Twizzler-Security
A Capability-Based Security System for Twizzler

BSc Thesis

written by

Surendra Jammishetti

under the supervision of **Owen B. Arden**, and submitted to the

Examinations Board in partial fulfilment of the requirements for the degree of

Computer Engineering B.S.

at the *University of California, Santa Cruz.*

Members of the Thesis Committee:

Dr. Peter Alvaro
Dr. Andi Quinn

Abstract

Traditional operating systems permit data access through the kernel, applying security policy as a part of that pipeline. The Twizzler operating system flips that relationship on its head, focusing on an approach where data access is a first-class citizen, getting rid of the kernel as a middleman. This data-centric approach requires us to rethink how security policy interacts with users and the kernel. In this thesis, I present the design and implementation of core security primitives in Twizzler. Then I evaluate the security model with a basic and advanced scenario, as well as microbenchmarks of core security operations. Lastly, I discuss future work built off this thesis, such as the incorporation of Decentralized Information Flow Control.

Contents

Chapter 1

Introduction

Twizzler is a research operating system focused on new programming paradigms possible via NVM (Non-Volatile Memory)[Bit+20]. It gives programmers direct access to underlying data by removing the kernel from the datapath, which results in huge performance gains, while using NVM to make that data persistent across power cycles. However this reimagining of an operating system leaves many questions for how security is undertaken.

In mainstream operating systems, an omnicient and all-powerful kernel enforces security policy at runtime. It acts as the bodyguard, holding all I/O and data hostage unless the requesting party has the authorization to access some resource. This tight coupling of security policy and access mechanisms works well since any access must be done through the kernel, so why not perform security checks alongside accesses? This coupling gets challenged as soon as one tries to decouple access mechanisms from the kernel, as we see in Twizzler.

1.1 Data-Centric Operating Systems

Twizzler defines itself as a data-centric operating system, meaning it is built upon two key principles [Bit+20]:

1. Providing direct, kernel-free, access to data.
2. Pointers are tied to the data they represent.

These principiles emerge from treating persistent data as a first class citizen. Since NVM removes the necessity of the kernel to serialize and deserialize data from storage devices and memory, it only makes sense for it to be removed from the access path. If applications want to utilize memory as truly persisent, they require a persistent way to access that memory, leading to a notion of persistent pointers.

With the decoupling of the kernel and access methods, we have to rethink how security policy for objects is enforced. While the kernel doesn't manage the connection between applications and data, its still responsible for creating that connection. This provides one area of enforcement, where the kernel can check access rights before granting the application access to the object, and then stay out of the way after. Twizzler programs the MMU, per thread, to grant access rights, allowing for a point of enforcement; more detail can be found in section 4.2. Now we have to build the underlying system that must be enforced.

1.2 Capability Based Security Systems

Capability-based security systems have a rich history in research, and offer an alternative approach to security, in opposition to the Access Control Lists of prevalent OS's [ZL09]. Boiled down, a

capability is a token of authority, holding at minimum some permissions and a unique identifier to which "thing" those permissions apply to [Lev84]. There are some additions we make to this basic defenition in order to apply capabilities in Twizzler, most notablity the addition of a cryptograhic signature. Since capabilities are stored on-disk, the kernel needs a way to ensure the policy its enforcing is coming from a trusted entity. If this weren't the case, it would be trivial for a bad actor to manipulate capabilties, and the kernel would be none-the-wiser as it goes to enforce it. Thus we have this construciton of an cryptographically signed capability, in which the kernel only enforces after it verifies the signature to be authentic.

This paradigm permits kernel-free access of data, while also guaranteeing security by enforcing it right before the point of access through the MMU.

1.3 Our Contributions

In this thesis, I detail the fundamentals of security in the Twizzler operating system, and discuss how I implement and refine some of the high level ideas described in Twizzler [Bit+20] and an early draft of a Twizzler security paper [Bit+]. Additionally, we evaluate these systems inside kernel and user space, using Alice/Bob scenarios and microbenchmarks.

A list of merged PR's to Twizzler:

1. Old Security Port to Main
   - Implementation of SigningKey and VerifyingKey mentioned in seciton 2.
   - Implementation of Capabilities mentioned in section 3.
   - Support to compile twizzler-security for the kernel and userspace
2. Adds creation of SigningKey / Verifying object pairs.
   - Implementation of the keypair objects containing singing and verifying keys, mentioned in section 2.
   - Userspace tests for keypair creation and usage of signing / verifying keys.
3. Security Contexts and Benchmarking
   - Implements Security Contexts for kernel and userspace, as described in section 4.
   - A benchmarking framework for the kernel.
   - Benchmarks for cryptographic operations inside the kernel, and can be viewed in seciton 5.
   - Userspace benchmarks of security policy creation, as shown in section 5.

More details can be found in this Github tracking issue.

Chapter 2

Key Pairs

Key-Pairs in Twizzler are two objects, one containing a signing key, and the other having a verifying key. The signing key is used to form the signature when creating a capability, while the verifying key is used by the kernel to validate a capability before granting access rights. More detail can be found about capability signatures in section 3.1.

They keys are represented as follows:

```
struct Key {
    key: [u8; MAX_KEY_SIZE],
    len: u64,
    scheme: SigningScheme,
}
```

Since the underlying data is just a byte array, the keys themselves are scheme-agnostic, enabling support for multiple cryptograhic schemes, as described in [Bit+20]. This also makes backward compatibility trivial when adding new signing schemes.The keys are stored inside of objects, allowing for persistent or volatile storage depending on object specification, and allows for keys themselves to be treated as any other object and have security policy applied to them.

2.1 Abstraction

Currently we use the Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] to sign capabilities and verify them, but the simplistic data representation allows for any arbitrary algorithm to be used as long as the key can be represented as bytes.

An existing drawback for backward compatibility is the maximum size of the buffer we store the key in. Currently we set the maximum size as 256 bytes, meaning if a future cryptographic signing scheme was to be created with a key size larger than 256 bytes, we would have to drop backwards compatibility. While this can be prevented now by setting the maximum size to something larger, it ends up being tradeoff between possible cryptographic schemes vs the real on-disk cost of larger buffers, something we plan to investigate in future work.

2.2 Compartmentalization

To create an object in twizzler, you specify the ID of a verifying key object so the kernel knows which key to use to verify any capabilities permitting access to the object. Since keys are represented as objects in twizzler, security policy applies on them as well, creating satisfying solutions in regards to key management.

Suppose for instance we have Alice on Twizzler, and all users on Twizzler have a “user-root” keypair that allows for them to create an arbitrary number of objects. Also suppose that access to

this user-root keypair is protected by some login program, where only alice can log in. This means that Alice can create new keypair objects from her user-root keypair. Since all **her** new keypairs originate from **her** original user-root keypair, only **she** can access the keys required to create new signatures allowing permissions into **her** objects. It forms an elegant solution for key management without the involvement of the kernel.

Chapter 3

Capabilities

Capabilities are the atomic unit of security in Twizzler, acting as tokens of protections granted to a process, allowing it to access some object in the ways it describes. Colloquially a capability is defined as permissions and a unique object to which those permissions apply, but in Twizzler we add the signature component to allow the kernel to validate that the security policy was created by an authorized party.

Thus, a Capability is represented as follows:

```
struct Cap {
    target: ObjID,      // Object ID this capability grants access to.
    accessor: ObjID,    // Security context ID in which this capability resides.
    prots: Protections, // Specific access rights this capability grants.
    flags: CapFlags,    // Cryptographic configuration for capability validation.
    gates: Gates,       // Additional constraints on when this capability can be used.
    revocation: Revoc,  // Specifies when this capability is invalid, i.e. expiration.
    sig: Signature,     // The signature.
}
```

3.1 Signature

The signature is what determines the validity of the capability. The only possible signer of some capability is who ever has permissions to read the signing key object, or the kernel itself. The signature is built up of a array with a maximum length and a enum representing what type of cryptographic scheme was used to create it; quite similar to the keys mentioned previously. The fields of the capability are serialized and hashed to form the message that gets signed, and then stored in the signature field. Currently we support Blake3 and Sha256 as hashing algorithms.

3.2 Gates

Gates act as a limited entry point into objects. If a capability has a non-trivial gate, which is made up of an offset field, and a length field, the kernel will read that and ensure that any memory accesses into that object are within the gate bounds. The original Twizzler paper [Bit+20] describes gates as a way to perform IPC, and calls between distinct programs, but in the context of this thesis it is sufficient to think of them as a region of allowed memory access.

3.3 Flags

Currently, flags in capabilities are used to specify which hashing algorithm to use to form a message to be signed. We allow for multiple algorithms to be used to allow for backward capability when newer, more efficient hashing algorithms are created.

The flags inside a capability is a bitmask providing information about distinct feautures of that capability. Currently we only use them to mark what hashing algorithm was used to form the message for the signature, but there's plenty of bits left to use. We hope for future work to develop more expressive ways of using capabilities, i.e. Decentralized Information Flow Control, as specified in 6.1.

Chapter 4

Security Contexts

Security Contexts are objects that threads attach to in-order to inherit the permissions inside the context. The contexts store capabilities, allowing for userspace programs to add capabilities to contexts, and kernel space to efficiently search through them to determine whether a process has the permissions to perform a memory access.

4.1 Base

Since security contexts can be interacted with by the kernel and userspace, there needs to be a consistent definition that both parties can adhere to, which we define. Objects in Twizzler have a notion of a `Base` which defines an arbitrary block of data at the "bottom" of an object that is represented as a type in rust. We define the `Base` for a security context as follows:

```
struct SecCtxBase {
    map: Map<ObjID, Vec<CtxMapItem>>, /// A Map from ObjIDs to possible capabilities.
    masks: Map<ObjID, Mask>,          /// A map from ObjID's to masks.
    global_mask: Protections,         /// Global mask that applies to granted prots.
    flags: SecCtxFlags,               /// Flags specific to this security context.
}
```

4.1.1 Map

The map holds positions to Capabilities relevant to some target object, which the relevant security context implementations for kernel and userspace to parse security context objects. Implicitly, the kernel uses this map for lookup while the user interacts with this map to indicate the insertion, removal, or modification of a capability. The `Map` type here and for `masks` is a flat data-structure, and stores offsets into the object where capabilities can be found for a target object.

4.1.2 Masks

Masks act as a restraint on the permissions this context can provide for some targeted object. This allows for more expressive security policy, such as being able to quickly restrict permissions for an object, without having to remove a capability and recreating one with the dersired restricted permissions.

The global mask is quite similar to the masks mentioned above, except that it operates on permissions granted by the security context as a whole rather than a mask per distinct object id.

4.1.3 Flags

Flags is a bitmap allowing for a Security Context to have different properties. Currently, there is only one value, UNDETACHABLE, marking the security context as a jail of sorts, as once a process attaches to it, it won't be able to detach. This acts as a way to limit the transfer of information if

a thread attaches to a sensitive object. Once a thread attaches to such a context, it is forced to end its execution with the objects that context grants permission to. We also plan to utilize these flags in future works, as described in 6.1.

4.2 Enforcement

All enforcement happens inside the kernel, which has a seperate view into Security Contexts than userspace. The kernel keeps track of all security contexts that threads in Twizzler attach to, instantiating a cache inside each one. Additionally, a thread can attach to multiple security contexts, but can only utilize the permissions granted by one unless they switch [Bit+20]. To manage these threads, the kernel assigns a Security Context Manager, which holds onto security context references that a thread has.

There exists only 1 point of enforcement for security policy if we wish to keep the kernel out of the access path; the creation of the path itself! On page fault, the point in which a process requests the kernel to map an object in is when we have access to the security policy we seek to enforce (the signed capabilities inside the security context), the target object, and most importantly, kernel execution! Its the only time we can program the mmu according to the desired protections, and transfer control of enforcement to the hardware [Bit+20].

Upon page fault, the kernel inspects the target object and identifies the default permissions of that object. Then the kernel checks if the currently active security context for the accessing thread has either cached or capabilities that provide permissions. If default permissions + the active context permissions arent enough to permit the access, the kernel then checks each of the inactive contexts to see if they have any relevant permissions. If there exists such permissions, then the kernel will switch the active context of that process to the previously inactive context where the permission was found. If it fails all of these, then the kernel terminates the process, citing inadequate permissions.

Chapter 5

Results

5.1 Validation

The first test is a basic scenario as a check to make sure the system is behaving as intended, and a more expressive test to demonstrate the flexibility of the model. Eventually, I intend to work with my advisor and peers to form a proof of correctness for the security model, as well as empirical testing to demonstrate its rigidity.

5.1.1 Basic

TBA

5.2 Expressive

TBA

5.3 Micro Benchmarks

Additionally, we have microbenchmarks of core security operations in Twizzler. All benchmarks were run with a Ryzen 5 2600, with Twizzler virtualized in QEMU. Unfortunately I ran out of time to perform benchmarks on bare metal, but hope to find any discrepencies between virtualized and actual performance in future work.

5.3.1 Kernel

The kernel benchmarking framework takes a closure (a block of code we want to benchmark), runs it atleast 100 times, and scales the number of iterations to reach 2 seconds of total runtime, and stores the time it takes for each run. Then it computes the average, and the standard deviation from the timings.

There a couple of things we benchmark inside the kernel, including core cryptographic operations like signature generation and verification, as well as the total time it takes to verify a capability.

Benchmark	Time
Hashing (Sha256)	267.86 ns ±163 ns
Hashing (Blake3)	125.99 ns ±117 ns
Signature Generation (ECDSA)	199.90 $\mu s \pm 9.45 \mu s$
Signature Verification (ECDSA)	342.20 $\mu s \pm 6.28 \mu s$
Capability Verification (ECDSA, Blake3)	343.59 $\mu s \pm 5.32 \mu s$

Table 1: Collection of Kernel Benchmarking Results

We see that signatures are vastly more expensive than hashing, on an order of 10^3, meaning that your choice of hashing algorithm doesn't affect the total time taken for the verification of a capability. It's also important to note that this cost of verifying a capability for access is done on the first-page fault, then the kernel uses caching to store the granted permissions and provides those on subsequent page faults into that object. In the future, I hope to measure the difference between a cached and uncached verification. Secondly, we only measure verification inside kernel space; as discussed in section 3, capability creation only takes place in user space.

5.3.2 UserSpace

Userspace benchmarks were calculated using rust's built in benchmarking tool.

In userspace, we benchmark keypair and capability creation, as these operations are core to creating a security policy.

Benchmark	Time
Capability Creation	347.97 $\mu s \pm 5.78 \mu s$
Keypair Objects Creation	651.69$\mu s \pm 187.90 \mu s$
Security Context Creation	282.10$\mu s \pm 119.90 \mu s$

Table 2: Collection of UserSpace Benchmarking Results

Almost all the time spent in creating a capability is the cryptographic operations used to form its signature, which is why it's in the same ballpark as the signature creation we saw earlier.

The high standard deviation in Keypair objects and Security context creation happens from the unpredictable time it takes for the kernel to create an object on disk. The reason keypairs are almost 2x more expensive since they create two separate objects, one for the signing key, and one for the verifying key.

Chapter 6

Conclusion

In short we provide a general overview of the critical security components for security system in Twizzler, along with implementation details and desgin descisions. The evaluation programs show how security policy can be expressed and verifies that the kernel is enforcing as programmed. Lastly we go over microbenchmarks to show and explain the cost of these operations.

The results affirm our intuition that performance would be greatly improved via caching. The cost of verifying a signature everytime a new page from an object had to be mapped into a process's memory space would be redundant. Additionally, the performance of the kernel verifying signatures is bottlenecked by the performance of the cryptograhpic scheme, meaning its a good plan to allow for the addition of new schemes while allowing for backwards compatibility since adopting a more performant scheme would lead to pure performance gains.

6.1 Future Work

There are a number of things I hope to achieve in future work, listed as follows.

- Perform a cost-benefit analysis between key sizes and performance, trying to optimimze for a future proof key size in order to maximize backwards compatibility.
- Program the kernel to perform access rights checks with a processes secuirty context during a page fault. I was hoping to get this completed before the end of this quarter, but we ran into some bugs and were unable to resolve them in time. Once this is hooked up, we plan to design scenarios that test the degress of expressivity allowed by our secuirty model to ensure it operates as expected.
- Investigate areas of the secuirty model that could be extended to support Decentralized Information Flow Control, inspired by the work done in FLUME [Kro+07].
- Create a onboarding process that allows new students to learn the essentials of the Twizzler operating system, to foster an environment for increased student contributions to the project.
- Clear code documentation so that users wanting to interface with the library have an easier time integrating it with their applications.

6.2 Acknowledgements

I couldn't have done the work for this thesis and for Twizzler if it wasn't for the support I've recieved from my advisor Owen Arden and my technical mentor Daniel Bittman! I owe both of you so much, not just for this thesis but also for how much I've learned in this endeavor. Thanks guys!

Bibliography

[Bit+20] Bittman D, Alvaro P, Mehra P, Long DDE, Miller EL. Twizzler: a Data-Centric OS for Non-Volatile Memory. In:. 2020 USENIX Annual Technical Conference (USENIX ATC 20), USENIX Association; 2020, pp. 65–80.

[ZL09] Zhai G, Li Y. Analysis and Study of Security Mechanisms inside Linux Kernel. In:, 2009, pp. 58–61. https://doi.org/10.1109/SecTech.2008.17.

[Lev84] Levy HM. Capability-Based Computer Systems. USA: Butterworth-Heinemann; 1984.

[Bit+] Bittman D, Alvaro P, Mehra P, Long DDE, Miller EL. A Data Centric Model for OS Security. In:, USENIX Association; n.d., pp. 65–80.

[JMV01] Johnson D, Menezes A, Vanstone S. The Elliptic Curve Digital Signature Algorithm (ECDSA). Int J Inf Secur 2001;1:36–63. https://doi.org/10.1007/s102070100002.

[Kro+07] Krohn M, Yip A, Brodsky M, Cliffer N, Kaashoek MF, Kohler E, et al. Information flow control for standard OS abstractions. SIGOPS Oper Syst Rev 2007;41:321–34. https://doi.org/10.1145/1323293.1294293.