Lightweight Affine Static Capabilities

Brian Mastenbrook and Kevin Marth

AirStash
{brian,kevin.marth}@airstash.com

Project

Abstract. Kiselyov and Shan (2007a) introduced the concept of *lightweight static capabilities*, which provide a method of simulating dependent types in a language with higher-ranked or existential types. We adapt their work to the Rust type system and extend it with *affine capabilities*, which are able to model resource usage. Motivated by a new implementation of the Transport Layer Security (TLS) protocol, we show that affine static capabilities can guarantee properties that are relevant to safe implementation of TLS.

1 Motivation

Transport Layer Security is the backbone of Internet security, providing authenticated encryption for HTTP, SMTP, chat, and numerous other protocols. TLS implementations have in recent years been found to be subject to a number of vulnerabilities which can erode the security of the connection and also of the client or server itself. Three major classes of attacks against TLS are:

- Memory safety vulnerabilities such as "heartbleed", an out-of-range access error that could be exploited to leak private keys (Carvalho et al. 2014)
- Invalid state attacks, where malicious clients or servers can cause a defective implementation to skip states necessary to authentication (Beurdouche et al. 2015)
- Timing attacks, where variations in the number of cryptographic operations performed can be used to leak information about key material (Fardan and Paterson 2013)

TLS is typically provided by the operating system (SChannel on Windows, Secure Transport on Windows) or by a Linux or BSD system (usually, OpenSSL and derivatives). Each of these is implemented in C or C++, which affords the programmer few tools with which to verify the correctness of their programs. miTLS is an implementation of TLS implemented in the dependently-typed F* [1] language which has been verified with respect to the memory safety and state-machine vulnerabilities mentioned (Bhargavan et al. 2013). nqsb (Kaloper-Meršinjak et al. 2015) provides an implementation of TLS in OCaml which relies on the safety of the OCaml runtime against out-of-range errors and uses an explicit type-level representation of protocol states. We

[1] As originally published, miTLS was implemented in F# and specified in F7; however, the newest versions have been published in F*.

build on the experience of these implementation efforts in our work. Both of these implementations introduce a new timing side-channel to TLS through their reliance on a garbage-collected runtime, and due to the dependence on this runtime are also not suitable as a replacement for existing C implementations in all cases.

Our implementation of TLS in Rust, moatTLS (Mitigating Oracles with Affine Types), provides some level of statically guaranteed safety with respect to all three mentioned classes of attacks, and does so through a single mechanism, the use of *lightweight affine static capabilities*. Additionally, Rust's type system enables an implementation of TLS which performs no dynamic memory allocation after connection setup, reducing the possibility of timing side-channels arising from allocation or garbage collection. The implementation is delivered as a purely functional library which is embeddable in either a functional or imperative program. All I/O and other side-effecting operations are represented as explicit types which are returned from the library when appropriate. As the library neither allocates nor frees memory or resources of any kind, it is constrained by the Rust type system to never return a closure [2]. This prevents the implementation from being structured as a monadic or trampolined CPS implementation which captures its environment when I/O is needed, and necessitates the use of explicit, appropriately-typed state objects which are returned to the caller.

TLS is composed of a nested state machine (Beurdouche et al. 2015) in which a connection-level protocol is encapsulated in a record layer which multiplexes several kinds of content; the record layer itself is encapsulated in a stream-oriented transport layer (such as TCP). We represent this in our implementation in three layers:

- The connection layer receives data from the underlying transport, and sends data on that transport. Unlike the layers above, it has no choice about the amount of data it receives from the underlying transport, and must parcel that data out to the record layer as needed.
- The record layer requests data from the connection layer and handles the internal record-level fragmentation imposed by TLS for the protocol layer.
- The protocol layer handles the connection setup state machine, which is composed of a multi-part handshake protocol followed by a transition to encrypted and authenticated data transport.

Within the record layer, the implementation must handle a potentially unbounded amount of protocol-layer traffic during the connection setup phase, as well as an unbounded amount of application data once the connection has been negotiated.

2 Description

We build on the work of Kiselyov and Shan (2007a) in describing *lightweight static capabilities* in OCaml, and extend the concept through the use of affine types in Rust. Kiselyov and Shan describe their work as a style with three ingredients:

[2] In the current version of Rust (1.8) there is no way to name the type of a closure, and thus no way to return an unboxed closure. Putting the closure in a box allows the closure to be treated as a *trait object* and the specific type of the closure's environment to be erased.

- *A compact kernel of trust that is specific to the problem domain*
- *Unique names (capabilities) that confer rights and certify properties, so as to extend the trust from the kernel to the rest of the application.*
- *Static (type) proxies for dynamic values.*

This style is achieved through the use of higher-ranked polymorphism to introduce *brands* as type eigenvariables. [3] These brand types are associated with a branded resource in the trusted kernel, such as an array type, via a phantom type and are thus uninstantiated.

Recall that our TLS implementation is pure (returning to the library user whenever I/O is needed), and due to restrictions on the use of closure types in Rust, cannot be structured as a monadic or trampolined-CPS style program. This structure makes this implementation of capabilities unworkable, as the return type would need to have one type parameter for each capability object created during the execution of the library. Capabilities in our implementation are very fine grained, and may correspond to a single field within a given TLS record; additionally, larger capabilities (such as the capability to read the entire ClientHello message in a TLS handshake) must survive across multiple state transitions. The only way for introduced brand types to be hidden from the caller of the library is if they are captured in a closure, which is not possible in a non-allocating Rust implementation.

In order to solve this challenge, we present a method of parameterizing the connection state with a single brand type, and from that deriving a family of derived brand types. In order to handle circular state transitions inherent in the TLS record protocol which necessitate the creation of an unbounded number of capabilities during the lifetime of a single connection, we introduce the concept of an affine *mark* which is owned by a capability and may be safely recycled when the capability is no longer required. In the lightweight affine static capability approach, we have the following types:

- *Brands* are type eigenvariables used as the certificate of a property.
- *Marks* are uniquely instantiated, affine values parameterized by a brand.
- *Capabilities* are affine objects which certify properties, and are parameterized by a brand and own a mark parameterized by that brand.
- *Branded values* represent properties which have been certified by the kernel, and refer to (and are constrained by the lifetime of) a capability.

Because marks do not own any data (the brand is a phantom parameter), they are zero-sized types, and thus do not impose any representation overhead. Our approach then extends the lightweight static capabilities approach with:

- uniquely instantiated affine mark values parameterized by brand types;
- the ability to derive new unique marks from a mark value, and to recombine derived marks;

[3] p. 14: "A type system that supports either higher-ranked polymorphism or existential types generates a type eigenvariable fresh in the universal introduction or existential elimination rule." Type eigenvariables are used because they are unique, opaque, and propagated by type inference, which allows a suitably expressive typed language to be used as a capability language.

- the ability to recycle mark values on newly created capabilities; and
- the ability to associate other values with a particular marked capability without consuming the mark, and to ensure the associated values do not outlive the capability.

We address the classes of TLS attacks listed above with affine capabilities as follows:

- Memory safety vulnerabilities are mitigated through the use of *slice capabilities* which guarantee that any access to a array ("slice" in Rust terminology) is within bounds through the use of associated indices.
- Invalid state attacks are mitigated through an explicit type-level representation of the protocol state machine, as well as *I/O capabilities* which ensure that each state correctly performs the I/O operations expected of it before any transition to a legal following state
- Timing attacks are mitigated through *cryptographic operation capabilities* which represent a count of operations which have visible timing effect that must be performed before the next state transition

When the brand type is introduced, a single instance of an affine mark type parameterized by the brand is created. Because the mark is parameterized by the brand which is introduced as a type eigenvariable, it is otherwise impossible to create an instance of that mark type. We achieve the introduction of the brand type in Rust using a similar structure to what is used in OCaml.

```
struct Mark<B> {
    brand: PhantomData<B>,
}

trait MarkContinuation<R> {
    fn call<B>(self, Mark<B>) -> R where Self: Sized;
}

fn call_with_mark<R, K>(k: K) -> R
    where K: MarkContinuation<R>
{
    struct IntroducedBrand;
    k.call(Mark { brand: PhantomData::<IntroducedBrand> })
}
```

Several differences between the OCaml approach and the Rust encoding are apparent. In Rust, phantom types must be explicitly represented within a record via the use of the PhantomData type. The MarkContinuation trait is used to encode the type of `call_with_mark`, which would otherwise need to have a higher-ranked type (a feature which Rust does not implement). Users of `call_with_mark` must instantiate the MarkContinuation trait with an appropriate implementation which receives an instance of the mark type. The introduced brand type must likewise be made explicit in the Rust version; scoping rules prevent misuse of the brand type. An example use of `call_with_mark` is given as follows:

```
fn use_mark(i: u8) -> u8 {
    struct K {
        i: u8,
    };
    impl MarkContinuation<u8> for K {
        fn call<B>(self, mark: Mark<B>) -> u8 {
            self.i
        }
    }
    call_with_mark(K { i: i })
}
```

The implementation of the `MarkContinuation` trait for `K` cannot implicitly capture variables from its environment, so explicit capture must be performed.

In Rust, any type which does not implement the Copy trait is affine. Because the mark is an affine value, we can provide operations implemented as ordinary functions that produce derived marks. The derived marks are associated with derived brand types parameterized by the brand of the source mark.

```
fn derive<B>(mark: Mark<B>)
             -> (Mark<DerivedLeft<B>>, Mark<DerivedRight<B>>) {
    (Mark { brand: PhantomData::<DerivedLeft<B>> },
     Mark { brand: PhantomData::<DerivedRight<B>> })
}

fn combine<B>(left: Mark<DerivedLeft<B>>,
              right: Mark<DerivedRight<B>>)
              -> Mark<B> {
    Mark { brand: PhantomData::<B> }
}
```

The `derive` operation consumes the unique instance of the mark type with brand B, and produces two new unique derived marks, one with brand `DerivedLeft<B>` and the other with brand `DerivedRight<B>`. Because the given mark is the unique instance of the mark parameterized by B, it produces two fresh uniquely-branded marks. The `combine` operation consumes the two marks produced by `derive` and reproduces the unique instance of the mark with brand B. If instances of `Mark<DerivedLeft<B>>` and `Mark<DerivedRight<B>>` exist, then no instance of `Mark<B>` exists as it would have been consumed by the `derive` operation that produced the derived marks. These operations allow the safe production of an unbounded number of derived marks.

In order to handle circular state transitions in TLS, capabilities must be destroyed and re-created. Consider the I/O capability that allows the record layer to read a fragment body. Each such capability must be created, used, and then ultimately destroyed when the correct amount of data has been read. On the next fragment, the capability will be re-created with a new fragment size. Affine marks allow this operation to be performed safely: because the I/O capability owns an affine mark, the mark can only be extracted when the capability is destroyed, and only one such capability may be created with a given mark.

The recycle operation is specified by the following trait:

```
pub trait Recycle<B> {
    fn recycle(self) -> Mark<B>;
}
```

Because traits do not have access to the members of their implementations, the recycle operation must be implemented by the capability itself. The signature specifies that the recycle operation consumes the capability and extracts and returns the mark; thus, callers of `recycle` cannot retain capabilities after extracting their marks. Because capabilities own marks, the Rust type system requires that capabilities have affine type as well.

Associating values with capabilities allows arithmetic to be performed on indices into a slice while ensuring that only certified indices can be used to access the slice. To associate a value with a capability, the associated value owns an instance of AssociatedValueMark, which is parametric in the lifetime of the mark associated with the capability as well as the brand type. Lifetimes in Rust are type parameters which track the region of the program during which a borrowed reference is alive. By associating the value with the lifetime of a borrow of the mark, the compiler will ensure that the associated value does not outlive the capability which owns the mark.

```
struct AssociatedValueMark<'a, B : 'a> {
    phantom : PhantomData<&'a Mark<B>>
}

fn associated_value_mark<'a, B: 'a>(mark: &'a Mark<B>)
                                    -> AssociatedValueMark<'a, B> {
    AssociatedValueMark { phantom: PhantomData }
}
```

3 Examples

3.1 Certifying integer properties

The following example presents a certifier for unsigned integers which are below some given value, which is the same technique used to prevent memory safety errors by certifying array indices.

```
struct LessThanCap<B> {
    mark: Mark<B>,
    limit: usize,
}

fn make_less_than_cap<B>(mark: Mark<B>, val: usize)
                         -> LessThanCap<B> {
    LessThanCap {
        mark: mark,
        limit: val,
    }
```

```
}

struct LessThanVal<'a, B: 'a> {
    associated: AssociatedValueMark<'a, B>,
    val: usize,
}

fn certify_is_less_than<B>(val: usize, bc: &LessThanCap<B>)
                           -> Option<LessThanVal<B>> {
    if val < bc.limit {
        Some(LessThanVal {
            associated: associated_value_mark(&bc.mark),
            val: val,
        })
    } else {
        None
    }
}
```

A value of type `LessThanVal<B>` is only created if and only if the integer value it wraps is less than the value contained within the capability itself. Because the lifetime of the AssociatedValueMark instance owned by `LessThanVal<B>` is limited to the lifetime of the mark contained in the `LessThanCap<B>` object, the Rust type system ensures that the certified value never outlives the originating capability. The following example shows use of the capability:

```
fn is_less_than(a: usize, b: usize) -> bool {
    struct K {
        a: usize,
        b: usize,
    };
    impl MarkContinuation<bool> for K {
        fn call<B>(self, mark: Mark<B>) -> bool {
            let l = make_less_than_cap(mark, self.b);
            if let Some(_) = certify_is_less_than(self.a, &l) {
                true
            } else {
                false
            }
        }
    }
    call_with_mark(K { a: a, b: b })
}
```

3.2 Tracking resource usage

Resource-tracking capabilities are used to represent a requirement which must be discharged by the implementation to perform a certain amount of I/O, or to perform a

certain number of cryptographic operations. In the Lucky Thirteen attack (Fardan and Paterson 2013), a timing side-channel resulting from a failure to perform the required number of MAC compression operations acts as a padding oracle. Notably, in (Albrecht and Paterson 2015) an implementation of TLS released after the Lucky Thirteen attack was disclosed was found to be subject to the attack, despite the authors' attempts at mitigation. Using a resource-tracking capability for the underlying cryptographic operation ensures that the time-consuming operation is called at least the correct number of times.

The following example shows a resource-tracking capability which represents a count of operations to be performed; the `consume` function simulates performing an operation.

```
struct ResourceCap<B> {
    mark: Mark<B>,
    count: usize,
}

struct DepletedResourceCap<B> {
    mark: Mark<B>,
}

impl<B> Recycle<B> for DepletedResourceCap<B> {
    fn recycle(self) -> Mark<B> {
        self.mark
    }
}

impl<B> ResourceCap<B> {
    fn make(mark: Mark<B>, val: usize)
            -> Result<ResourceCap<B>, DepletedResourceCap<B>> {
        if val > 0 {
            Ok(ResourceCap {
                mark: mark,
                count: val,
            })
        } else {
            Err(DepletedResourceCap { mark: mark })
        }
    }

    fn consume(self)
               -> Result<ResourceCap<B>, DepletedResourceCap<B>> {
        ResourceCap::make(self.mark, self.count - 1)
    }
}
```

The following example shows a function which consumes all of the resources from a capability, and its use.

```
fn consume_resources<B>(resources: ResourceCap<B>)
                        -> DepletedResourceCap<B> {
    let mut to_consume = resources;
    loop {
        match to_consume.consume() {
            Ok(cap) => {
                to_consume = cap;
            }
            Err(cap) => {
                return cap;
            }
        }
    }
}

fn make_and_consume_resources<B>(mark: Mark<B>, count: usize)
                                 -> Mark<B> {
    match ResourceCap::make(mark, count) {
        Ok(cap) => consume_resources(cap).recycle(),
        Err(cap) => cap.recycle(),
    }
}
```

4 Related Work

Besides Kiselyov and Shan (2007a), there have been several other approaches to the lightweight embedding of dependent-style programming in existing typed functional languages. Eisenberg and Weirich (2012) present a singletons library for Haskell which provides a generator for the necessary boilerplate using Template Haskell. (Brady et al. 2008) present the position that such techniques are limited and full dependently-typed programming is much more expressive; we do not disagree with their position, but find even a limited form of dependence preferable to the alternative given our implementation constraints. Kiselyov and Shan (2007b) also demonstrates the use of lightweight static capabilities for low-level embedded programming.

Tov and Pucella (2011) describe Alms, a language with affine types and affine capabilities which are branded through the use of existential types, and provide a strong theoretical underpinning for their language as well as experience with use of the language in practice. While Rust currently lacks such a theoretical foundation, their work provides strong evidence that affine capabilities are useful and sound in principle.

miTLS (Bhargavan et al. 2013) is the first implementation of TLS with verified security. While the use of refinement types in the specification of miTLS affords much greater ability to statically guarantee the security of the implementation, at least one

property could not be verified in absence of an affine type system, demonstrating that affine types are necessary to the verification of a modular TLS implementation.

5 Acknowledgments

This work was supported in part by DARPA SBIR contract number D15PC00141.

Bibliography

Martin R. Albrecht and Kenneth G. Paterson. Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. Cryptology ePrint Archive, 2015/1129, 2015.

Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A Messy State of the Union: Taming the Composite State Machines of TLS. In *Proc. IEEE Symposium on Security & Privacy 2015*, 2015.

Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with Verified Cryptographic Security. In *Proc. IEEE Symposium on Security & Privacy 2013*, 2013.

Edwin Brady, Christoph Herrmann, and Kevin Hammond. Lightweight Invariants with Full Dependent Types. In *Proc. TFP 2008*, 2008.

Marco Carvalho, Jared DeMott, Richard Ford, and David A. Wheeler. Heartbleed 101. *IEEE Security & Privacy* 12(4), pp. 63–67, 2014.

Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Proc. 2012 Haskell Symposium*, pp. 117–130, 2012.

N. J. Al Fardan and K. G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In *Proc. IEEE Symposium on Security & Privacy 2013*, pp. 526–540, 2013.

David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-quite-so-broken TLS: lessons in re-engineering a security protocol specification and implementation. In *Proc. 24th USENIX Security Symposium*, pp. 223–238, 2015.

Oleg Kiselyov and Chung-chieh Shan. Lightweight Static Capabilities. In *Proc. programming languages meets program verification workshop*, pp. 79–104, 2007a.

Oleg Kiselyov and Chung-chieh Shan. Lightweight static resources. In *Proc. TFP 2007*, 2007b.

Jesse A. Tov and Riccardo Pucella. Practical Affine Types. In *Proc. 38th ACM Symposium on Principles of Programming Languages (POPL'11)*, 2011.