The Sting Framework (SF)

IC3

Follow

--

Listen

Share

v1.0

by Kushal Babel, Nerla Jean-Louis, Mahimna Kelkar, Yunqi Li, Carolina Ortega Perez, Aditya Asgoankar, Sylvain Bellemare, Ari Juels, and Andrew Miller

TLDR

: The Sting Framework

(SF) is a new idea for bolstering the security of systems at risk of information leakage. SF addresses the case where a corrupt service (called a Subversion Service

) arises that enables adversaries to exploit such leakage. SF presumes a player, called an informer

, that wishes to alert the community to the presence of the corrupt service — either as a public service or to claim a bounty. SF enables the informer to generate a publicly verifiable proof

that the corrupt service exists.

SF works by generating an input to the system called a stinger

. The stinger's contents are unknown to everyone, even the informer. The informer can generate a proof of existence of the corrupt service by proving an ability to obtain leaked data about the stinger.

SF harmonizes particularly well with SGX-based systems. SGX can be used to generate stingers and SF can also be used to detect information leakage from SGX-based systems — a particular concern given their documented side-channel vulnerabilities.

We report on a prototype SF system (available [here

](https://github.com/initc3/Sting-Flashbots/)) that we created for the Flashbot SGX builder. It includes a bug bounty application implemented in a smart contract. With the proliferation of SGX into block-building and other applications (e.g., the Signal messaging system), we believe that SF can serve as a key defensive mechanism.

Introduction

In this post, we introduce the Sting Framework

(SF). SF aims to bolster the security of systems that maintain secret state. It's especially useful for applications built using trusted execution environments (TEEs) such as Intel SGX, although it can be used for any system meant to provide confidentiality, whether it uses TEEs, MPC, or something else. For concreteness, our focus in this post is on TEEs.

If an exploit arises, SF helps a user with access to it generate a proof

that the exploit exists. SF-generated proofs can serve as trustworthy warnings to the community about compromised TEE-backed applications. They have other uses as well, such as enabling automated bug-bounty programs.

TEEs are proliferating across industry systems. They help enforceprivate contact discovery in Signal, for instance, and will serve similarly to protect transaction flows in blockchain systems. At the same time, Intel SGX, the most prevalent TEE for multi-user applications, has experienced a number of exploits. TEEs are designed to enforce confidentiality between applications that operate on the same computing device and share resources. That proves to be a tricky proposition, as is coding secure software within the security model enforced by TEEs. So vulnerabilities in TEE-backed applications are a real concern.

SF operates in a setting when a user, called an Informer

, has access to an exploit against a target TEE-backed application. The Informer may have direct knowledge of the exploit or may instead have access to it through a hidden functionality or corrupt service that we call a Subversion Service

. The Informer wishes to make the existence of the exploit public — as a public service or perhaps to collect a bug bounty. SF enables the Informer to generate a publicly verifiable proof that the exploit exists and that it compromises the security of the TEE-backed application.

The term “sting” in SF derives from the notion of a sting operation. The Informer in an SF plays a role analogous to an undercover operative. She poses as a legitimate user of the Subversion Service while amassing proof of its existence and efficacy. To ensure success, the informer’s behavior must appear legitimate to the Subversion Service — just as an informer in a real-life sting operation must pass muster with the criminals against which she is gathering evidence.

SF proofs can serve as credible warnings for communities of users around affected systems. SF proofs are verifiable by smart contracts, so they can also enable automated bug-bounty programs that award cryptocurrency to incentivize informers. For most TEE-backed applications, SF is easy to adopt. It generally requires little or no modification of the application itself.

In this blog post, we detail the main concepts in SF. We also report on a prototype we’ve implemented to protect the Flashbots SGX Builder, a TEE-backed system that orders transactions for blocks in Ethereum.[1]

A recent incident affecting block production in Ethereum offers an excellent cautionary tale and good motivation for use of SF as a safeguard that helps protect block-building infrastructure. The Flashbots MEV-Boost Relay — a service that selects blocks for validators — was supposed to reveal transactions in blocks only after

signing by validators, but had a bug that leaked transactions to validators before

signing. A rogue validator stole some $20 million from arbitrageurs by exploiting the bug. This shows how block-building infrastructure is a prime target for exploits and how leakage can have a grave impact on its security.

SF overview

The key idea in SF is to have an Informer send to the TEE-backed application a special input called a stinger

. An Informer constructs the stinger in a manner so that when she inputs the stinger to the application and concurrently makes use of the exploit, she gains evidence for an SF proof.

Typically, the stinger contains data that is secret, i.e., unknown even to the Informer. Stinger data is in this case sent in encrypted form to the application. If the application were functioning correctly — i.e., if there were no Subversion Service (or other exploit) — the application would keep the data in the stinger secret for at least some period of time. The Informer demonstrates, however, that she can gain access to secret stinger data during a period when it should still be secret. In this way, the Informer demonstrates the existence of an exploit against the application.

Example 1 below illustrates stinger use in SF.

Example 1 above involves use of a stinger generated by a trusted third party (TTP). The existence of a TTP is a strong assumption. Iona could instead, however, realize the TTP using her own TEE. Iona’s TEE would in this case run a special SF application that generates secret stingers and checks the correctness of decrypted stinger data furnished by Iona. In other words, Iona’s TEE would do everything the TTP does in Example 1.

Given such use of a TEE, an SF proof would show either the existence of a successful exploit against the contact-discovery application or

that Iona has broken her own TEE.[2] If Iona can break a TEE, the SF proof still

demonstrates a problem with the contact-discovery application. That’s because a TEE break is a good indication that the application, which itself relies on a TEE, is in jeopardy.

Note:

SF is also useful in settings where a target application isn’t

running in a TEE. In this case, though, one needs to assume that the TEE generating a stinger is secure. Otherwise, the Informer might for example learn the stinger prematurely and create a false proof of existence of a Subversion Service. In Example 1, this would mean that the Informer could frame

Contactr.

As an alternative to use of TEEs, stingers can be generated using secure multi-party computation.

SF security model: Subversion Services

Exploits are often turned into services, typically for profit. Malware-as-a-service, including ransomware-as-a-service and rentable botnets are common ways to monetize exploits against user devices and internet infrastructure. Blockchain systems too have seen systematic monetization of what may be viewed as exploits against users, especially Miner / Maximal Extractable Value (MEV), which now underpins a whole industry.

For this reason, SF is designed mainly for a setting where there's a Subversion Service, as defined above. A Subversion Service may be thought of as a parasitic service or backdoor-as-a-service that weaponizes an exploit against a target TEE-backed application. Some subset of (malicious) users / customers called Subverters

can call the Subversion Service to bypass the security features of the target service.

Take our Contactr example above, for instance. Iona might have access to a Subversion Service, advertised in dark markets by a hacker group, that pierces the privacy of Contactr's contact-discovery database. Alternatively, Iona might be an Contactr whistleblower who has discovered an exploit that Contactr refuses to fix. Iona doesn't want to reveal the exploit itself publicly because an adversary could then use it to attack Contactr. Iona might instead set up her own Subversion Service to prove publicly that the exploit exists

, without explicitly revealing it.

Of course, the operator of a Subversion Service is likely to be aware of the threat posed by SF. The operator could try to protect against SF by having the Subversion Service reject inputs that look like stingers. Returning to our example, a Subversion Service that leaks Contactr's database might filter out contacts with improbable names (e.g., "BlottyBorg…"). By purging potential stingers this way, the Subversion Service operator would make Iona's job harder.

One additional feature of the Subversion Service is its taking as input from a Subverter what we call a tamper instruction t

. A Subversion Service leaks secret data to a Subverter in order to allow it to manipulate the Target Application. The Subversion Service may (or may not) also help the Subverter accomplish this manipulation, in which case it relies on a tamper instruction t

from the Subverter to do so.

One of the main challenges in designing a good SF application is the need to generate stingers that: (1) Appear plausible, i.e., are drawn from a distribution that is indistinguishable from that of ordinary inputs to the target TEE-backed application and (2) Provide strong proof of the existence of an exploit.

While (1) is the harder challenge, happily it isn't insurmountable. In some cases, it is possible to create stingers that are computationally or statistically indistinguishable from ordinary inputs. But even if not, one thing working in our favor is the fact that stingers needn't be perfectly

indistinguishable from ordinary inputs. If a Subversion Service filters out Iona's stinger contacts most of the time, but not always, Iona can keep submitting stinger contacts until one goes through successfully. She may need to pose as a succession of distinct, unconnected users in case, e.g., the Subversion Service blocks accounts that have submitted stingers.

Another related challenge is ensuring that the tamper instruction used by an SF also isn't distinguishable to the Subversion Service as involving the use of an SF, as the Subversion Service might then fail to act on the instruction. The same helpful fact applies to this challenge as well: perfect indistinguishability isn't required to make SF work.

SF protocol details

Figure 1:

Entities in SF

SF involves five different entities, shown in Figure 1:

- A Target Application

, which runs in a TEE.

- A Subversion Service

that leaks information about the Target Application and may also allow for tampering with its state.

- TheInformer

, Iona in Example 1, whose aim is to generate an SF proof, which shows the existence of the Subversion Service.

- An SF Application

, which generates stingers and SF proofs for the Target Application.

- A Verifier

, which consumes and checks SF proofs generated by an informer.

Figure 2:

Protocol steps in SF.

Figure 2 shows the protocol steps in SF. The Informer ① requests and ② receives a stinger s

from the SF Application drawn from a distribution D

that is representative of the inputs submitted by the Informer. In other words, drawing s

from D

ensures that s

is indistinguishable from inputs normally expected from the Informer. The informer ③ feeds s

to the Target Application. (In some protocol variants, e.g., where s

is a plaintext value sent over TLS, it might instead be sent directly to the Target Application.) She ④ obtains leaked data ℓ from the Subversion Service. In response, she may ⑤ submit a tamper instructiont

to the Subversion Service. This is a direction to the Subversion Service to modify the state of the Target Application. (We give an example below in our discussion of SF for block building.) On ⑥ obtaining output o

from the Target Application, the Informer ⑦ feeds both o

and the leaked data ℓ

to the SF Application, which uses them to ⑧ generate an SF Proof Π. The Informer can then ⑨ send Π to the Verifier to prove the existence of the Subversion Service.

SF for block-building

As a concrete example of a TEE-backed Target Application, consider a block-building application for the Ethereum blockchain, specifically the Flashbots SGX Builder prototype. This application ingests encrypted transactions — specifically, sequenced sets of transactions called bundles

, provided by users called searchers

. The application's goal is to assemble the transactions into a block for inclusion on the blockchain. Once it has formed a block B

, the application transfers this block to a validator, i.e., a node that generates blocks for the blockchain.

The role of SGX is to ensure the pre-trade

privacy

of the bundles prior to the commitment of the block. The bundles submitted to the application may for example contain private order flow (or be the result of proprietary strategies executed by arbitrageurs, or may contain first-come-first-serve order flows that do not wish to be unbundled.) A builder that sees a bundle can replicate and, in effect, steal the implementation of an arbitrageur's strategy or learn and exploit knowledge of private order flows.

Bundle privacy is thus an essential security feature of a block-builder. We might therefore consider a Subversion Service that leaks as ℓ

the bundles in the Target Application to a malicious user. The malicious user taking advantage of this information would be able to create their own bundles that yield a profit, for example by front-running the transactions present in the block-builder.

For this particular setting, we don't need to assume that the Subversion Service accepts tamper instructions (although it might). We only need to assume that an Informer can get its own bundle accepted into a given block B

. This is in fact a requirement for a Subversion Service to be useful to a Subverter, and thus a reasonable assumption.

We propose the following SF-Application design for generation of SF proofs. We construct our stinger distribution D

based on the ordinary behavior of a searcher. That is, we start with a searcher algorithm that produces a bundle d

based on the public mempool and any privately known transactions. Additionally we can assume the bundle contains a transaction. T

, signed by the informant — typically a profit-taking transaction. The SF Application generates a stinger s,

consisting of the bundle d

except the transaction T

is generated randomly within the enclave and including random bits e.

After submitting s

to the Subversion Service, the Informer learns and therefore e

. The informer then adds its own bundle d’

into the block B

(possibly using a different identity / address than that used for e)

. The bundle d

’ encodes a commitment to the secret bits e

. For example, if e

has high enough entropy, d’

can include a hash of e

.

The issued block B

contains evidence that the Informer learned the secret bits e

— and thus that the Subversion Service exists. Specifically, the Informer should not have been able to learn e

until after

the block B

was released by the block-builder. The fact that d’

was included in B

shows that the Informer learned e

before B

was released. Thus it provides evidence of a Subversion Service.[3]

The Informer can feed B

to the SF Application, which created and therefore knows s

. As it runs in a TEE, it can then generate an attestation on the Informer’s evidence. This attestation constitutes a proof of the existence of the Subversion Service.[4]

The strength of the proof Π, i.e., its soundness, depends on the number of bits in e

. As long as there’s a single transaction T

in s

that is unknown to the informer, s

will have high entropy. That’s because an ECDSA digital signature — the kind used in Ethereum today — contains a 256-bit random string, often called a nonce. Thus by signing T

within the SF application itself and including it in s

we have sufficient entropy.

Signaturesalso

provide a convenient way to encode a cryptographic commitment to e

in d’

, as required by SF. The informer can simply use such a commitment as the randomness for its own digital signature in d

'. It can then provide a decommitment to the SF Application in step ⑦ of Figure 2 as auxiliary data.[5]

The SF framework can also incorporate a reward smart contract that releases a cryptocurrency reward upon receiving a valid SF proof, which incentivizes participants to report vulnerabilities in TEE-backed block-builder. In principle, verifying TEE-based attestation on-chain is the right way to go. Later on we discuss limitations and workarounds with smart contract tools for this today.

Comment: We can also design an SF application in a stronger adversarial model where the information ℓ

leaked by the Subversion Service does not contain an entire bundle, but only summary information about transactions (e.g., "there is a very large buy transaction for ETH-USDT in this block"). In this case, ℓ

may have low entropy, but by running the SF proof protocol over many rounds, a high-assurance SF proof can still be generated.

Public testnet prototype of block-building SF.

We now describe some remaining details in our SF prototype, following along with the step by step illustration.

We start by describing the on-chain registration process, by which an Informant registers their Sting enclave with the bounty smart contract that will eventually verify the evidence and issue a reward to the Informant. An initiator (potentially an administrator of the Target Application, or an independent third party like a Foundation) deploys a Reward contract on-chain. Benefactors can contribute donations to a reward pool. To register their Sting enclave, an informant runs the Setup function. This generates the enclave's signing key and attestation report in the enclave. The public signing key is included in the report's user report data field. It then sends this report to the Intel Attestation Service (IAS) to get a signature for the report. The report, IAS signature, and the enclave public signing key are posted on chain.

Due to the lack of practical implementation of verifying x509 certificates in Solidity, we instead rely on a multi-sig administrator to check the attestation proofs off-chain, sending an on-chain message to approve them. Our protocol is designed so that the Informant must interact with this contract once, prior to running the Sting protocol. Since anyone could verify the attestations locally, an administrator that approves invalid proofs or ignores valid ones would be clearly visible. Note also that nothing stops us from implementing x509 certificate checking in Solidity, and in fact some projects have made steps along this direction, but we left it out of scope. Other approaches might make use of Oracles or computation on L2 networks.

We implement the SF enclave by adapting a public searcher program in SGX using the Gramine library and taking transactions from a combination of the public mempool and potentially private order flow as inputs. When creating the stinger, the SF Application takes a bundle generated outside of the enclave and a separate unsigned transaction tx1

. The sampling of the nonce for the ECDSA signature e

from U(0, 2

^{256}) to complete the signature is carried out within the enclave. The secret e

is then"sealed" for future use by the TEE. The bundle s

(sting_bundle in the figure) is sent to the SGX builder through the TLS channel for privacy.

The enclave state, including secrets sampled when generating the stinger and the enclave's signing key, are all stored as sealed files using the SGX MRENCLAVE policy. This ensures that only the exact same code can be used to handle and decrypt secret data.

We implement the Subversion Service by explicitly introducing code into the Builder that leaks entire bundles to the informer. In a real world scenario, due to remote attestation, such a change would be visible; the Subversion Service would either need to be obfuscated within the codebase, or else obtained using side channels against the SGX enclave itself.

Figure 3:

Illustration of our demo for Flashbots SGX builder.

1. (not shown in the figure) The administrator deploys and funds the Reward contract.
2. The informant generates a pair of new signing keys in the enclave and prepares attestation-related material. The submitEnclave

function on the Reward contract is invoked to provide the public signing key, attestation proof, and IAS signature. Seal the private signing key. Wait for the approval from the administrator.

1. The contract administrator verifies the attestation-related material provided by the new participants and calls

approveEnclave

if all checks passed.

1. Create stinger in the Sting Enclave. Prepare a bundle and an extra transaction outside of the enclave. The Sting Enclave generates and encodes the commitment of secret e

in tx1. Seal e.

Pass the hash of tx1 to the informant (outside of the Sting enclave) as a backdoor.

1. Append tx1 into the bundle and send the bundle through the TLS channel to the SGX builder.
2. The Subversion Service leaks all the bundles received by the SGX builder to the Informant.
3. The informant identifies the sting_bundle and tx1 using the backdoor. Create tx2 which encoding the commitment to tx1's signature. Make and send the evidence_bundle (including tx2) to the SGX builder.
4. The SGX builder constructs a block containing both the sting_bundle and evidence_bundle and submit it to an Relayer. Once the block is mined, the informant generates merkle proofs for tx1 and tx2,demonstrating their presence in the same block. The informant also passes the binding factor of the commitment, block number and block hash to the verification program running in Sting Enclave.
5. The Sting Enclave verifies the evidence provided by the informant. Once all checks pass, it signs on the block number and block hash with the private signing key and passes the resulting signature to the informant, who invokes the collectReward

function to claim the rewards.

When the informant receives several bundles leaked by the Subversion Service and they have to use the backdoor provided by the SF Application, the hash of tx1

, to locate bundle d

and then tx1

. The informant creates bundle d'

, which contains a new transaction, tx2

, signed by the common account and with ECDSA signature nonce equals to Commit(hash(tx1

.sig), r) where r is a randomly sampled binding factor. We use the Pedersen commitment scheme in our implementation. Subsequently, the Informant sends bundle d'

to the builder with a high enough gas price so that d' would appear in the same block as d

. It is worth noting that without the Subversion Service, the informant knows only the stinger transaction hash but not more details, especially the content of the signature.

In order to prove the success of the Sting protocol, the informant provides evidence to the Sting enclave that tx1

and tx2

are included in the same block, the block is finalized, and that tx2

contains a commitment to tx1

. The verification program runs inside SGX then outputs a proof of the success of the sting protocol in the form of a signed message containing the block number and block hash.

Finally to collect the reward from the contract the informant posts the signed proof generated by the enclave. The contract checks that the enclave's public key was approved, that the signature is correct, and that the block number and block hash in the proof match.

Another example use case

As previously mentioned, SF protocols are relevant for systems beyond block-building. To illustrate this broader applicability, we give another example use case: Signal contact discovery. For details, we rely on Signal's website about contact discovery and their repository for Signal for Android.

Signal is a messaging app notable for its prioritization of security and privacy in their system. For a Signal user to learn which of their contacts also use Signal, the app runs a contact discovery protocol. At a high level, in this protocol, a user

Iona sends a list of their contacts to Signal. The app then compares the Iona's phone contacts list to a list of all of Signal's registered users and computes the intersection of both lists. Finally, Iona receives the result from the intersection and thus learns which of their contacts are Signal users.

In their threat model, Signal establishes that Iona's phone contacts list should remain private from everyone, including Signal's servers. This protects Iona's privacy if the servers were hacked, if a government agency requested access to Signal's servers, among other cases. To achieve this, Signal relies on SGX and ORAM in their servers. These technologies allow the contact discovery protocol to run in a privacy-preserving manner, where the servers can perform the set intersection computation without learning anything from Iona's contact list. Additionally, SGX allows for remote attestation, which enables verification of the code running in the servers; in particular verifying that the code does not leak Iona's data.

This is a real world scenario of the case similar to the one illustrated in Example 1. If Signal's SGX was broken and if there was a Subversion Service which exploits the SGX vulnerability, one can build an SF protocol for an informer Iona as follows. The target application would be Signal, and the target leaked data will be users' contact lists.

Figure 4:

Protocol steps in SF protocol for Signal Contact Discovery.

In Figure 4 above, Iona first requests that the SF application create a stinger (step ①), which in this case would be a new Signal user U

, who has a contact list ℓ

. In ①, Iona also sends a distribution D

, in this case specified as a concrete set of contacts — which might come from Iona herself, or else from a collection of users collaborating as informers. The SF application then creates U

with the contact list ℓ

— consisting of a random subset of the contacts in D

— and registers U

directly with Signal servers (step ②). Therefore, what Iona will have to prove is that she knows the subset of contacts from D

randomly selected by the SF application.

There are two challenges in the creation of the new user U

. The first is for the SF application to createU

and drawn from realistic probabilities distributions,

to prevent the Subversion Service from detecting that U

is part of an SF proof. To do this, the SF application first acquires a new phone number. For example the TEE can have access to a virtual phone number X

, provided by Iona. Then, the SF application uses a phone emulator to create a new signal user U

, and Iona can share the verification code sent to x

via SMS with the SF application.

Furthermore, a goal of the protocol is for U'

s social graph to appear real to the Subversion Service. Therefore, as noted above, D

can be a collection of the contacts of Iona pooled with the contacts of another small group of users, possibly Iona's friends. This way, it would appear like U

is another friend that recently downloaded Signal. The second challenge is ensuring that only the TEE knows, even if Iona knows U

's phone number and has access to its SMS. As it's Iona who created X

, she can create a new Signal account with U

's phone number. However, to obtain from the new registration, she would need to have access to U

's Signal backup, access to the phone emulator, or know U

’s PIN. Backups in Signal are disabled by default, so the TEE can choose not to enable it, and thus prevent Iona from retrieving a backup. Similarly, she cannot access the phone emulator, since it runs inside the TEE. Finally, Iona could attempt to guess U

’s PIN. However, the SF application can create a long alphanumeric PIN which would be infeasible to brute force. Additionally, fortunately, if Iona somehow were able to re-register the phone number X

to a new Signal account, the TEE’s account would be deactivated. Therefore, the TEE can verify that its account is still active when creating an SF proof. If this is no longer the case, the SF application can refuse to create the proof.

To complete our description of Figure 4, after registering U

and in Signal, the SF application sends U

to Iona (step ③), who then contacts the Subversion Service to request U

’s

contact list (step ④). If Signal’s SGX is broken, the Subversion Service can learn and leak it to Iona (step ⑤). With this information, Iona can request an SF proof to the SF application (step ⑥). After receiving the SF proof Π (step ⑦), Iona can finally send it to the verifier and ultimately prove that Signal’s SGX is broken and that its being exploited (step ⑧).