the program. This ensures that bad writes to the heap performed by the unverified part are detected. Second, we need a mechanism for protecting the integrity of local variables and control flow metadata of the verified module. For this, we rely on the recent work on *protected module architectures* (PMAs), which are low-level security architectures providing support for module-private data regions and restricted module code entry points. Early prototypes [26, 35] are hypervisor-based [37], while the most recent research prototypes [30] implement this protection in hardware, thereby reducing the trusted computing base to just the processor itself. Intel recently announced hardware support for PMA's under the name Intel Software Guard Extensions (SGX) [20], hence this type of protection mechanisms will be broadly available in the near future.

The combination of the module boundary checks and the PMA protection of local variables and control flow gives us a very strong modular soundness guarantee: *no verified assertion in the verified module will ever fail at runtime, even if the module runs as part of a vulnerable application that is subject to code injection attacks*. The main contributions of this paper are:

- The development of runtime checks for separation logic contracts at the boundary between a verified and an unverified part of a C program.
- A formal correctness proof of these runtime checks.
- The observation that existing fully abstract secure compilation techniques [2, 31] to a PMA ensure the soundness of these runtime checks even in the presence of code injection attacks exploiting memory safety errors in the unverified part.
- The development of a robust prototype, based on the VeriFast [22] verifier.
- The quantification of the performance cost of the technique by means of micro and macro benchmarks.

The remainder of this paper is structured as follows. First, we elaborate on the problem we solve and on our proposed solution in Sections 2 and 3. Section 4 gives an informal explanation of our program transformations, and Section 5 illustrates them using an example program. Section 6 formalizes the transformations. In Section 7 we discuss our prototype implementation and the results of our benchmarks. Finally, we discuss related work in Section 8 and conclude in Section 9.

2. Problem Statement

We assume as given a separation logic-based program logic for C [33], and a sound modular static verifier that checks compliance of C functions with contracts expressed in the program logic. For concreteness, we work in this paper with the VeriFast verifier [22], but our results are not specific to VeriFast. We use VeriFast syntax in our examples: separating conjunction is written as `&*&`, and `?x` introduces an existentially quantified logic variable.

For programs in which each function is statically verified and where the `main` function has an empty precondition, verification ensures that no routine ever performs an illegal memory operation and each routine upholds its contract. However, verifying the entire code base of a program is often infeasible, for instance because it is too costly in terms of programmer effort. Trying to prove full program correctness properties for partially verified programs would clearly be overambitious, since there are no guarantees about the behavior of the unverified parts. However, as this section will point out, even statements concerned only with the verified parts of the program cannot be proven in general for partially verified programs written in memory unsafe languages.

We consider single-threaded C programs consisting of two parts: an unverified *context* and a statically verified module. Each function of the verified module and each function prototype used by it specifies a separation logic contract, consisting of a precondition (**req**uires) and a postcondition (**ens**ures). Static verification ensures that the verified functions are memory safe and comply with their specifications, but only under the assumption that the precondition holds on function entry and that any function called from those functions complies to its own specification.

```
// Prototypes
int med(struct lst *l);


// Unverified functions
int main()
{
  struct lst *l =
    read_list();
  print(med(l));
}

void srt(struct lst *l)
{
  < unverified sort
    implementation >
}
```

```
// Prototypes
void srt(struct lst *l);
 req list(l, ?v0);
 ens list(l, ?v1) &*&
     val_eq(v0, v1) &*&
     sorted(v1);

// Verified functions
int med(struct lst *l)
 req list(l, ?v0) &*&
     0 < length(v0);
 ens list(l, v0) &*&
     res == median(v0);
{
  int s = len(l);
  struct lst *l0 = copy(l);
  srt(l0);
  <proof statements>
  return nth(l0, s/2);
}
```

Consider the example program shown above. On the left is the context, consisting of a `main` function, a `srt` function and a prototype for a function `med`. On the right is the verified module, which contains the function `med` and a prototype for `srt`. The `med` function takes as input a non-empty list and claims that, after execution, the list still contains the original values and the return value will be the median of the input list. This function relies on functions `len`, `nth` and `srt` to perform its task. The verifier relies on the contracts of those functions, in addition to the proof statements in `med` itself, to prove that the function will uphold its contract. The implementation of `len` and `nth` is not shown in the example, but they are assumed to be part of the verified module, hence the verifier can verify those functions as well. On the other hand, for the `srt` function the verified module only contains a prototype. Hence the verifier can only *assume* that its implementation will uphold the specified contract.

Linking the two parts of the example program together and executing them may still lead to violations of the verified module spec-

ifications, if one of the functions has a *bug*. We say that a function has a bug if it does not comply with its contract. That is, there exists an execution of the function that satisfies its precondition, but exhibits an invalid memory access, violates the postcondition, or performs another function call and violates the precondition of the called function. We assume that the static verifier is *sound*, i.e., it rejects any function with a bug. Hence there are only bugs in the unverified context, not in the verified module.

For instance, if `read_list` (called from `main`) has a bug and returns an invalid (e.g., unallocated) memory address, the `len` function or some other verified function could perform an illegal memory operation. Likewise, if `srt` has a bug and violates the contract specified in its function prototype, then the verified function `med` might not uphold its contract either. Furthermore, because C is a memory unsafe language, `srt` can write to arbitrary memory locations, thereby modifying data belonging to the `med` function. For instance, `srt` could write to the original list `l`, instead of the copy `l0` that it was given. Hence, any properties verified to hold by the verifier about *l* while verifying `med` might be violated at runtime after a call to the unverified function `srt`. Note that `srt` can also *read* memory that it is not allowed to by its contract. This is also a bug, but it will not violate any property of the verified module assumed by the verifier, hence our runtime checks will allow this.

How bad the effects of bugs in the unverified part of the program can be, depends on how the program is executed. A *safe* execution performs complete runtime checking and will detect bugs as soon as they appear. Nguyen et al. [29] propose a way to perform such executions. Every memory access is checked and contracts are checked on each function entry and exit. Hence, safe executions are expensive. It is sound to remove the runtime checks from the verified module, as the soundness of verification implies that these checks will never fail in the safe execution. But as long as there is a significant unverified part, the performance cost will be high.

Because of this performance cost, executions of C programs are usually *not* safe. Hence, executions can enter an *error state* and continue executing. We say an execution is in an error state if it has performed memory accesses resulting in undefined behavior according to C semantics or if the execution has violated some of the separation logic specifications. An execution can only *enter* an error state in a function with a bug. That function is the *root cause* of the error state. An execution *fails* at the point where it detects the error state and terminates. Safe executions fail immediately at the root cause of a bug, but other executions may continue after entering an error state. Typically what happens then is implementation-dependent: the program behavior depends on details of the compiler and the machine on which the compiled code is executed. Most C compilers will generate code that may detect some error states such as the dereference of a memory address that the operating system has not even allocated to the program. But in general, failure of the execution may happen long after going into an error state. As a consequence, executions may enter an error state in the unverified context, but then fail in the verified module. The verified module may also be operating while in an error state, yet *not* fail, and possibly further mess up the runtime state and worsen the error state of the execution. This is exactly why partial verification is less useful than it could be, as discussed in the introduction.

Our approach is to develop efficient runtime checks to be inserted at the boundary between verified and unverified code, that make sure that no failures can occur in the verified module. Executions can enter an error state while executing in the unverified context and the execution may then continue in an error state, but we have the guarantee that *any error state that might impact the verified module will be detected before control flow enters the verified module*. As a consequence, we have that the execution never fails while control is inside the verified module.

3. Overview of our solution

To guarantee that error states never impact verified modules, we need to model the execution of programs with memory safety errors. We describe two such models below.

3.1 Control-flow safe execution

The *control-flow safe* execution models programs as commands that operate on a heap. This is a standard model of unsafe imperative programs in the separation logic literature [33]. Memory safety errors may impact any part of the heap, but they cannot modify local variables or the control flow. In other words, code-injection attacks or stack smashing are not modeled.

For the control-flow safe execution, it is sufficient to perform runtime checks at the boundary between the verified module and the unverified context. Roughly speaking, the checks that need to be performed at the boundary are the following. Each function of the verified module should (1) check that its precondition holds on entry from an unverified function, (2) check that the callee's postcondition holds after an *outcall* (i.e., a call from the verified module to an unverified function), and (3) ensure that unverified functions did not modify any heap locations that could affect the verified function's correct execution. In our approach, the first two checks are based on a translation of separation logic pre- and postconditions into equivalent C code that will check the validity of those conditions at runtime. For the third check, our approach keeps track of the *footprint* of the verified module, i.e., the memory locations that the module can read or write, and it uses an integrity check to ensure that unverified functions do not modify the contents of those locations (except for the locations explicitly allowed by the precondition of the called unverified function). Right before performing a call from a verified function to a function of the unverified context, a cryptographic checksum is calculated over the contents of the verified module's footprint, which is recalculated and compared against the original on re-entry of the verified module.

We develop a program transformation on the verified module that injects the necessary runtime checks in Section 4, and we prove them correct for the control-flow safe execution of programs in Section 6.

3.2 Unsafe execution

Of course, for most realistic C compilers, the control-flow safe execution model is too abstract. Control flow information and local variables (i.e., the runtime stack) are stored in the same memory space as the heap, and hence memory safety errors can also modify control flow or contents of local variables. This is particularly relevant if we consider the possibility that our program might be under attack, and an adversary provides input that triggers a memory safety error in the unverified part of the program by performing one of the many possible low level attacks against C programs [14].

Hence, we also consider *unsafe* executions, where programs are compiled in the standard way to a Von Neumann style processor architecture. Under such unsafe executions, the boundary checks discussed above are insufficient, as memory safety errors might corrupt the integrity checksum that the boundary checks compute. Also corruption of the control flow or of local variables in the verified functions may lead to failures in the verified module.

To restore the property that no failures occur in the verified module, we build on a recently proposed fully abstract secure compilation technique towards protected module architectures [2, 31]. This compilation technique protects modules from a potentially malicious context and ensures that any possible effect that the malicious context can have on the hardened module can be understood at source code level. By composing this secure compilation technique with the program transformation for the control-flow safe execution, we get the desired property that no failures can occur in

the verified module, even in the presence of stack-smashing, code injection attacks or other exploitations of memory safety errors in the unverified context.

4. Program transformations

This section describes how a verified module can be transformed into a *hardened* module containing runtime boundary checks, and how our prototype implements these checks. At an architectural level, the hardened module can be divided into a *functional* part and a *boundary checking* part. The former is essentially a copy of the verified module given as input, where the separation logic contracts and proof statements have been removed and the functions have been given a fresh name and marked `static` in order to remove them from the module's public interface. The latter part contains new functions and data structure definitions to actually perform the runtime boundary checks. The hardened module is constructed such that each transition between the context and the functional part passes through the appropriate function of the boundary checking part. The transformation is based solely on the source code of the verified module and the annotated function *prototypes* of the unverified part. Hence, the transformation does not require access to the source code of the unverified part. The resulting hardened module can be linked with the unverified part as-is: no recompilation of the unverified part is necessary.

The sections below explain concretely how different kinds of separation logic constructs can be translated into C code for checking them. We assume the control-flow safe execution model, since it is the execution model provided to us by the fully abstract compilation scheme, as described in Section 3.2.

4.1 Pure assertions

Pure assertions, as opposed to *spatial* assertions, only reference local variables and hence do not make claims about the heap. Such assertions can be translated straightforwardly into a C expression, as shown by the example below.

```
// Original module
int fac(int x)
 req x >= 0;
 ens res > 0;
{
  if (x == 0) return 1;
  int p = prod(x,
    fac(x-1));
  return p;
}

int prod(int x, int y);
 req true;
 ens res == x * y;
```

```
// Hardened module
// (Functional part)
static int _fac(int x) {
  if (x == 0) return 1;
  int p = _prod(x,
    _fac(x-1));
  return p;
}

// (Boundary checking part)
static
int _prod(int x, int y) {
  int r = prod(x, y);
  if (! (r == x * y)) trap();
  return r;
}

int fac(int x) {
  if (! (x >= 0)) trap();
  return _fac(x);
}
```

All assertions in this example, i.e., the contracts of `fac` and `prod`, are pure. In the hardened module, `fac` has been replaced by an *entry stub* that first checks the precondition, before calling `_fac`, which is a slightly modified version of the original `fac`. The only functional difference is that the modified version calls the `_prod` *outcall stub* instead of the original function `prod` in the context. The outcall stub first calls `prod` in the context and then checks whether the postcondition holds. If any check fails, the `trap` function is called, which ends execution. The functions `_prod` and `_fac` have been marked `static` to indicate they are not in the public interface of the hardened module.

4.2 Spatial assertions

Spatial assertions describe (parts of) the heap: they indicate that a certain memory region should contain certain values. The assertions need not necessarily specify exactly what those values are, they can instead existentially quantify over them, by binding a logic variable (see, for instance, the logic variable bindings `?a` and `?b` in the precondition of the original module in the example below). The difficulty with spatial assertions is that a function in the context can overwrite these values, even though that function might not be allowed to do so by its contract, thereby possibly violating properties of the verified module verified to hold by the verifier. In separation logic terms, this corresponds to a violation of the *frame rule*. As described in Section 3, we use a cryptographic checksum over the memory footprint of the hardened module to solve this problem.

```
// Original module
struct pair {int a, b;};


void f(struct pair* p)
 req p->a |-> ?a &*&
     p->b |-> ?b
 ens p->a |-> _ &*&
     p->b |-> _;
{
 <...>
 ct(p);
 <...>
}


void ct(struct pair* p);
 req p->a |-> ?n;
 ens p->a |-> ?m &*&
     m == n + 1;
```

```
// Hardened module
struct pair {int a, b;};

static
void _f(struct pair* p) {
 <...>
 _ct(p);
 <...>
}

static
void _ct(struct pair* p) {
 char h0[32], h1[32];
 int n = intp(&(p->a),C);
 fp_hash(h0);
 ct(p);
 fp_hash(h1);
 if (!eq(h0, h1)) trap();
 int m = intp(&(p->a),P);
 if (m != n+1) trap();
}

void f(struct pair* p) {
 a = intp(&(p->a),P);
 b = intp(&(p->b),P);
 _f(p);
 intp(&(p->a),C);
 intp(&(p->b),C);
}
```

The code above shows our approach. In the hardened module on the right, the verified function `f` has been replaced by an entry stub that first calls `intp` for both integer *points-to* assertions in the precondition of the original `f`, then calls `_f` and finally calls `intp` again for both integer *points-to* assertions in the postcondition.

The `intp` function is a *data type checking function* provided by our runtime checking system. It takes as arguments a pointer `p` to an integer and an enumeration value, and performs two important functions. The function first checks whether `p` points to a valid integer, which it does by simply reading `*p`. Secondly, the function adds or removes the memory region occupied by the integer (i.e., the memory region of `sizeof(int)` bytes starting at address `p`) to or from a global data structure describing the hardened module's footprint. When the enumeration value is `P` (for *produce*), the memory region is added to the footprint and when it is `C` (for *consume*) the region is removed. When adding a region to the footprint, `intp` checks that the region does not overlap with the existing footprint. This corresponds to the semantics of the separating conjunction, which requires that the footprint of each of its conjuncts occupies a *disjoint* part of the heap. We do not support non-separating conjunction but argue in Section 6.3 that this does not pose expressibility problems. If all checks pass, the `intp` function returns the integer value at the given address, or ends execution by calling `trap` otherwise. Functions similar to `intp` are provided for other prim-

itive data types (`char`, `unsigned int`, ...) and pointers, because the memory sizes of those data types can differ.

In the function `_f` of the hardened module, the call to the `ct` function of the context has been replaced by a call to the `_ct` outcall stub. This stub first removes (consumes) the footprint of `ct`'s precondition from the hardened module's footprint, before calculating a hash over the memory regions described by the remaining footprint. This hash is stored in the local variable `h0`, where it is protected from the context by the secure compilation scheme. Next, the stub calls `ct`, handing over control to the context. When the context function returns, the outcall stub verifies that the memory described by the footprint has not been tampered with, by recalculating the hash and comparing it to the original value stored in `h0`. Finally, the stub checks whether the postcondition of `ct` holds by producing its footprint and checking whether the values in the corresponding memory region adhere to the contract. Note that we do not prevent the context from *reading* memory it is not supposed to by its contract, because this can never violate properties of the verified module assumed by the verifier.

For our prototype, we implemented the footprint data structure as a radix trie. This data structure supports $O(k)$ addition, removal and overlap testing, and $O(n)$ traversal, with k the number of bits in a memory address (e.g. 64 for a 64-bit CPU) and n the number of memory regions in the trie. The footprint description must be protected from being overwritten by memory safety errors in the unverified context and hence we store it in the module-private memory section provided by the PMA.

4.3 Predicates

Separation logic predicates are named, parameterized assertions, used to provide data encapsulation and to describe recursive data structures. For instance, the following predicate describes a linked list of integers of a certain size.

```
struct list { int value; struct list* next; };

pred list_pred(struct list* l; int size) =
  l == 0 ? size == 0 : l->value |-> _ &*& l->next |-> ?n
      &*& list_pred(n, ?size0) &*& size == size0 + 1;
```

Predicates can have an arbitrary number of parameters and separation logic allows us to existentially quantify over each of them. For instance, a valid precondition could be `list_pred(?l, 5)`, meaning that there must be a linked list of size 5 somewhere on the heap. Translating this quantification into a runtime check would however be problematic, since in general the entire heap would need to be examined in order to bind a value to `l` at runtime. Hence, we require predicates to be *precise*: predicates must separate their parameters into input and output parameters and only output parameters can be quantified over when using a predicate. In a predicate definition, each output parameter of the predicate must be assigned a value in all execution paths of the predicate's body. We discuss the implications of this requirement in Section 6.3. In VeriFast syntax, parameters before the semicolon in the parameter list of a predicate definition are input parameters and the other parameters are output parameters. Hence, for `list_pred` defined above, `l` is an input parameter and `size` is an output parameter.

```
static void
list_pred(struct list* l, int* size, enum op_type op) {
  if (l == 0) { *size = 0; }
  else {
    intp(&(l->value), op);
    struct list* n = ptrp(&(l->next), op);
    int size0;
    list_pred(n, &size0, op);
    *size = size0 + 1;
  }
}
```

The code above shows the transformation of the `list_pred` predicate. It is a *predicate checking function* with one more parameter than the original predicate. This extra parameter is an enumeration value indicating whether the predicate will be used for consumption or production and its value is simply passed on to the data type checking functions (e.g. `intp`) described in Section 4.2. Input parameters have the same type in the runtime checking function as in the original predicate and output parameters are *pointers* to the type of the parameter in the original predicate. The predicate's body is transformed straightforwardly into equivalent C code. When an output parameter is assigned a value in the predicate body, the corresponding pointer parameter in the runtime checking function is assigned a value as well. As exemplified by the recursive call to `list_pred`, a predicate call assertion is transformed into a call to the corresponding predicate checking function. If an assertion uses a constant value or previously bound variable for an output argument instead of binding a new variable (e.g. `list_pred(l, s)` with `s` already bound, instead of `list_pred(l, ?s)`), then this is pre-transformed to first binding a fresh variable to the output parameter and then constraining it with an equality (e.g. `list_pred(l, ?s0) &*& s0 == s`). This allows the core transformation to assume that output arguments are always existentially quantified.

4.4 Inductive data types

While spatial assertions and predicates are in some cases sufficient to prove memory safety, they are insufficient to prove full functional correctness for most programs. For instance, the `list_pred` predicate defined in the previous section specifies the size and memory footprint of a linked list, but does not say anything about its *contents*. VeriFast supports a rich specification language with inductive data types and fixpoint functions (i.e., primitive recursive functions) over them. The example below shows how such constructs can be used for specifying functional correctness properties.

```
induct ints = ints_nil() | ints_cons(int, ints);

pred list_pred(struct list* l; ints values) =
  l == 0 ?
    values == ints_nil()
  :
    l->value |-> ?v &*& l->next |-> ?n &*&
    list_pred(n, ?vs_tail) &*&
    values == ints_cons(v, vs_tail);

fixpoint int head(ints lst) {
  switch (lst) {
    case ints_nil(): return 0;
    case ints_cons(v, tail): return v;
  }
}

int get_first(struct list* l)
  req list_pred(l, ?values);
  ens list_pred(l, values) &*& res == head(lst);
{
  <implementation omitted>
}
```

The behavior of `get_first` is completely specified by its contract. Our transformation translates such inductive data type specifications into *tagged structures*, a known technique for implementing variant types in C. The code below shows the data structure definitions corresponding to `ints` and its two constructors.

```
struct ints { int tag; };

struct ints_nil {
  int tag;
  // No members
};

struct ints_cons {
  int tag;
  int _1;
  struct ints* _2;
};
```

To prevent the context from tampering with instances of these data structures when the context is in control, the data must either be stored in the private memory section provided by the PMA, or be included in the module's footprint such that it's incorporated in the cryptographic checksum described in Section 4.2. We chose to store the data in the private memory section, which is faster than the checksum-based approach, but does require more private memory.

Besides these structure declarations, the transformation also generates an equality comparison function for each inductive data type. Finally, fixpoint function definitions are translated straightforwardly into equivalent C functions.

4.5 Function pointers

VeriFast allows programs using function pointers to be verified, by letting users associate contracts with *classes* of functions. The code below shows how a verified module can use a function pointer to call a function in the unverified part.

```
typedef                          void f(int_func* g, int x)
int int_func(int x);               req is_int_func(g);
  req true;                        ens true;
  ens result > 0;                {
                                     int y = g(x);
                                     <...>
                                   }
```

The typedef on the left defines `int_func` as the class of functions that take an integer argument and return a strictly positive integer result. On the right, the function `f` takes a pointer `g` to such a function, applies it to a local variable `x` and stores the result in `y`. The contract for `g` is specified by the `is_int_func(g)` assertion in the precondition of `f`, which refers to the typedef on the left.

Our transformation handles function pointer calls almost the same way it handles regular calls. That is, an outcall stub is generated for each defined function pointer typedef, and the hardened module calls this outcall stub instead of calling corresponding function pointers directly. Function pointer outcall stubs take as an argument a pointer to the concrete function to call. For instance, for the example code above, the function pointer outcall stub would be:

```
static int _int_func(int_func* f, int x) {
   <calculate footprint hash>
   int result = f(x);
   <verify footprint hash, check postcondition>
   return result;
}
```

Indirect function calls from the context to the hardened module are also supported naturally. Since all functions of the functional part of the hardened module have been made `static`, the only publicly accessible functions of the module are those in the boundary checking part. The fully abstract compilation scheme uses the PMA's restriction on module entry points to ensure that only those public functions can be called from the context at runtime. Hence, the necessary runtime checks are performed whenever the unverified part calls the hardened module through a function pointer.

5. Example program

This section uses an example program to illustrate how the transformations described above affect the hardened module's footprint description. Figure 1 depicts the example program and the footprint at various execution points. Note that the program is an abstract example created to illustrate the footprint evolution under various control flow transitions, and is not intended to model any useful computation. The program consists of the verified functions `vf1`, `vf2` and `vf3`, and, in addition to the standard C library, two unverified functions `main` and `uvf`. The unverified function `uvf` is annotated with a separation logic contract, so it can be called from

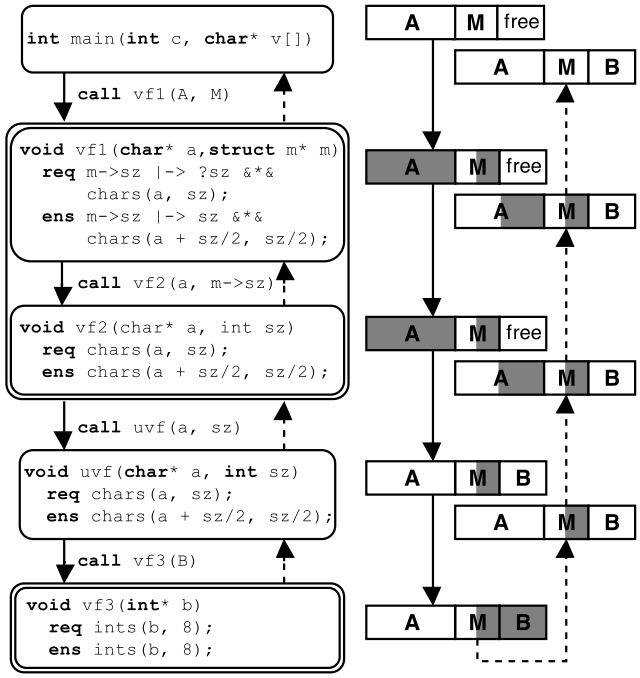


Figure 1. When executing the example program on the left, the footprint evolves as shown on the right. The single-bordered boxes on the left are unverified functions while the double-bordered boxes are verified functions. The boxes on the right represent the heap, with the grayed-out parts representing the hardened module's footprint. Solid lines are function calls and dashed lines are returns.

the verified functions. The function signatures and contracts shown in Figure 1 refer to a `struct m` and predicate `chars`, defined as:

```
struct m {                pred chars(char *a, int sz) =
  int x; int sz;           sz <= 0 ? true :
};                         char(a) &*& chars(a+1, sz-1);
```

A `chars(a, sz)` instance represents a character array of size `sz`, starting at heap address `a`. The `char` predicate used by `chars` is a VeriFast primitive which asserts that its argument points to a valid character in memory. Its translation is a call to the `charp` data type checking function (similar to `intp` described in Section 4.2).

We now discuss how each of the function calls shown in Figure 1 affects the hardened module's footprint description. Assume `main` allocates an array `A` of `N` bytes and an instance `M` of `struct m`, using the standard (unverified) `malloc` function, and assigns the value `N` to `M`'s `sz` field. At this point, the heap contains `A` and `M` and the footprint description is empty. Next, `main` calls the verified function `vf1(A, M)`, and hence this function's entry stub will check whether its precondition holds. The precondition check consists of reading the `sz` field of `M` and verifying that `A` is in fact a character array of size `N`. As part of this check, the memory occupied by `M`'s `sz` field and the entire array `A` will be added to the footprint description. The memory occupied by `M`'s `x` field is not mentioned in `vf1`'s precondition and hence is not added to the footprint description.

Next, `vf1` calls `vf2(a, m->sz)` directly, without passing through a boundary checking function, because both caller and callee are in the hardened module. No runtime checks are performed and hence the footprint description remains the same.

The `vf2` function now makes an outcall `uvf(a, sz)`, which passes through the corresponding outcall stub. The stub first removes the array `A` from the footprint description, because it is ref-

erenced in `uvf`'s precondition, and then hashes the memory in the remaining footprint description (consisting of only the `sz` field of `M`), before calling `uvf`.

We assume `uvf` allocates an array `B` of eight integers, again using the standard `malloc` function, and then calls `vf3(B)`. This function's entry stub will verify that `B` is a valid array of eight integers and will add `B` to the footprint description. The `vf3` function now executes its (unspecified) body and then returns, thereby removing `B` from the footprint description, as specified by its postcondition.

Now `uvf` is back in control and returns to its outcall stub in the hardened module. The stub will first check that the memory in the footprint (still consisting of only `M`'s `sz` field) has not been modified, by recalculating the hash and comparing it to the original. Execution is aborted if any change is detected. If the new hash matches the original, the stub checks whether `uvf`'s postcondition holds and adds the second half of `A` back to the footprint description, as specified by `uvf`'s postcondition. The stub then returns control to `vf2`. Note that it is impossible that `vf2` now tries to access the first half of `A`, since this would have been detected by the static verifier when verifying `vf2`.

The `vf2` function now returns to `vf1`, without any change in the footprint description, because both functions are part of the hardened module. Finally, `vf1` returns to `main`, removing the second half of `A` and `M`'s `sz` field from the footprint description, as specified by `vf1`'s postcondition. Control is now back at the `main` function and the footprint description is empty.

6. Formalization

This section formalizes the transformations described in Section 4 and shows that they are *safe* and *precise*. Safety means that the hardened module does not fail, even when it is interacting with a context that does not uphold its contracts. Precision means that the hardened module behaves exactly like the original verified module when interacting with a context that *does* uphold its contracts. Focusing on the essentials of the transformation, we do not formalize inductive data types nor function pointers. Due to space constraints, we only give a high-level overview of the formalization here and leave the full text and proofs for our extended technical report [3].

6.1 Programming language

We first define a simple imperative programming language that models C. The syntax of this language is defined in Figure 2 and its operational semantics are described in Figure 3. We write $[\![e]\!]_s$ to indicate the value of an expression e evaluated under a store s, assuming standard non-negative integer expression evaluation. In addition to standard imperative language constructs such as heap lookup, mutation, allocation and deallocation, the language provides two assertion commands. The first is **assert**(b), which asserts that the boolean expression b holds and the other is **alloced**(l), which asserts that the memory address l is allocated. Both commands evaluate to **skip** if the assertion succeeds and to **trap** otherwise. Trapping (as opposed to failing) is a clean way of indicating an abnormality: our runtime checks trap whenever they discover a bug in the context. Once a program traps, it remains in the trapped state forever (i.e., it diverges).

Program states $\Sigma \vdash \langle \overline{s}, h\rangle \mid c$ consist of a map Σ from routine names to routine definitions, a stack $\overline{s}$, a heap h and the program under execution c. The stack is a list of stores s, each of which is a partial function from $Vars$ to $\mathbb{N}^+$. The heap is a partial function from memory locations in $\mathbb{N}^+$ to values in $\mathbb{N}^+$. Execution starts from an empty store and heap.

We assume our programs are well-formed. That is, they never try to use a variable that was not defined earlier in the code, and they do not refer to undefined routines.

Definition 1 (Failure)**.** *We say a program c* fails *when it is stuck. For well-formed programs this corresponds to reading, writing or deallocating an unallocated memory location.*

Since we prove safety for *any* hardened module, absence of failure according to this definition also implies absence of assertion failures.

6.2 Separation logic

We define separation logic triples of the form $\Gamma \vdash \{P\}\ c\ \{Q\}$ and $\Delta \vDash \{P\}\ c\ \{Q\}$. Triples of the first form mean that the *static verifier* asserts that, given the function Γ mapping routine names to contracts, if P holds then c will not fail and Q will hold after executing c. The Γ corresponds to the *prototypes* of the functions of the context (including their contracts). Triples of the latter form mean that if P holds in some state (stack and heap), then *executing c under the context* Δ won't fail and Q will hold in the resulting state. The Δ corresponds to a concrete context, in the form of a map from routine names to routine definitions.

Because the verifier is sound, $\Gamma \vdash \{P\}\ c\ \{Q\}$ implies $\Delta \vDash \{P\}\ c\ \{Q\}$, under the critical condition that *the routines of Δ uphold the contracts defined in Γ*. The essence of our formalization is to show that our program transformation will allow us to discard this critical condition. We are only concerned with partial correctness, so c is allowed to diverge, which is what happens when our runtime checks trap (see Figure 3). The assertions P and Q are defined as:

$$P, Q ::= b \mid b\ ?\ P : Q \mid e \mapsto ?x \mid p(e, ?x) \mid y := e \mid P * Q \mid P \wedge Q \mid P \vee Q \mid \neg P \mid \top$$

where booleans b and expressions e are defined as in Figure 2, p refers to a user-defined predicate **pred** $p(x, y) = a$ and $?x$ introduces a logic variable x. The first parameter of a user-defined predicate is an input parameter and the second is an output parameter. The assignment assertion $y := e$ is used to bind a value to predicate output parameters. The formal semantics of this assertion language are defined in Figure 4, where a judgment $s, h \vDash P \rightsquigarrow s'$ means that P holds under store s and heap h and binds new logic variables (using $e \mapsto ?x$ and $p(e, ?x)$) to end up with the updated store s'.

6.3 Contract assertion language

Although assertions in our meta-theory range over the full language defined above, routine and predicate contracts come from a more restricted language of *precise* assertions.

$$a ::= b \mid b\ ?\ a : a \mid e \mapsto ?x \mid p(e, ?x) \mid y := e \mid a * a$$

In particular, routine and predicate assertions do not include standard conjunction, disjunction, negation nor top. Furthermore, we require user-defined predicates to constrain their output parameter to a single value (using the $y := e$ construct) exactly once on each possible execution path of their body. These requirements make existential quantification *constructive*: assertions indicate how each variable can be assigned a value, thereby avoiding an exhaustive search which would entail an enormous runtime performance cost.

While excluding disjunction, negation and non-separating conjunction between spatial predicates might seem to limit the expressiveness of the contract assertion language, this language subset corresponds exactly to the assertion languages supported by VeriFast [22], Smallfoot [6] and other separation logic-based program verifiers [8, 13]. Extensive experience with these tools has shown that this subset is sufficiently expressive for practical purposes [32].

In the rest of the text, we will consistently use symbols P and Q for meta-level assertions and symbol a for contract assertions.

6.4 Transformations

In this section, we formalize the transformations described in Section 4. We start by defining a function $\mathsf{prod}(a)$ from assertions to commands that models the *production* of a (see Section 4.2). The code generated by this function assumes there is a program variable `fp` containing a set of memory locations that represents the hardened module's current footprint. The generated code will (1) trap if a does not hold or if its footprint would overlap with the footprint in `fp`, (2) create a program variable x for each logic variable $?x$ in a and (3) add the assertion's footprint to `fp`. The function is defined as follows.

$$\begin{aligned}
\mathsf{prod}(y := e) &= (y := e) \\
\mathsf{prod}(b) &= \textbf{assert}(b) \\
\mathsf{prod}(b\ ?\ a_1 : a_2) &= \textbf{if}\ b\ \textbf{then}\ \mathsf{prod}(a_1)\ \textbf{else}\ \mathsf{prod}(a_2) \\
\mathsf{prod}(e \mapsto ?x) &= \begin{cases} x := e;\ x := \mathtt{in}(x, \mathtt{fp}); \\ \textbf{assert}(x = 0);\ x := e;\ \textbf{alloced}(x); \\ \mathtt{fp} := \mathtt{add}(x, \mathtt{fp});\ x := [x] \end{cases}
\end{aligned}$$

where `fp` is the program variable that stores the footprint, $\mathtt{in}(x, y)$ returns 1 if x is in the list represented by y and 0 otherwise, and $\mathtt{add}(x, y)$ adds x to the list represented by y.

$$\mathsf{prod}(p(e, ?x)) = (x := e;\ \{\mathtt{fp}, x\} := prod_p(\mathtt{fp}, x))$$

where $prod_p$ implements the production part of the *predicate checking routine* for p, defined as **routine** $prod_p(\mathtt{fp}, x) = \mathsf{prod}(a)$; $\mathtt{res} := \{\mathtt{fp}, y\}$ with a the body of **pred** $p(x, y)$. The $\{\mathtt{fp}, x\}$ is syntactic sugar for a tuple consisting of `fp` and x.

$$\mathsf{prod}(a_1 * a_2) = \mathsf{prod}(a_1); \mathsf{prod}(a_2)$$

Next we define the additional transformation functions required to safely perform *outcalls* from a hardened module to routines of the context that might not uphold their contract. We first define $\mathsf{cons}(a)$, a function that models the *consumption* of an assertion. As explained in Section 4.2, this function needs to be called right before making an outcall, to consume (remove) the footprint of the context function's precondition from the current footprint `fp`. The structure of $\mathsf{cons}(a)$ is identical to that of $\mathsf{prod}(a)$, but it has to *remove* a's footprint instead of adding it. Furthermore, there is no need for $\mathsf{cons}(a)$ to check that a actually holds, because the static verifier already ensured this when checking the verified module. Hence, we do not need to use the **assert** and **alloced** commands in the definition of $\mathsf{cons}()$. The function is defined as follows.

$$\begin{aligned}
\mathsf{cons}(y := e) &= (y := e) \\
\mathsf{cons}(b) &= \textbf{skip} \\
\mathsf{cons}(b\ ?\ a_1 : a_2) &= \textbf{if}\ b\ \textbf{then}\ \mathsf{cons}(a_1)\ \textbf{else}\ \mathsf{cons}(a_2) \\
\mathsf{cons}(e \mapsto ?x) &= x := e;\ \mathtt{fp} := \mathtt{rem}(x, \mathtt{fp});\ x := [x]
\end{aligned}$$

where $\mathtt{rem}(x, y)$ removes x from the list represented by y.

$$\mathsf{cons}(p(e, ?x)) = (x := e;\ \{\mathtt{fp}, x\} := cons_p(\mathtt{fp}, x))$$

where $cons_p$ implements the consumption part of the *predicate checking routine* for p, defined as **routine** $cons_p(\mathtt{fp}, x) = \mathsf{cons}(a);\ \mathtt{res} := \{\mathtt{fp}, y\}$ with a the body of predicate **pred** $p(x, y)$

$$\mathsf{cons}(a_1 * a_2) = \mathsf{cons}(a_1); \mathsf{cons}(a_2)$$

Now we just need a function $\mathsf{harness}_\Gamma(r)$ that can generate outcall stubs for routines $r(\overline{x})$ of the context. This function takes a mapping Γ from routines to contracts, corresponding to the prototypes of functions defined in the context. For $\Gamma(r) = (a_{pre}, a_{post})$, $\mathsf{harness}_\Gamma(r)$ is defined as:

$$\begin{aligned}
&\textbf{routine}\ stub_r(fp,\ \overline{x}) = \\
&\quad \mathsf{cons}(a_{pre});\ s := \mathtt{snap}(fp); \\
&\quad res := r(\overline{x}); \\
&\quad s' := \mathtt{snap}(fp);\ \textbf{assert}(s = s'); \\
&\quad \mathsf{prod}(a_{post});\ res := \{fp,\ res\}
\end{aligned}$$

where we assume all introduced variables are fresh and $\mathtt{snap}(x)$ returns a snapshot of the contents of the footprint x (corresponding to calculating the hash of the footprint, as described in Section 4.2).

Finally, we can define the full transformation function $[c]_{\Gamma,a}$, using a helper function $[c]'_\Gamma$, as follows:

$$[x := r(\overline{x})]'_\Gamma = \{fp,\ x\} := stub_r(fp, \overline{x})$$

where r is a routine of the context and $stub_r$ is the name of the outcall stub routine generated by $\mathsf{harness}_\Gamma(r)$

$$\begin{aligned}
[c_1;\ c_2]'_\Gamma &= [c_1]'_\Gamma;\ [c_2]'_\Gamma \\
[\textbf{if}\ b\ \textbf{then}\ c_1\ \textbf{else}\ c_2]'_\Gamma &= \textbf{if}\ b\ \textbf{then}\ [c_1]'_\Gamma\ \textbf{else}\ [c_2]'_\Gamma \\
[x := \textbf{alloc}]_\Gamma &= x := \textbf{alloc};\ \mathtt{fp} := \mathtt{add}(x, \mathtt{fp}) \\
[\textbf{dealloc}(x)]_\Gamma &= \textbf{dealloc}(x);\ \mathtt{fp} := \mathtt{rem}(x, \mathtt{fp}) \\
[c]'_\Gamma &= c \qquad \text{(for all other kinds of } c\text{)}
\end{aligned}$$

The full transformation function $[c]_{\Gamma,a}$ then is

$$[c]_{\Gamma,a} = (\mathtt{fp} := \emptyset;\ \mathsf{prod}(a);\ [c]'_\Gamma)$$

6.5 Safety and precision

We now come to the two crucial properties our transformation must have: safety and precision.We only state the theorems here, but leave the full proofs for our extended technical report [3]. We first need a new definition before we can state these main theorems.

Definition 2 (No-fail)**.** *The function* $\mathsf{nofail}(\Delta)$ *returns a non-failing variant of the program* Δ.

How nofail works is of no importance to our formalization, but one can see that the **alloced** command could be used to check each memory location before accessing it, thereby preventing failure.

The safety theorem states that if the context does not fail, but does not necessarily uphold its contracts either, then the hardened module will never fail.

Theorem 1 (Safety)**.** *For any command* c*, environment* Γ*, assertions* a_{pre} *and* a_{post} *such that* $\Gamma \vdash \{a_{pre}\}\ c\ \{a_{post}\}$*, and an arbitrary context* Δ*, we have* $\mathsf{nofail}(\Delta) \vDash \{\top\}\ [c]_{\Gamma,a_{pre}}\ \{\top\}$.

Finally, our precision theorem states that our transformations do not change the expected behavior of the hardened module when the context upholds its contracts.

Theorem 2 (Precision)**.** *For any command* c *and assertions* a_{pre} *and* a_{post} *such that* $\Gamma \vdash \{a_{pre}\}\ c\ \{a_{post}\}$*, we have that* $\forall\Delta.\ \Delta \vDash \Gamma \Rightarrow \Delta \vDash \{a_{pre}\}\ [c]_{\Gamma,a_{pre}}\ \{a_{post}\}$.

The $\Delta \vDash \Gamma$ condition means that the context Δ upholds the contracts specified by Γ. Under standard separation logic, this means (amongst other conditions) that context functions cannot read outside the footprint specified by their precondition. However, our precision theorem can actually be slightly stronger than this, because the theorem holds even if the context is allowed to read outside its designated footprint, as argued in Section 4.2.

7. Prototype performance

We have implemented the transformations described in Section 4 as a source-to-source translator written in OCaml. This translator

$$
\begin{array}{ll}
e ::= n \mid x \mid e + e \mid e - e & b ::= true \mid false \mid e = e \mid e < e \mid \neg b \\
c ::= x := e \mid x := [x] \mid [x] := x \mid x := r(\overline{x}) & l, n \in \mathbb{N}^+ \quad x, y, z \in Vars \\
\quad \mid \textbf{if } b \textbf{ then } c \textbf{ else } c \mid x := \textbf{alloc} \mid \textbf{dealloc}(x) & Routine ::= \textbf{routine } r(\overline{x}) \ = c \\
\quad \mid \textbf{assert}(b) \mid \textbf{alloced}(x) \mid \textbf{skip} \mid c; c & Program := \mathcal{P}(Routine)
\end{array}
$$

Figure 2. Syntax definition of our imperative language

$$E ::= \ \langle\ \rangle \mid E; c \qquad \text{Evaluation contexts}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![e]\!]_s = n \qquad \overline{s'} = s[x \to n] :: \overline{s_t}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[x := e] \to \Sigma \vdash \langle \overline{s'}, h\rangle \mid E[\textbf{skip}]} \text{ varAssign}$$

$$\frac{}{\Sigma \vdash \langle \overline{s}, h\rangle \mid \textbf{trap} \to \Sigma \vdash \langle \overline{s}, h\rangle \mid \textbf{trap}} \text{ trapLoop}$$

$$\frac{}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{skip}; c] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid E[c]} \text{ skip}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![e]\!]_s = true}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{if } e \textbf{ then } c \textbf{ else } c'] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid E[c]} \text{ ifTrue}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![e]\!]_s = false}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{if } e \textbf{ then } c \textbf{ else } c'] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid E[c']} \text{ ifFalse}$$

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & s(x') = l \\ l \to n \in h & \overline{s'} = s[x \to n] :: \overline{s_t}\end{array}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[x := [x']] \to \Sigma \vdash \langle \overline{s'}, h\rangle \mid E[\textbf{skip}]} \text{ heapRead}$$

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & s(x) = l \\ l \to n' \in h & h' = h[l \to s(x')]\end{array}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[[x] := x'] \to \Sigma \vdash \langle \overline{s}, h'\rangle \mid E[\textbf{skip}]} \text{ heapWrite}$$

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & s(\overline{y}) = \overline{n} \\ \overline{s'} = \{\overline{x} \to \overline{n}\} :: \overline{s} & \Sigma(r) = (\textbf{routine } r(\overline{x}) \ = c)\end{array}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[z := r(\overline{y})] \to \Sigma \vdash \langle \overline{s'}, h\rangle \mid E[c; \ z := \textbf{ret}]} \text{ call}$$

$$\frac{\overline{s} = s :: s' :: \overline{s_t} \qquad \overline{s'} = s'[x \to s(\text{`res'})] :: \overline{s_t}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[x := \textbf{ret}] \to \Sigma \vdash \langle \overline{s'}, h\rangle \mid E[\textbf{skip}]} \text{ return}$$

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & l \notin \{l' | l' \to n \in h\} \\ \overline{s'} = s[x \to l] :: \overline{s} & h' = h[l \to n']\end{array}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[x := \textbf{alloc}] \to \Sigma \vdash \langle \overline{s'}, h'\rangle \mid E[\textbf{skip}]} \text{ alloc}$$

$$\frac{\begin{array}{cc}\overline{s} = s :: \overline{s_t} & s(x) = l \\ l \to n \in h & h' = h \setminus \{l \to n\}\end{array}}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{dealloc}(x)] \to \Sigma \vdash \langle \overline{s}, h'\rangle \mid E[\textbf{skip}]} \text{ dealloc}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![x]\!]_s = l \qquad l \mapsto n \in h}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{alloced}(x)] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{skip}]} \text{ allocedTrue}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![x]\!]_s = l \qquad l \mapsto n \notin h}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{alloced}(x)] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid \textbf{trap}} \text{ allocedFalse}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![b]\!]_s = true}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{assert } b] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{skip}]} \text{ assertTrue}$$

$$\frac{\overline{s} = s :: \overline{s_t} \qquad [\![b]\!]_s = false}{\Sigma \vdash \langle \overline{s}, h\rangle \mid E[\textbf{assert } b] \to \Sigma \vdash \langle \overline{s}, h\rangle \mid \textbf{trap}} \text{ assertFalse}$$

Figure 3. Small-step operational semantics of our imperative language

$$\frac{[\![b]\!]_s = true \qquad h = \emptyset}{s, h \vDash b \rightsquigarrow s} \text{ pure}$$

$$\frac{[\![e]\!]_s = l \qquad h = \{l \to n\}}{s, h \vDash e \mapsto ?x \rightsquigarrow s[x \to n]} \text{ pointsTo}$$

$$\frac{[\![b]\!]_s = true \qquad s, h \vDash P \rightsquigarrow s'}{s, h \vDash b \ ? \ P : Q \rightsquigarrow s'} \text{ condTrue}$$

$$\frac{[\![b]\!]_s = false \qquad s, h \vDash Q \rightsquigarrow s'}{s, h \vDash b \ ? \ P : Q \rightsquigarrow s'} \text{ condFalse}$$

$$\frac{\begin{array}{cc}\textbf{pred } p(x, y) = P & [\![e]\!]_s = n \\ \{x \to n\}, h \vDash P \rightsquigarrow s' & m = s'(y)\end{array}}{s, h \vDash p(e, ?z) \rightsquigarrow s[z \to m]} \text{ predicate}$$

$$\frac{\begin{array}{cc}s, h_1 \vDash P \rightsquigarrow s' & s', h_2 \vDash Q \rightsquigarrow s'' \\ h_1 \perp h_2 & h = h_1 \cup h_2\end{array}}{s, h \vDash P * Q \rightsquigarrow s''} \text{ sepConj}$$

Where $h_1 \perp h_2 \Leftrightarrow dom(h_1) \cap dom(h_2) = \emptyset$

$$\frac{s, h \vDash P \rightsquigarrow s'}{s, h \vDash P \vee Q \rightsquigarrow s'} \text{ disjL} \qquad \frac{s, h \vDash Q \rightsquigarrow s'}{s, h \vDash P \vee Q \rightsquigarrow s'} \text{ disjR}$$

$$\frac{s, h_1 \vDash P \rightsquigarrow s' \qquad s', h_2 \vDash Q \rightsquigarrow s'' \qquad h = h_1 = h_2}{s, h \vDash P \wedge Q \rightsquigarrow s''} \text{ conj}$$

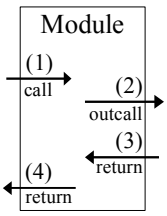
$$\frac{\begin{array}{cc}\textbf{pred } p(x, y) = P & [\![e]\!]_s = n \\ \{x \to n\}, h \vDash P \rightsquigarrow s' & s(z) = s'(y)\end{array}}{s, h \vDash p(e, z) \rightsquigarrow s} \text{ concrPredicate}$$

$$\frac{[\![e]\!]_s = l \qquad h = \{l \to n\} \qquad s(x) = n}{s, h \vDash e \mapsto x \rightsquigarrow s} \text{ concrPointsTo}$$

$$\frac{[\![e]\!]_s = n \qquad h = \emptyset}{s, h \vDash y := e \rightsquigarrow s[y \to n]} \text{ assign} \qquad \frac{}{s, h \vDash \top \rightsquigarrow s} \text{ top}$$

$$\frac{\neg \exists s' : s, h \vDash P \rightsquigarrow s'}{s, h \vDash \neg P \rightsquigarrow s} \text{ negation} \qquad \frac{s, h \vDash P \rightsquigarrow s'}{s, h \vDash P} \text{ assertion}$$

Figure 4. Semantics of our assertion language



	Check assertion validity	Add/remove assertion footprint	Hash module footprint
(1)	X (pre)	X	
(2)		X	X
(3)	X (post)	X	X
(4)		X	

Figure 5. Actions performed for all types of boundary transitions.

takes as input a verified C module with VeriFast annotations (including annotated function prototypes for any function of the context called from the verified module), and outputs a hardened version of the module. The translator reuses significant parts of the existing VeriFast codebase, such as its lexer, parser and typechecker. Although VeriFast's license prevents us from releasing the source code at this time, a binary version of the translator is available online[1]. The translator has been approved by the POPL Artifact Evaluation Committee.

In the sections below, we describe the results of measuring the performance impact of the inserted runtime checks versus the verified module without any runtime checks. We ran micro and macro benchmarks on a standard desktop system, without a protected module architecture, in order to quantify the overhead of *just* the runtime checks, and we discuss the additional overhead introduced by a PMA in Section 7.3. All benchmarks were compiled with GCC 4.8.2, using optimization level 3, and were executed on a system with a 3.10 GHz Intel Core i5-2400 CPU with 8 GiB of RAM, running Ubuntu 14.04. The hash function used to calculate the hash over the footprint when performing an outcall is BLAKE2b [19].

7.1 Micro benchmarks

Since our transformations introduce checks at the *boundary* between the verified and unverified part, there will be a performance overhead when crossing the verified-unverified boundary. During a boundary check, up to three actions are performed: (1) checking whether the assertion (pre- or postcondition) holds, (2) adding or removing the assertion's footprint from the footprint description maintained by the module, and (3) hashing the memory in the module's footprint description. Figure 5 shows which actions are performed for each kind of boundary transition. We measured the contribution of each of these factors using two micro benchmarks based on simple data structures, similar to those used in [29].

The first micro benchmark is a verified module that takes as input a linked list of integers and sorts it using insertion sort. The second micro benchmark is another verified module that does an in-order traversal of a binary search tree to produce a sorted linked list. Both modules have been verified for memory safety (i.e., not for full functional correctness). The entry point signatures of these two modules are as follows.

```
struct list_node* insertion_sort(struct list_node* l);
  req list_pred(l);
  ens list_pred(result);

struct list_node* bst_to_list(struct bst_node* bst);
  req bst_pred(bst, ?v);
  ens bst_pred(bst, v) &*& list_pred(result);
```

Figure 6 shows the distribution of execution time over the different actions performed by the runtime checks for these benchmarks, for input lengths of 10^1, 10^2, 10^3 and 10^4 elements. The number

[1] `https://distrinet.cs.kuleuven.be/software/sound-verification`

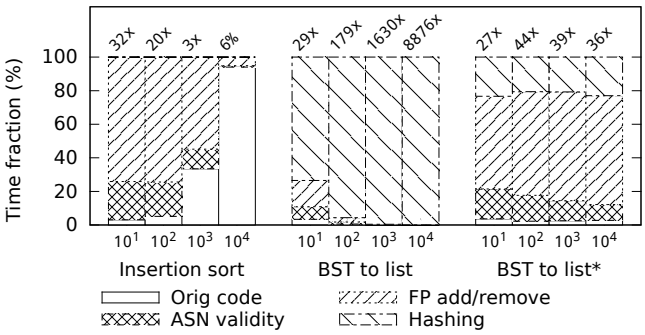


Figure 6. The execution time distribution over the different runtime checking actions for our micro benchmarks, for different input lengths. The numbers above the bars indicate the total execution time overhead in comparison to the unhardened code.

above each bar indicates the total overhead in comparison to the unhardened code.

The left bar chart shows that, for small input sizes, the insertion sort module spends significant time modifying the footprint description and checking assertion validity. As the input size increases however, the relative overhead due to these actions drops to the point where it becomes insignificant. This is because modifying the footprint and checking assertion validity are $\mathcal{O}(n)$ operations that are only performed when entering or exiting the module, and insertion sort is a $\mathcal{O}(n^2)$ algorithm. Hence, the time spent doing useful calculations inside the module increases faster than the time spent performing runtime checks. No time is spent on hashing, because the benchmark does not make any outcalls.

The middle bar chart shows that the BST module spends almost all of its time hashing its footprint, resulting in a huge performance overhead that increases with increasing input size. This is because the benchmark performs an outcall to `malloc` for each node of the input BST that it visits: the memory for the output list is allocated piece-by-piece while traversing the input. Because hashing the module footprint is an $\mathcal{O}(n)$ operation and it is performed n times, we have an $\mathcal{O}(n^2)$ hashing overhead. Since the BST to list algorithm itself is only $\mathcal{O}(n)$, the hashing overhead quickly dominates the execution time. It is however possible to reduce the hashing overhead to $\mathcal{O}(n)$ by using a slightly modified algorithm that first calculates the size of input BST and then allocates memory for the entire output list with a single `malloc` call. This will cause the module footprint to be hashed only once, instead of n times. The performance overhead of this algorithm is shown in the right bar chart of Figure 6. While the relative overhead is still significant, it now remains constant with increasing input size. This shows that the choice of boundary between verified and unverified code, and the number of times this boundary is crossed, can have a large impact on performance.

7.2 Macro benchmarks

While the micro benchmarks from Section 7.1 show how the execution time overhead is distributed over the different actions performed during a runtime check, they do not show the major advantage of our approach: the fact that there is no performance impact on code running completely in the verified or in the unverified part but not transitioning between the two. To show this effect and to assess the real-world feasibility of our approach, we have constructed three realistic macro benchmarks in which we verify and harden a small, security-critical part of an application, but leave the bulk of the application unverified. We measured both the execution time and memory overhead for these macro benchmarks.

7.2.1 Apache httpd modules

The first two macro benchmarks are modified Apache httpd authentication modules, which are used by the web server for verifying user credentials (for instance as part of HTTP Basic Authentication). The first Apache benchmark is based on the `mod_authn_anon` module, and uses a single pair of valid username/password credentials hardcoded in memory. The other benchmark is based on `mod_authn_file`, which reads the list of valid credentials from a file on disk. Both modules provide a single entry point function that takes client credentials (sent to the web server by the browser) as input and returns an integer indicating whether or not they are valid. The signatures of these functions are shown below.

```
int check_password_mem(char *u, char *p);
  req string(u, ?user) &*& string(p, ?pass);
  ens string(u, user) &*& string(p, pass) &*&
      result == 1 ?
        user == "username" &*& pass == "secret"
      :
        result == 0 ? true : result == 2;

int check_password_file(char *u, char *p);
  req string(u, ?user) &*& string(p, ?pass);
  ens string(u, user) &*& string(p, pass);
```

The modules' code consists mainly of a number of outcalls to various I/O and string processing functions of the standard library. In particular, the memory-based module performs 2 such outcalls per HTTP request, while the file-based module performs 34. As can be seen from the signatures above, the memory-based module has been verified for full functional correctness, while the file-based module has only been verified for memory safety, but this makes no difference at runtime. The path of the valid credentials file has been hardcoded in the source of the file-based module.

We set up the pre-forked version of Apache httpd 2.4.7 to serve the default WordPress 3.9.1 sample website with a MySQL 5.5.37 database backend. We used the Apache HTTP server benchmarking tool `ab` to measure the time required to perform 5,000 HTTP requests using 10 concurrent client threads. The client and server were executed on the same host to eliminate any network bottlenecks, and we made sure the web server did not use any form of credential caching. The memory overhead was measured by comparing the peak resident set size of the modules.

The first two rows of Table 1 show the results for the Apache benchmarks. The execution time overhead is low, averaging at 0.68% and 3.74% over three benchmarking runs for the memory-based and file-based module respectively. The memory overhead is also low, averaging at 0.08% and 6.60%. The difference in overhead between the two modules is due to the different number of outcalls they perform and because the file-based module needs a relatively large buffer for reading lines from the password file. This buffer is part of the module's footprint and hence needs to be described by the footprint description and hashed when making outcalls.

7.2.2 NetKit FTP daemon

The NetKit FTP daemon is an FTP daemon shipped with many current Linux distributions. It contains a `checkuser` function that is used to determine whether the names of users trying to log in appear in the `/etc/ftpusers` file of blocked users. We have verified this function and have moved it into a separate module, which we then hardened with our prototype translator. The signature of this function is shown below.

```
int checkuser(char *fname, char *name);
  req string(fname, ?fn) &*& string(name, ?n);
  ens string(fname, fn) &*& string(name, n);
```

The implementation of this function is quite similar to the `mod_authn_file` Apache module, performing 30 outcalls to various I/O and string processing functions per FTP session. The benchmark consists of performing 500 FTP sessions using 10 concurrent client threads, where each session consists of a user logging in, downloading a 1 KiB file and then disconnecting again.

The third row of Table 1 shows the results obtained by taking the average of three benchmarking runs. Both the execution time and memory overhead are again low, confirming our claim that real-world applications consisting mainly of unverified code plus a small hardened module, incur only a small performance overhead.

7.3 PMA overhead

As explained in Section 3, our runtime checks assume a control-flow safe execution model, which we propose to achieve using a fully abstract compilation [2, 31] of the hardened source code to a PMA. Since the micro and macro benchmarks described above were performed on a standard desktop system without a PMA, their results do not yet represent the overhead of our full end-to-end approach. While recent developments [20] indicate low-overhead hardware-based PMA platforms will be available for commodity desktop systems in the near future, the currently available PMAs for desktop systems are still in an experimental state, which prohibits us from running our benchmarks on top of them.

In order to still quantify the overhead of a fully abstract compiler and a PMA, we developed a benchmark to be run on Sancus [30], which is a fully-functional PMA for low-end networked microcontrollers. The Sancus prototype consists of a fully abstract compiler towards a small PMA-enabled 16-bit microcontroller (based on the TI MSP430) featuring 48 KiB of ROM and 10 KiB of RAM. The benchmark consists of a hardened module that provides a function for calculating the median of a linked list of integers, similar to the code example given in Section 2. The function's precondition asserts that the list is a valid non-empty linked list and its body performs three outcalls: one to copy the list, one to sort the copy and one to free the copy before returning. The hash function used for this benchmark was SHA-256. The results indicate the overhead of Sancus (both the compiler and the platform) is below 1%.

7.4 Reducing hashing overhead

The micro benchmarks show that considerable time can be spent hashing the module's footprint. One way of reducing this overhead is to do away with hashing and instead copy the entire module footprint contents to a secure location in memory (e.g., the PMA's private memory region) when making an outcall and to check the footprint against this copy on return. Experiments show that this gives a performance benefit of between 0% and 20% in comparison with hashing, but it obviously requires much more memory.

Another potential performance issue is that, as the verified codebase of an application grows, the size of the hardened module's footprint grows as well, which means more data must be hashed on each boundary transition. However, as the verified codebase grows, the part of the data that is used *exclusively* by the verified part of the application is likely to grow as well. Hence, this data can be placed in private memory, where it can be accessed only by the hardened module and hence need not be hashed on boundary transitions.

An interesting way to solve both issues is by taking advantage of hardware page protection support to reduce the amount of data needing to be hashed on boundary transitions. If an entire memory page is part of the module's footprint, it can be marked read-only in hardware before making an outcall and be reverted to read-write access on return. However, memory pages are typically at least 4 KiB in size, making this approach too coarse-grained to be used directly. A hybrid approach where pages that are completely in the footprint description are set read-only and the rest of the footprint is hashed or copied, is viable, but we consider it out of scope for this paper.

	Execution time (s)			Peak resident set size (KiB)		
	unhardened	hardened	overhead	unhardened	hardened	overhead
mod_authn_anon	33.164	33.388	0.224 (0.68)%	33,356	33,384	28 (0.08%)
mod_authn_file	33.554	34.809	1.255 (3.74)%	33,324	35,524	2,200 (6.60%)
ftpd	23.193	23.242	0.049 (0.21)%	952	976	24 (2.52%)

Table 1. The macro benchmarks show a low real-world performance overhead in terms of execution time and memory consumption.

7.5 Summary

Our micro benchmarks show that the performance overhead of the runtime checks can be significant if there is little computation in- or outside the verified module, compared to the computation required for the boundary checks. Most of this overhead is due to hashing the module's footprint and adding/removing memory regions to/from the footprint description. Nevertheless, the macro benchmarks show that this overhead becomes negligible once more computation is performed in the unverified context. Hence, when developing modules to be verified and hardened, it is critical that the boundary between verified and unverified code is chosen wisely. That is, developers should try to minimize the number of verified/unverified boundary crosses in order to minimize the performance overhead. Although we could not run our full set of benchmarks on a PMA-enabled system, a separate benchmark performed on Sancus indicates the platform overhead is negligible. These results demonstrate the practical feasibility of our approach.

8. Related work

Separation logic-based formal verification ensures memory safety, which can be considered one of its main advantages for memory unsafe languages such as C. There are however many other notable solutions for making C memory safe, such as Safe-C [4], CCured [28] and Cyclone [23]. These systems rely on a combination of type system extensions, static analyses and runtime checks to ensure memory safety, but make no attempt at providing correctness guarantees beyond that. Furthermore, these solutions protect against input-providing attackers, while we protect against more powerful in-code attackers (i.e., attackers that have already gained the ability to execute code in the unverified part).

The idea that software modules should specify contracts in the form of pre- and post-conditions was popularized by Meyer [27] in the programming language Eiffel. Such contracts can then be checked statically or dynamically, and there is a huge amount of literature both on static and on dynamic checking of contracts. Some notable examples include the Java Modeling Language (JML) based tools [7], and .NET Contracts [5].

We rely on fully abstract secure compilation for providing a control-flow safe execution platform and ensuring the soundness of our runtime checks in the presence of code injection attacks. Full abstraction was pioneered by Abadi [1], and has recently been used as a basis for secure compilation to machine code [2, 31] and JavaScript [17]. A related approach is that of TS* [36], a gradually-typed subset of JavaScript that ensures type-safety even when interacting with an untrusted JavaScript context. Although the techniques used in TS* are different from our approach, this work shares our goal of providing a robust foundation for security-sensitive code, while still allowing interaction with an untrusted environment. In the remainder of this section, we limit our attention to the most relevant and closely related works.

Our approach combines modular static verification with runtime checking, to achieve a non-trivial soundness property in the context of an unsafe programming language. The approach is based on separation logic [33] so that there is a clear notion of memory ownership and we can compute the footprint (i.e., the owned region of memory) of a module and take a snapshot of that region's contents. For our implementation and experiments we have used the VeriFast [21, 22] separation logic-based assertion language and static program verification tool for C and Java. Other separation logic-based program verifiers include Smallfoot [6], JStar [13], HIP [9], and Space Invader/Infer [8]. Another notable modular static verification tool for C programs is VCC [10]. However, instead of separation logic, it uses a verification logic that is heavily based on ghost variables, so it is not clear how one would generate runtime checks for module specifications written in VCC's annotation language.

Runtime checking of separation logic assertions is known to be challenging because of the frame rule. A related approach is that of Nguyen et al. [29]. Although some of the techniques used in their approach are similar to ours (e.g., tracking footprints and splitting predicate parameters into input and output parameters), their objective is different from ours. Their runtime checker aims to stay as close to the standard separation logic semantics as possible, while our approach only aims to ensure that no failures can occur in verified code. We can hence allow unverified code to read arbitrary memory, which is not allowed under standard separation logic. Nguyen et al. use a heap coloring technique and runtime checks at every method invocation and field access in unverified code to check framing. This introduces a large performance overhead (on the order of $10,000\times$ if all necessary checks are done) that increases as the size of the unverified code grows. As shown in Section 7, the relative performance impact of our approach decreases with a larger unverified codebase. Also, since the implementation of Nguyen et al. needs to instrument unverified code, the entire codebase must be recompiled, whereas we only need access to the verified module. Finally, the implementation of Nguyen et al. is for Java, so they do not address the complications related to the lack of memory safety of C.

Another related approach is that of Yarra [34], in which runtime checks are used to protect C programs from non-control data attacks. Developers must annotate *critical data structures* with special type declarations, from which point on they should only access those data structures using those special types. In its *whole program protection mode*, runtime checks are inserted throughout the entire codebase to detect illegal accesses to the critical data structures, causing a large performance overhead. In its *library protection mode* however, only the memory accesses of a small core of the application (loosely corresponding to the verified module of our approach) are instrumented. Critical memory writes in the core are modified to maintain a shadow copy of critical objects on separate memory pages, which are made read-only using hardware page protections before calling untrusted code. Critical memory reads in the core are instrumented to check consistency of both copies, thereby detecting unauthorized writes to critical objects from untrusted code. The library protection mode is similar to how we enforce the separation logic frame rule, in the sense that critical regions of memory are integrity protected when calling untrusted code. Our solution provides stronger guarantees than Yarra, since we ensure validity of arbitrary separation logic assertions, instead

of only data structure integrity. Also, Yarra does not prevent untrusted code from disabling the shadow page protections, making it vulnerable to code-injection attacks in the unprotected part. Finally, although the performance cost of Yarra's library protection mode is low, it grows with both the number of boundary crossings *and* the number of reads and writes to critical data in the core part of the application.

Kosmatov et al. [25] described the runtime checking of E-ACSL annotations for C programs, in the context of the Frama-C platform. E-ACSL is an executable subset of ACSL, a behavioral specification language for C programs. Both function contracts and in-body annotations can be specified and can be translated into runtime checks by the E-ACSL2C translator. In order to perform such runtime checks, each memory allocation, deallocation and variable assignment is instrumented to record information about the modified region of memory into a dedicated *data store*. This store hence contains a copy of the program's data and some meta data about it. The runtime pre, post and in-body annotation checks query the store in order to determine the annotations' validity. Although the approach mentions the use of static analyses to statically discard some of the runtime checks, there is no notion of a verified and an unverified part. Hence, the entire program must be instrumented for the checks to be sound and complete. This results in a high overall performance cost, ranging from $13\times$ to $800\times$.

The problem of checking contracts at the boundary between statically checked modules and unchecked modules has also been studied extensively in higher-order programming languages. Findler and Felleisen pioneered this line of work and proposed higher-order contracts [15], which have been implemented in the Racket programming language [16]. The main challenge addressed is that of function values passed over the boundary. Compliance of such function values with their specified contract is generally undecidable. But it can be handled by wrapping the function with a wrapper that will check the contract of the function value at the point where the function is called. This is similar to how we handle function pointers: the corresponding contract is checked when the function is called. One concern that has received extensive attention is the proper assignment of *blame* once a contract violation is detected [12, 18]. While this line of research shares our goal of safely composing a statically checked module with an unchecked module, the issues of higher order contracts and blame assignment are largely orthogonal to the problems we address in this paper.

9. Conclusion

Separation logic-based verification of C code is a powerful technique for guaranteeing the absence of code failures. However, verifying large programs is difficult and requires significant expertise and developer effort. Modular verification tools support partial verification, where only the most critical modules are verified, and where over time more and more modules get verified. Unfortunately, this kind of partial verification gives only limited guarantees at runtime. Bugs in the unverified part of the program can also impact the state of the *verified* part, and hence might trigger failures in verified modules.

We have proposed a way to transform and compile partially verified programs such that the runtime guarantees are significantly better, without imposing severe performance penalties. After our code transformations, no failures can ever occur in the verified module; if a bug is triggered in the unverified part of the program, this is detected before it can impact the state or control flow of the verified module. This is useful for testing, as it detects bugs faster, and for security as it can guarantee verified properties of modules even in the presence of code injection attacks against the unverified part of the program.

Acknowledgments

We thank Greta Yorsh and our anonymous reviewers for their valuable comments and suggestions that have improved the quality of this paper. This work has been supported in part by the Intel Lab's University Research Office, and by the Research Fund KU Leuven. Pieter Agten holds a PhD fellowship of the Research Foundation - Flanders (FWO).

References

[1] M. Abadi. Protection in programming-language translations. In *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, ICALP '98, pages 868–883, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-64781-3.

[2] P. Agten, R. Strackx, B. Jacobs, and F. Piessens. Secure compilation to modern processors. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 171–185, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4718-3. . URL http://dx.doi.org/10.1109/CSF.2012.12.

[3] P. Agten, B. Bart Jacobs, and F. Piessens. Sound modular verification of C code executing in an unverified context: extended version. Technical Report CW 676, KU Leuven, 2014. URL http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW676.abs.html.

[4] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. . URL http://doi.acm.org/10.1145/178243.178446.

[5] M. Barnett and W. Schulte. Runtime verification of .net contracts. *J. Syst. Softw.*, 65(3):199–208, Mar. 2003. ISSN 0164-1212. . URL http://dx.doi.org/10.1016/S0164-1212(02)00041-9.

[6] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 115–137, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-36749-7, 978-3-540-36749-9. . URL http://dx.doi.org/10.1007/11804192_6.

[7] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005. ISSN 1433-2779. . URL http://dx.doi.org/10.1007/s10009-004-0167-4.

[8] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 259–274, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-10671-2. . URL http://dx.doi.org/10.1007/978-3-642-10672-9_19.

[9] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, Aug. 2012. ISSN 0167-6423. . URL http://dx.doi.org/10.1016/j.scico.2010.07.004.

[10] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03358-2. . URL http://dx.doi.org/10.1007/978-3-642-03359-9_2.

[11] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *31st International Conference on Software Engineering, ICSE 2009*, pages 429–430, May 2009. .

[12] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '11, pages 215–226, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. . URL http://doi.acm.org/10.1145/1926385.1926410.

[13] D. Distefano and M. J. Parkinson J. jstar: Towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 213–226, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-215-3. . URL http://doi.acm.org/10.1145/1449764.1449782.

[14] U. Erlingsson. Low-level software security: Attacks and defenses. In A. Aldini and R. Gorrieri, editors, *Foundations of Security Analysis and Design IV*, pages 92–134. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3-540-74809-1, 978-3-540-74809-0. URL http://dl.acm.org/citation.cfm?id=1793914.1793919.

[15] R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 1–15, New York, NY, USA, 2001. ACM. ISBN 1-58113-335-9. . URL http://doi.acm.org/10.1145/504282.504283.

[16] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010. http://racket-lang.org/tr1/.

[17] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 371–384, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1832-7. . URL http://doi.acm.org/10.1145/2429069.2429114.

[18] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 353–364, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. . URL http://doi.acm.org/10.1145/1706299.1706341.

[19] J. Guo, P. Karpman, I. Nikolic, L. Wang, and S. Wu. Analysis of blake2. Cryptology ePrint Archive, Report 2013/467, 2013. http://eprint.iacr.org/.

[20] Intel Corporation. Intel software guard extensions, 2013. URL http://software.intel.com/en-us/intel-isa-extensions#pid-19539-1495.

[21] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 271–282, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. . URL http://doi.acm.org/10.1145/1926385.1926417.

[22] B. Jacobs, J. Smans, and F. Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian Conference on Programming Languages and Systems*, APLAS'10, 2010.

[23] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference*, ATEC '02, pages 275–288, Berkeley, CA, USA, 2002. USENIX Association. ISBN 1-880446-00-6. URL http://dl.acm.org/citation.cfm?id=647057.713871.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. . URL http://doi.acm.org/10.1145/1629575.1629596.

[25] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, pages 167–182, 2013. . URL http://dx.doi.org/10.1007/978-3-642-40787-1_10.

[26] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for tcb minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 315–328, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-013-5. . URL http://doi.acm.org/10.1145/1352592.1352625.

[27] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992. ISSN 0018-9162. . URL http://dx.doi.org/10.1109/2.161279.

[28] G. C. Necula, S. McPeak, and W. Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 128–139, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9. . URL http://doi.acm.org/10.1145/503272.503286.

[29] H. H. Nguyen, V. Kuncak, and W.-N. Chin. Runtime checking for separation logic. In *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'08, pages 203–217, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78162-5, 978-3-540-78162-2. URL http://dl.acm.org/citation.cfm?id=1787526.1787545.

[30] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 479–494, Berkeley, CA, USA, 2013. USENIX Association. ISBN 978-1-931971-03-4. URL http://dl.acm.org/citation.cfm?id=2534766.2534808.

[31] M. Patrignani, P. Agten, R. Strackx, B. Jacobs, D. Clarke, and F. Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, accepted for publication in ACM TOPLAS.

[32] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82(1):77–97, Mar. 2014. URL https://lirias.kuleuven.be/handle/123456789/388689.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1483-9. URL http://dl.acm.org/citation.cfm?id=645683.664578.

[34] C. Schlesinger, K. Pattabiraman, N. Swamy, D. Walker, and B. Zorn. Modular protections against non-control data attacks. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, pages 131–145, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4365-9. . URL http://dx.doi.org/10.1109/CSF.2011.16.

[35] R. Strackx and F. Piessens. Fides: Selectively hardening software application components against kernel-level or process-level malware. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 2–13, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. . URL http://doi.acm.org/10.1145/2382196.2382200.

[36] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P.-Y. Strub, and G. Bierman. Gradual typing embedded securely in javascript. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 425–437, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. .

[37] A. Vasudevan, S. Chaki, L. Jia, J. McCune, J. Newsome, and A. Datta. Design, implementation and verification of an extensible and modular hypervisor framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 430–444, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4977-4. . URL http://dx.doi.org/10.1109/SP.2013.36.