The Secrets Must Not Flow: Scaling Security Verification to Large Codebases

Linard Arquint[†], Samarth Kishor[‡], Jason R. Koenig[‡], Joey Dodds[‡], Daniel Kroening[‡], and Peter Müller[†]
[†]*Department of Computer Science, ETH Zurich, Switzerland*
[‡]*Amazon Web Services, USA*

***Abstract*—Existing program verifiers can prove advanced properties about security protocol implementations, but are difficult to scale to large codebases because of the manual effort required. We develop a novel methodology called Diodon that addresses this challenge by splitting the codebase into the protocol implementation (the Core) and the remainder (the Application). This split allows us to apply powerful semi-automated verification techniques to the security-critical Core, while fully-automatic static analyses scale the verification to the entire codebase by ensuring that the Application cannot invalidate the security properties proved for the Core. The static analyses achieve that by proving *I/O independence*, i.e., that the I/O operations within the Application are independent of the Core's security-relevant data (such as keys), and that the Application meets the Core's requirements. We have proved Diodon sound by first showing that we can safely allow the Application to perform I/O independent of the security protocol, and second that manual verification and static analyses soundly compose. We evaluate Diodon on two case studies: an implementation of the signed Diffie–Hellman key exchange and a large (100k+ LoC) production Go codebase implementing a key exchange protocol for which we obtained secrecy and injective agreement guarantees by verifying a Core of about 1 % of the code with the auto-active program verifier Gobra in less than three person months.**

1. Introduction

Security protocols such as TLS or Signal ensure security and privacy for browsing the web, sending private messages, and using cloud services. It is, thus, crucial that these ubiquitous and critical protocols are designed *and* implemented correctly.

Automatic protocol verifier tools such as Tamarin [2], [3] and ProVerif [4] make it viable to formally verify protocol *models*. Their applications to TLS [5], EMV [6], Signal [7], and 5G [8], [9] demonstrate that they can handle realistic protocols. However, proving protocol *models* secure does not result in secure *implementations* on its own. Coding errors such as omitted protocol steps (as in the Matrix SDK [10]) or ignored errors (e.g., returned by a TLS library [11], [12]) may invalidate all security properties proven for the corresponding models.

Verifying security properties for protocol *implementations* is possible as well [13], [14], [15]. For instance, Arquint et al. [14] first verify security properties for a Tamarin model of the protocol in the presence of a Dolev–Yao (DY) attacker [16] fully controlling the network. Then, they prove that the protocol implementation refines this model, i.e., that the model justifies every I/O operation performed by the implementation. Refinement guarantees that the implementation inherits the security properties proven for the model.

Existing approaches to verifying protocol implementations are sound *only* if they are applied to the *entire* implementation. Verifying only a subset of the codebase is unsound, and would fail to prevent, e.g., code seemingly unrelated to a security protocol accidentally logging key material [17], [18]. However, the required expertise and annotation overhead make it infeasible to verify entire *production* codebases, which often consist of hundreds of thousands of lines of code.

This work. We present Diodon[1], a proved-sound methodology that scales verification of security properties to large production codebases. Diodon works with codebases where a small, syntactically-isolated component implements a security protocol, whose security argument can be made separately from the rest of the code. Our methodology decomposes the overall codebase into this protocol implementation (the Core) and the remainder (the Application).

This decomposition allows us to apply different verification techniques to the two parts. We verify the Core using Arquint et al.'s approach to show refinement w.r.t. a verified Tamarin model, which requires precise reasoning about, e.g., the payloads of I/O operations. Instead of applying the same annotation-heavy approach to the Application, we use automatic static analyses to ensure that security-relevant data of the Core (in particular, secrets such as keys) does not influence any I/O operation within the Application. If this *I/O independence* holds, the Application cannot perform any I/O operations that could interfere with the protocol and invalidate its proven security. Additionally, we use static analyses to prove that the Application satisfies the assumptions made for the proof of the Core, in particular,

1. Diodon is a genus of fish known for their inflation capabilities. Erecting spines and scaling their volume by a multiple provide security, like our verification methodology.

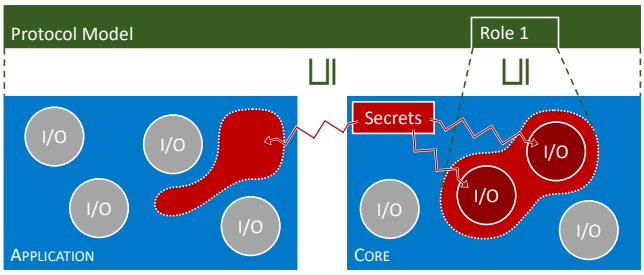

Figure 1. The Diodon methodology. We partition the codebase (blue) into the module implementing a protocol (Core) and the remaining codebase (Application). We prove that the Core refines (trace inclusion on the right) a particular role of the verified protocol model (green) by auto-active verification. We apply static analyses to the entire codebase to enforce that secrets (red) do not influence (red arrows) the I/O operations (gray circles) of the Application and to ensure that the Application cannot invalidate the security properties proved for the Core. Consequently, Diodon guarantees that the entire codebase refines the protocol model (trace inclusion in the middle) and, thus, enjoys all security properties proved for that model.

that the preconditions of Core functions hold when called from the Application and that the Application does not violate any invariants of Core data structures. These checks ensure that the proofs of the Core and the Application compose soundly. Consequently, the entire codebase refines the protocol model and enjoys all security properties proved for the model. Diodon significantly reduces the proof effort of verifying software that contains protocol implementations. Fig. 1 illustrates our methodology.

We prove I/O independence for the Application by executing an automatic taint analysis on the entire codebase to identify I/O operations that are possibly affected by secrets (also implicitly via control flow) and checking that all such operations are within the Core, which shows that the codebase's decomposition is valid and the Core is sufficiently large. It would be too restrictive to enforce that all secrets are confined within the Core. In most implementations, secrets exist outside the Core, e.g., the Application might have access to secrets either via program inputs or the Core's state (red area within the Application in Fig. 1). It is therefore essential to ensure (via a whole-program analysis) that the Application does not *use* these secrets to violate the security properties of the Tamarin model.

Most I/O operations within the Core correspond to a protocol step and are relevant for proving refinement w.r.t. a protocol model. In production code, however, the Core might also contain operations irrelevant to the protocol, such as logging a protocol step. To reduce the verification effort further, we also check I/O independence *within* the Core to classify each I/O operation based on whether it depends on secrets occurring in a protocol run (dark red circles in Fig. 1) or not (gray circles). The former need to be considered during the refinement proof, while the latter can safely be ignored. This classification simplifies the refinement proof and shortens the abstract protocol model.

We prove refinement of the Core w.r.t. a protocol model using an *auto-active* program verifier [19]. These program verifiers take as input an implementation annotated with specifications such as pre- and postconditions and loop invariants, and attempt to verify the implementation automatically using a satisfiability modulo theories (SMT) solver.

Auto-active verification is generally sound only if it is applied to the entire codebase because *all* callers of a function must establish its precondition and *all* functions must preserve data structure invariants. To ensure that our methodology is sound while avoiding this requirement for the Application, we design our methodology such that static analyses automatically discharge the proof obligations on the Application. Nevertheless, our methodology is flexible enough to permit complex interactions between the Core and Application, e.g., through concurrency and callbacks. Some assumptions remain, in particular, the absence of data races and undefined behavior; we discuss those in Sec. 4.4.

We prove Diodon sound, providing a blueprint for combining the distinct formalisms of auto-active verifiers and static analyses. First, we prove that a DY attacker can simulate all secret-*independent* I/O operations. Consequently, if a Tamarin model permits every secret-*dependent* I/O operation in a codebase, then this codebase refines the model. Second, we show that Diodon reasons about these secret-*dependent* I/O operations *without* verifying the entire codebase. I.e., we construct the corresponding proof for the entire codebase by starting from the proof for the Core, which we obtain from auto-active verification, and discharging the remaining proof obligations using our static analyses.

We evaluate Diodon on two Go implementations, a signed Diffie–Hellman (DH) key exchange and a fork of the Amazon Web Services (AWS) Systems Manager Agent (SSM Agent) [20], a large (100k+ LoC) codebase. Part of the latter codebase implements an experimental protocol for encrypted shell sessions. We prove secrecy for and injective agreement on the session keys established by both protocols. For the SSM Agent codebase, Diodon allowed us to limit auto-active verification to only about 1% of the entire codebase, which took less than three person months. This demonstrates that Diodon enables, for the first time, the verification of strong security properties at the scale of production codebases. Our static analyses and case studies are open-source [21], [22].

Contributions. We make the following contributions:

- ➤ We present a scalable verification methodology for implementations of security protocols within large codebases, which applies to any codebase with a clear distinction between the protocol core and the rest of the code.
- ➤ We identify I/O independence, enabling concise protocol models for complex implementations.
- ➤ We show how to use static analyses to automatically discharge the Core's proof obligations, enabling Diodon to scale to large codebases.
- ➤ We prove the soundness of I/O independence w.r.t. a DY attacker, and the soundness of Diodon's combination of auto-active verification and static analyses.
- ➤ We evaluate our methodology on two case studies, an implementation of the signed DH key exchange and an AWS Systems Manager Agent fork, to demonstrate that Diodon scales to large, production codebases.

```
1  package core

3  type Chan struct {
4    psk []byte
5    cb  Cb
6  }

8  type Cb = func(msg []byte)

10 //@ req  acc(msg, 1)
11 //@ func CbSpec(msg []byte)

13 //@ req  cb != nil ==> cb implements CbSpec{}
14 //@ pres psk != nil ==> acc(psk, 1)
15 //@ ens  Inv(c)
16 func InitChannel(psk []byte, cb Cb) (c *Chan) {
17   //@ inhale AliceIOPermissions()
18   c = &Chan{append([]byte(nil), psk...), cb}
19   go continuousRecv(c)
20   return c
21 }

23 //@ pres c != nil ==> Inv(c)
24 //@ pres msg != nil ==> acc(msg, 1)
25 func Send(c *Chan, msg []byte) {
26   if c == nil || msg == nil { return }
27   fmt.Printf("Send␣%x\n", msg)
28   packet := append(msg, HMAC(msg, c.psk)...)
29   sendToNetwork(packet)
30 }

32 /*@ pred Inv(c *Chan) {
33   c != nil && acc(c, 1/2) &&
34   acc(c.psk, 1/2) && AliceIOPermissions() &&
35   (c.cb != nil ==> c.cb implements CbSpec{})
36 } @*/
```

Figure 2. Sample CORE for a simple MAC communication. In Go, function definitions take a list of input parameters and may have a second list for outputs. We omit the `continuousRecv` goroutine's implementation that invokes the `c.cb` closure (if non-**nil**) whenever a message has been received. We simplify the representation of I/O permissions, which describe permitted protocol-relevant I/O operations, and omit proof-related statements.

2. Running Example of DIODON

We demonstrate the core ideas of DIODON on a sample program in the Go programming language, which implements a simple message authentication code (MAC) protocol that sends and receives signed messages using a pre-shared key. In the remainder of this section, *we* refers to a user of DIODON. First, we manually split the codebase into CORE and APPLICATION following function boundaries. We make the CORE as small as possible to reduce auto-active verification efforts while making sure that the entire protocol implementation is contained therein and that we can define an invariant for the CORE's application programming interface (API) with which the APPLICATION interacts.

We model the protocol and prove security properties with the TAMARIN protocol verifier [2], [3]. The goal is to prove that the entire program, i.e., the composition of the CORE and APPLICATION, refines the TAMARIN model and, thus, satisfies the same security properties as the protocol model. We auto-actively verify the CORE using GOBRA [23] and apply the automatic ARGOT [24] static analyses to the entire codebase.

CORE. The CORE (Fig. 2) consists of a struct definition,

```
1  package main

3  import . "core"

5  func main(psk []byte) {
6    cb := func(m []byte) {fmt.Printf("%x\n", m)}
7    c := InitChannel(psk, cb)
8    Send(c, []byte("hello␣world"))
9    fmt.Printf("Log:␣message␣sent.\n")
10   // fmt.Printf("%v\n", c)
11 }
```

Figure 3. Sample APPLICATION that is a client of Fig. 2. We omit parsing of command line arguments for presentation purposes and, thus, assume that `psk` stores the parsed pre-shared key.

```
1  rule Alice_Send:
2      let packet = <msg, sign(msg, psk)> in
3      [ Alice_1(rid, A, B, psk), In(msg) ]
4    --->
5      [ Alice_1(rid, A, B, psk), Out(packet) ]
6  rule Alice_Recv:
7      let packet = <msg, sign(msg, psk)> in
8      [ Alice_1(rid, A, B, psk), In(packet) ]
9    --->
10     [ Alice_1(rid, A, B, psk), Out(msg) ]
```

Figure 4. TAMARIN model excerpt for the MAC protocol implemented in Fig. 2.

two API functions, `InitChannel` and `Send`, which access this struct, and a predicate `Inv` that represents the separation logic [25] invariant used to verify the functions. The definition of `Inv` includes permissions to access the struct fields and the pre-shared key's bytes. Separation logic controls heap access with these permissions to reason about side effects and to prove data-race freedom, as detailed in Sec. 3.2. Accessibility predicates (`acc`) represent permissions in specifications. Their first argument indicates the heap location and the second argument characterizes the permitted access: a value of 1 provides exclusive read and write access, and any value strictly between 0 and 1 provides read-only access that might be shared. For instance, `acc(msg,1)` on line 24 passes full permission to write the contents of `msg` (if it is non-**nil**) from a caller to function `Send`, and back to the caller when the function returns. Pre- and postconditions start with the keyword `req` and `ens`, respectively, and we use `pres` as syntactic sugar for properties that are preserved, that is, act as pre- *and* postconditions.

To receive incoming packets, the CORE spawns a goroutine (lightweight thread) on line 19 executing the function `continuousRecv`. We omit its implementation in the figure for space reasons. The goroutine repeatedly calls a blocking receive operation, checks the MAC's validity, and on success calls the closure that is stored in the struct field `cb` as a callback. If the callback is non-**nil**, it delivers the resulting message to the APPLICATION.

We verify the CORE for any callback closure that satisfies the specification `CbSpec` (cf. line 13 & 10–11), which states that a caller must pass permission for modifying the message to the closure when invoking it and that the closure does not have to return any permissions. On line 18, we duplicate the pre-shared key (which the APPLICATION obtains

as a program input) to keep the Core's memory footprint separated from the Application. Thus, we can pass half of the permissions for accessing the struct fields to the goroutine spawned on line 19 and store the remaining permissions in the invariant `Inv`, which is then returned to the caller of `InitChannel`.

Application. The Application (Fig. 3) consists of a single function that creates a closure that will print any incoming message, initializes the Core with the pre-shared key `psk` and this closure, and then sends a message by invoking the `Send` function of the Core. In realistic programs, the Application might have thousands of lines of code, making auto-active verification prohibitively expensive. Diodon allows us to apply automatic static analyses instead, as detailed below.

Protocol model. Fig. 4 excerpts our abstract protocol model as a multiset rewriting system in Tamarin (cf. Sec. 3.1) with two protocol roles, Alice and Bob, each starting off with a pre-shared key `psk`. Both roles can send and receive unboundedly many packets, each of which are the composition of a message plus the appropriate MAC. To make zero assumptions about the messages themselves, we treat them as being attacker-controlled, i.e., the sending role obtains a message from the attacker-controlled network via an `In` fact, as shown on line 3. For this protocol model, we prove that all received messages were previously sent by either Alice or Bob, unless the attacker obtains the pre-shared key, which Tamarin proves automatically.

In order to prove that our program is actually a refinement of this model and, thus, inherits all proven properties, Diodon combines auto-active verification and static analyses to obtain provably-sound guarantees.

Verification. Our goal is to ensure that the composition of the Application and Core refines the abstract Tamarin model, i.e., the program's I/O behavior is contained in the model's I/O behavior. This refinement implies that any trace-based safety property proven in Tamarin also holds for the program because the program performs the same or fewer I/O operations than the protocol model. Diodon splits the refinement proof into three steps: We prove that (1) non-protocol I/O is independent of protocol secrets, (2) all remaining I/O refines a protocol role, and (3) the proof steps soundly compose.

First, we manually identify protocol-relevant calls to I/O operations within the Core. In our example, these are the `sendToNetwork` call and the corresponding network receive operation. We then perform an automatic taint analysis on the entire codebase to prove I/O independence for all other calls to I/O operations (in our example, the calls to `Printf`), i.e., we check that they do not use tainted data. Uncommenting line 10 in Fig. 3 would result in printing all struct fields of variable `c` including the pre-shared key `psk`, which is the only secret. I/O independence would correctly fail for this modified program, resulting in an error message indicating the flow of secret data to the print statement. In general, we treat data as a secret (i.e., tainted) if the protocol model's attacker might not know this data. Checking I/O independence ensures that we do not miss any protocol-relevant I/O operations and that the chosen Core is sufficiently large.

The Core may execute protocol-relevant operations not only by performing I/O operations, but also by communicating with the Application. For example, Alice's protocol step of taking an arbitrary message from the environment (before signing and sending it), is implemented by the Core obtaining `msg` from the Application (line 25 in Fig. 2). Similarly, Alice may (after receiving a packet and checking its signature) release its payload to the environment, which is implemented as passing the payload to the Application when invoking the closure `c.cb` (not shown in Fig. 2). To handle such protocol-relevant operations uniformly, we treat them as *virtual* protocol-relevant I/O operations. This allows us to enforce or assume constraints on the arguments' taint status while creating the necessary proof obligations in the next step of the refinement proof. Here, the fact that releasing the payload is permitted by the protocol model (line 10 in Fig. 4) informs the taint analysis that the callback's argument may be considered as untainted, which allows printing it on line 6 in Fig. 3.

Second, we prove the Core using the auto-active Gobra verifier. This proof includes showing that the protocol model permits every protocol-relevant I/O operation, including virtual I/O. Note that step (1) ensures that these operations must all be in the Core. We use an I/O specification for each protocol role describing the permitted protocol-relevant I/O operations (cf. Sec. 3.3). In our example, Alice obtains the permissions to perform these operations during the initialization of the Core (line 17) and maintains them as part of the invariant (line 34), where `inhale` adds the specified permissions to the current program state by assumption. When performing a protocol-relevant I/O operation, like `sendToNetwork`, Gobra proves that the I/O specification permits this operation with the specific arguments. Otherwise, Gobra reports a verification failure.

Third, since the Gobra proof for the Core assumes that callers respect the functions' preconditions, Diodon restricts the class of supported pre- and postconditions such that static analyses are able to prove that the Application satisfies them. For example, the precondition of `Send` requires exclusive access for the argument `msg`; Diodon enforces this condition using a combination of static pointer, escape, and pass-through analyses to ensure that no other goroutine accesses the memory pointed to by `msg`. `Send`'s other precondition requires the Core's invariant to hold, which is established by `InitChannel`. The Application could in principle violate this precondition, for example, by creating a `Chan` instance without calling `InitChannel`, or by invalidating the invariant of a `Chan` instance through field updates or concurrency. We apply this combination of static analyses to prevent all such cases (cf. Sec. 4.3).

Together, these three proof steps ensure that the program refines the abstract Tamarin model and inherits the security properties proved for the model.

3. Background

In this section, we provide the necessary background on the verification techniques that we reference in the rest of the paper. We detail verification of abstract protocol models (Sec. 3.1), verification of implementations (Sec. 3.2), and code-level refinement (Sec. 3.3), which transfers security properties from a protocol model to implementations.

3.1. Protocol Model Verification

We model a security protocol and prove security properties about it using Tamarin, an automated protocol model verifier. A protocol model consists of protocol roles and a DY attacker that are expressed as multiset rewrite rules. Each rule has the shape L `--[` A `]->` R, where L and R are multisets of facts and A is a set of actions. The system's state S is a multiset of facts, which is initially empty, and a rule can be applied if L is (multiset) included in S, i.e., $L \subseteq_m S$. Applying a rule removes the facts in L from and adds those in R to the system state, i.e., results in a new state $S \setminus_m L \cup_m R$. While most facts are user-defined to represent the state of a protocol role, Tamarin uses certain predefined facts. In particular, `In(x)` and `Out(x)` facts represent receiving and sending a message `x` from and to the attacker-controlled network, respectively. Since Tamarin uses equational theories to describe otherwise uninterpreted functions, such as $dec(enc(p, pk(sk)), sk) = p$ to describe asymmetric decryption (dec) w.r.t. asymmetric encryption (enc), Tamarin performs multiset inclusion modulo equational theories. A possible sequence of rule applications forms a trace that consists of each rule application's set of actions (A). Tamarin symbolically explores all possible traces involving unboundedly many instances of protocol roles to prove a security property, or, if the proof fails, provides a trace of an attack as a counterexample to the security property.

Tamarin's symbolic DY attacker fully controls the network and can construct new messages by applying symbolic operations to terms it has observed. However, a symbolic attacker can neither perform arbitrary computations, nor can it exploit algorithmic weaknesses or side channels such as timing. Nevertheless, verification using Tamarin guarantees the absence of many relevant vulnerabilities and has proven effective in applications to 5G [8], EMV card protocols [26], and the Noise protocol family [27].

3.2. Code-Level Verification

We prove safety and functional properties of an implementation by reasoning about all possible executions statically, without any runtime checks. In this context, the term *safety* expresses that an implementation neither causes runtime exceptions nor undefined behavior. In particular, it covers the absence of memory errors, buffer overflows, and data races. *Functional properties* are implementation-specific and express the desired behavior, e.g., that a sorting algorithm's result is a sorted permutation of the input.

We perform *modular* verification, i.e., we reason about each function in an implementation in isolation. To do this, we equip a function with a *specification* that consists of a pre- and postcondition. A function's precondition is a logical formula specifying all valid program states in which this function can be called, and a function's postcondition specifies properties that hold for all valid program states after executing the function's body.

To reason about heap-manipulating programs, we use *separation logic* [25]. Separation logic allows us to express precisely which heap fragment f a function operates on and provides connectives that split and combine heap fragments, e.g., when calling another function that operates only on a subfragment $f' \subseteq f$, we know that the function does not modify any heap location in the *frame* $f \setminus f'$.

Separation logic associates a *permission* with each heap location. A permission represents ownership of a particular heap location and is required for each access. We use fractional permissions [28] to distinguish between exclusive and shared ownership, which permits multiple threads in a concurrent program to simultaneously share ownership of a heap location. In specifications, we express permission to a heap location `l` with fraction `p` as `acc(l,p)`, cf. Sec. 2.

The *separating conjunction* $*$ is akin to regular conjunction, but requires the *sum* of the permissions in both conjuncts. For instance, `acc(l1,1)` $*$ `acc(l2,1/2)` specifies full and half permissions for heap locations `l1` and `l2`, respectively. Additionally, this example implicitly specifies that the heap locations are disjoint, i.e., `l1` $\neq$ `l2`. Otherwise, if `l1` and `l2` were aliased, the permission amounts would add up to $3/2$ contradicting separation logic's invariant that at most a full permission exists for a heap location. In our code listings, we overload `&&` to denote separating conjunction.

We use separation logic *predicates* [29] to abstract over individual permissions to heap locations as demonstrated by the Core's invariant in our running example. Conceptually, we can treat a predicate instance such as `Inv(c)` in Fig. 2 as a shorthand notation for the predicate's body.

A separation-logic proof guarantees safety, in part by requiring a proof that each function has sufficient permissions for each heap access. E.g., a buffer overflow corresponds to accessing an array element out of bounds; this is prevented since allocating an array creates permissions only for in-bound elements. Similarly, data races are prevented since two threads simultaneously writing the same heap location would require that each thread has write permission for this heap location, which is impossible as there is only a single write permission for any given heap location.

Various separation logic-based program verifiers exist, including VeriFast [30] and VST [31] for C, Gobra [23] for Go, Nagini [32] for Python, and Prusti [33] for Rust. These verifiers are *auto-active*, i.e., use manual annotations and proof automation to prove properties about programs.

3.3. Code-Level Refinement

Arquint et al. [14] split verification of security protocol implementations into two steps: proving security properties

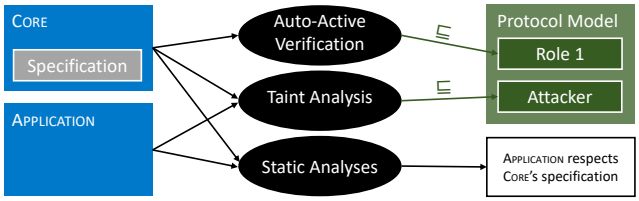

Figure 5. Diodon proves that the entire codebase (blue) refines a protocol model (green) by soundly composing auto-active verification with automatic static analyses. We auto-actively verify the Core based on its specification to show that the protocol-relevant I/O operations refine a protocol role (upper trace inclusion). This specification is partially generated from the protocol model, which is omitted. The static taint analysis proves that all other I/O operations within the entire codebase refine our attacker model (lower trace inclusion). Lastly, we discharge the Core's assumptions by applying automatic static analyses, proving that the Application satisfies the calling restrictions expressed in the Core's specification.

for an abstract protocol model using Tamarin, and using an auto-active program verifier to prove that an implementation refines this model. This approach disentangles proving global security properties from local reasoning about implementations, while exploiting each tool's automation.

To connect these two steps, Tamarin automatically generates an *I/O specification* for each protocol role in a given abstract protocol model. A protocol role's I/O specification describes the permitted I/O operations including their sequential ordering and arguments. By verifying that an implementation executes at most the operations permitted by the I/O specification, we prove that all executions of this implementation result in a trace of I/O operations that is included in the set of traces considered by Tamarin when verifying the security properties. Thus, the implementation satisfies the same security properties as the model.

The I/O specification is expressed in I/O separation logic [34], a dialect of separation logic, which is readily supported by separation logic-based verifiers. I/O separation logic extends the use of permissions beyond guarding heap accesses to guarding I/O operations by associating an *I/O permission* with each I/O operation. We equip library functions performing I/O operations with a specification that checks and consumes the corresponding I/O permission.

Successful code-level verification against the library functions equipped with I/O permissions guarantees that an implementation performs at most the I/O operations specified in the Tamarin model and respects their sequential ordering. Otherwise, verification fails because a prohibited I/O operation would require an unavailable I/O permission.

4. Diodon

Our methodology, Diodon, proves security properties for implementations by refinement and scales to large codebases by significantly reducing verification effort. Diodon enables more concise protocol models than previous approaches and leverages fully automatic analyses on most of the implementation to discharge proof obligations.

We manually decompose a codebase into two syntactically isolated components, the Core implementing a security protocol, and the Application consisting of the remaining code. Typically, this decomposition is natural and follows module boundaries as a protocol's implementation is localized. As illustrated in Fig. 5, this decomposition allows us to split the proof that the entire codebase refines a protocol model into three steps and uses the best-suited tool for each step. We explain in Sec. 4.1 how Diodon identifies which I/O operations are protocol-relevant by performing a static taint analysis on the entire codebase. Sec. 4.2 covers the Core's auto-active verification using Gobra proving that protocol-relevant I/O operations refine a particular protocol role. Finally, Sec. 4.3 explains how we discharge the assumptions made when auto-actively verifying the Core by performing static analyses on the Application.

4.1. I/O Independence

One of our key insights is to distinguish between I/O operations that are relevant for a security protocol from those that are not (e.g., sending log messages to a remote server). This distinction has two main benefits. First, protocol-irrelevant operations do not have to be reflected in the abstract protocol model, which makes the model concise, more general, and easier to maintain, review, and prove secure. Second, by ensuring that protocol-irrelevant I/O operations cannot possibly invalidate the security properties proven for the protocol model, we do not have to consider them during the laborious auto-active refinement proof and instead can check simpler properties using automatic static analyses. We classify all calls to I/O operations as either protocol-relevant or protocol-irrelevant. In the Core, an I/O operation is protocol-irrelevant if and only if its specification requires no I/O permissions. In contrast, all I/O operations in the Application are implicitly considered protocol-irrelevant.

To ensure that I/O operations classified as protocol-irrelevant indeed do not interfere with the protocol or invalidate proven security properties of the protocol, we check that they do not use any secret data (such as key material); more precisely, we check *non-interference* between protocol secrets and the parameters of these I/O operations. We call this important property of an I/O operation *I/O independence*. It guarantees that an I/O operation cannot possibly invalidate the protocol's proven security properties: any I/O operation that uses only non-secret data could also have been performed by the DY attacker and, thus, was already considered by Tamarin during the protocol verification. In other words, proving that all protocol-irrelevant I/O operations satisfy I/O independence guarantees that they refine our DY attacker (cf. Sec. 5.1).

From a cryptographic perspective, I/O independence allows us to reduce the security of an entire codebase to the security of its Core. This reduction is valid because most of the Application can be treated as part of the attacker, while the parts of the Application that manipulate secrets (e.g., code that loads long-term keys from disk) are shown not to perform I/O, and thus can conceptually be considered part of the Core, without introducing violations of the I/O specification of its protocol role.

We prove I/O independence by performing an automatic static taint analysis on the entire codebase that includes implicit information flows from control flow. A taint analysis checks for a set of sources and sinks whether there are any flows of information from a source to a sink. The analysis starts at each source, i.e., a function which produces secret data, and explores how secret information propagates through the program by keeping track of program locations storing a *tainted* value, i.e., a value that is influenced by a source. We disallow branching on tainted data to avoid information flows via control flow.

We configure the taint analysis to consider all calls to key-generation functions within the Core and long-term secrets that are passed as program inputs, like the pre-shared key in our running example, as sources because the DY attacker does not have access to them. This set of initial sources taints all protocol secrets including session keys. E.g., if the Core implements a DH key exchange, the analysis correctly considers the generated secret key and the resulting shared key tainted because the shared key is computed from the secret key and the other party's public key. We then configure the taint analysis to treat all I/O operations in the Application as well as all protocol-irrelevant I/O operations in the Core as sinks. We use Capslock [35] to identify such I/O performing functions in the Go standard library. We consider all functions with at least one of the following capabilities as a sink: write to the file system or network, modify the state of the operating system (e.g., `os.Setenv`), perform a system call, and execute external programs (e.g., `(*os/exec.Cmd).Run`).

We run the taint analysis on the entire codebase. If taint reaches a sink, verification fails because a secret reached a supposedly protocol-irrelevant I/O operation. Otherwise, we have correctly identified the protocol-relevant I/O operations (and thereby confirmed that we have correctly delimited the Core); it remains to reason about those I/O operations, as we discuss next.

4.2. Core Refinement

We auto-actively verify the entire Core, which allows us to state and prove (besides safety and functional correctness) precise constraints about protocol-relevant calls to I/O functions and their arguments. We prove that the implementation uses the payload for each I/O operation specified in the protocol model. The corresponding verification effort is feasible since, in industrial codebases like our main case study, the Core comprises only a small fraction of the codebase.

We prove that the Core refines a protocol role by building on the approach explained in Sec. 3.3. In particular, we equip each protocol-relevant I/O operation with a specification that requires an I/O permission for executing this operation with the provided arguments. Since we provide exactly the I/O permissions justified by the protocol role's model to the Core during its initialization, successful verification with Gobra implies that the Core executes at most the protocol-relevant I/O operations permitted by the model and, thus, refines this protocol role. This approach is inspired by Arquint et al. [14], but differs in three significant ways.

First, we do not auto-actively verify the entire codebase and, instead, verify only the Core. As we will discuss in Sec. 4.3, we syntactically restrict the preconditions of the Core functions so that we can apply automatic static analyses to check that each call from the Application satisfies them, which is necessary for soundness.

Second, our approach supports codebases that use multiple instances of the Core, e.g., to run multiple roles of the protocol or to run the protocol multiple times. Since Tamarin considers unboundedly many role instantiations, we can soundly create the required I/O permissions for executing a role instance whenever we create a new Core instance. These I/O permissions are bound to an instance's unique identifier such that each Core instance has its own set of I/O permissions for executing the security protocol once.

Third, to reflect that interactions in the model between the protocol and the environment may manifest as interactions between the Core and the Application in the implementation, we treat the boundary between them as a virtual network interface and enforce I/O permissions for the corresponding virtual I/O operations, as we illustrated in Sec. 2.

4.3. Analyzing the Application

We now show how to scale auto-active verification to the entire codebase. Applying auto-active verification to an entire codebase is typically not feasible within the resource constraints of industrial projects. A key insight of Diodon is that this is not necessary: we can use static analyses to automatically discharge separation logic proof obligations arising in the Application to obtain, together with the verified Core, a proof in separation logic for the *entire* codebase.

The refinement proof for the Core is valid in the context of the entire application if (1) each call to a Core function from the Application satisfies the function precondition, and (2) the Application respects permissions on memory accesses. Our soundness proof for Diodon (cf. Sec. 5.2) ensures that these proof obligations are sufficient and that our novel combination of static analyses can soundly discharge them. We illustrate these proof obligations and how we discharge them by considering the exemplary Core function in Fig. 6, taking two integer pointers as input and returning an integer pointer. This function maintains the Core invariant (if c is non-**nil**), needs full permissions for both inputs, and returns full permissions for the input and output parameters (if they are non-**nil**). Thus, we cannot allow, e.g., the Application to pass two aliased arguments (cf. Sec. 3.2) to this function or to concurrently access heap locations pointed to by these arguments as this would violate the precondition, i.e., the permissions specified therein.

Implicit annotations. To construct a proof for the entire codebase, we enrich the Application with a *hypothetical program instrumentation* that connects the Application to the necessary proof obligations imposed by the proof of the Core. These *implicit annotations* track the permissions

```
1  //@ pres c ≠ nil ⟹ inv(c)
2  //@ pres a0 ≠ nil ⟹ acc(a0)
3  //@ pres a1 ≠ nil ⟹ acc(a1)
4  //@ ens  r ≠ nil ⟹ acc(r)
5  func (c *Core) ApiFn(a0, a1 *int) (r *int)
```

Figure 6. Example of a signature and specification of a CORE API function.

that the APPLICATION owns by using sets of heap locations and a *program invariant* specifying permissions for the heap locations in these sets. More precisely, each thread has a set lhs (short for "local heap set") for thread-local objects such as buffers, and a set ihs (short for "invariant heap set") for CORE instances. Similarly, a global set ghs ("global heap set") keeps track of objects that might be shared between threads, which becomes relevant later. These sets are mutable and, thus, their content depends on a particular program point. The sets lhs and ihs allow us to state the following local program invariant that must hold at every program point in the APPLICATION.

$$\Pi_l \triangleq \left(\bigstar_{l \in \mathsf{lhs}} \mathbf{acc}(l)\right) \star \left(\bigstar_{i \in \mathsf{ihs}} \mathbf{inv}(i)\right)$$

Here, the iterated separating conjunction $\bigstar_{e \in s} a(e)$ conjoins the assertions $a(e)$ using separating conjunction for all elements e in set s. Π_l states that a thread holds full permissions for all objects in lhs and the CORE invariant for all instances in ihs. In addition, these permissions are disjoint allowing the APPLICATION to write to heap locations in lhs without breaking the CORE invariant. When a thread obtains or gives up permissions, our implicit annotations adjust lhs and ihs to maintain the program invariant.

Fig. 7 shows these *implicit annotations* for calls to `ApiFn`. To highlight that each statement in the APPLICATION maintains the program invariant, we assert Π_l on lines 1 and 22. For each permission required by the callee's precondition, we remove the corresponding heap location from one of the sets to reflect that ownership is being passed to the callee. Assuming (for now) that the location was originally in the set, this removal extracts the corresponding permission from Π_l, as illustrated by the intermediate assert statements starting on lines 3, 6, and 10 for the three arguments of the call. After the call, we conversely add those heap locations to the sets for which the callee's postcondition provides permissions.

For each permission in the precondition, if the corresponding heap location was contained in one of the sets before the removal operation, then we have effectively proved that the precondition holds (syntactic restrictions ensure that the preconditions cannot contain constraints other than permission requirements, see below). In the rest of this subsection, we explain how we use static analyses to check this set containment. Then, we explain the proof obligations for memory accesses within the APPLICATION.

Guaranteeing permissions for parameters. For the arguments `a0` and `a1` (we will discuss the core instance `c` below), we need to prove that (1) $\{\mathtt{a0}, \mathtt{a1}\} \subseteq \mathsf{lhs}$ holds before the call to `ApiFn` and (2) `a0` and `a1` do not alias. If (2) was violated, `a1` would no longer be in lhs after removing `a0` on

```
1  //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i))
2  //@ ihs := ihs \ {c}
3  //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
4  //@          (c ≠ nil ⟹ inv(c))
5  //@ lhs := lhs \ {a0}
6  //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
7  //@          (c ≠ nil ⟹ inv(c)) ⋆
8  //@          (a0 ≠ nil ⟹ acc(a0))
9  //@ lhs := lhs \ {a1}
10 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
11 //@          (c ≠ nil ⟹ inv(c)) ⋆
12 //@          (a0 ≠ nil ⟹ acc(a0)) ⋆
13 //@          (a1 ≠ nil ⟹ acc(a1))
14 r := c.ApiFn(a0, a1)
15 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i)) ⋆
16 //@          (c ≠ nil ⟹ inv(c)) ⋆
17 //@          (a0 ≠ nil ⟹ acc(a0)) ⋆
18 //@          (a1 ≠ nil ⟹ acc(a1)) ⋆
19 //@          (r ≠ nil ⟹ acc(r))
20 //@ lhs := lhs ∪ ({a0, a1, r} \ nil)
21 //@ ihs := ihs ∪ ({c} \ nil)
22 //@ assert (★_{l∈lhs} acc(l)) ⋆ (★_{i∈ihs} inv(i))
```

Figure 7. *Conceptually* inserted implicit annotations for a CORE API call `r := c.ApiFn(a0, a1)` in the APPLICATION. The assert statements solely illustrate our deductions and, thus, can be omitted.

line 5 in Fig. 7, i.e., we would obtain only $\mathbf{acc}(\mathtt{a0})$ instead of $\mathbf{acc}(\mathtt{a0}) \star \mathbf{acc}(\mathtt{a1})$.

We discharge these two proof obligations by checking the conditions (C6) and (C7) in Fig. 8, resp., using static analyses. We check (C6) by using a thread escape analysis, which delivers judgments $\mathrm{local}(x)$ for a particular program point expressing that $*x$ is definitely not accessible by any other thread. We show in Sec. 5.2 that (C6) suffices to discharge $\{\mathtt{a0}, \mathtt{a1}\} \subseteq \mathsf{lhs}$ (if the arguments are non-**nil**) by proving a lemma that relates $\mathrm{local}(x)$ for a program point p with $x \in \mathsf{lhs}$. We obtain (C7) by applying a pointer analysis, which computes may-alias information, i.e., $\mathrm{pts}(x)$ for a pointer x, where $a \in \mathrm{pts}(x)$ denotes that $*x$ may-alias any location allocated at site a. More precisely, we check for each pair of arguments that the sets of locations they may-alias are disjoint, which is sufficient as we restrict parameters to be shallow.

Guaranteeing the CORE invariant. Similarly to parameters, we have to prove that $\mathtt{c} \in \mathsf{ihs}$ holds such that removing `c` from ihs on line 2 grants us the CORE invariant $\mathbf{inv}(\mathtt{c})$, if `c` is non-**nil**. In Sec. 5.2, we prove that $\mathtt{c} \in \mathsf{ihs}$ if the following premises hold. (1) The CORE instance `c` must have been returned as a result from a CORE API function initially establishing the CORE invariant, e.g., `InitChannel` in our running example. (2) All heap modifications in the APPLICATION must not modify the internal state of the CORE instance, even through an alias, since this could invalidate the CORE invariant.

In a single-threaded program without callbacks from the CORE to the APPLICATION, the above premises are sufficient. However, in the presence of these two features, we need to ensure that the APPLICATION does not call two CORE functions on the same CORE instance simultaneously, which would effectively duplicate permissions and, thus, make reasoning unsound: (3) The APPLICATION must not pass the same CORE

	Condition	Details
C1	Core init	Core instances are created in a function ensuring the invariant in its postcondition
C2	No modification	Application does *not* write to Core instances' internal state, even through an alias
C3	Core preservation	Core instances are passed only to Core functions that preserve the invariant
C4	Core locality	Core instances are used only in the thread they are created in
C5	Core callback	Core APIs are not invoked in Application callbacks
C6	Parameter locality	Parameters to Core APIs are local
C7	Disjoint parameters	Parameters to the same Core API call do not alias one another
C8	Application access	Reads and writes in the Application occur to memory allocated in the Application or transferred from the Core

Figure 8. Sufficient conditions checked by our static analyses, grouped into those involving Core instances, other parameters to Core functions, and memory accesses in the Application.

reference to more than one thread, and (4) the Application must not call a Core function in a callback on the same instance that is invoking the callback.

We establish the four premises by checking the conditions (C1) to (C5) in Fig. 8. Conditions (C1) and (C3) can be enforced by checking that the Application calls only Core functions that establish or preserve the invariant. While the postconditions provide this information for Core instances that are passed as arguments or results, our analyses need to prevent a subtle loophole: We need to prevent Core functions from allocating a Core instance *without* establishing its invariant and letting the Application access it via global variables or shared memory. We implemented a pass-through analysis computing $\text{pass}_f(x, r)$ for a function f stating that outside of calls to f, $*x$ definitely passed through return parameter r. We use this pass-through analysis to ensure that all references to Core instances in the Application are obtained exclusively through the return parameter, such that the postcondition establishes the invariant.

To establish (C2), we use a pointer analysis to ensure that all reads and writes in the Application never access a Core instance's internal state. In particular, we ensure that the Application accesses *only* heap locations that must-not-alias locations reachable from ihs, i.e., internal state of Core instances. Since we use a sound pointer analysis, this check conservatively over-approximates the heap locations about which the Core invariant states properties. While it is possible to access Core memory without breaking the invariant, we could not treat the Core invariant as an opaque separation logic resource when analyzing the Application, which would require a static analysis capable of reasoning about fractional permissions and arbitrary functional properties.

For (C4), we use the thread escape analysis to ensure that each Core instance does not escape its thread (we show local(c) for each call to Core instance c), guaranteeing that each thread operates on a disjoint set of Core instances ihs. While it is possible to safely pass Core instances between threads, this would require a significantly more sophisticated static analysis that can reason about the ordering of concurrent executions. Condition (C5) is enforced by checking that the call graph does not contain Core functions invoked transitively from Application callbacks. Allowing such calls would require proving that the same instance is not used in the inner call, which requires a more precise pointer analysis.

Our explanations generalize from the exemplary Core function in Fig. 6 to arbitrary Core API functions as long as they satisfy the following restrictions on pre- and postconditions. We support an arbitrary number of input and output parameters with arbitrary value and pointer types. Our restrictions mandate that Core API functions preserve the Core invariant and full permissions for each parameter of pointer type, both only under the condition that the receiver and parameters are non-**nil**. Additionally, the postcondition specifies full permissions for each return parameter if it is non-**nil** and of pointer type. These restrictions ensure that preconditions do not specify functional properties, such as require an input array to have a certain length, which we cannot check using our static analyses. As seen with our example in Fig. 6, we cannot rule out that the Application passes **nil** as an argument because there is no sound nilness analysis for Go to the best of our knowledge and, thus, we account for this possibility in our restrictions.

Application memory access. Finally, we need to ensure that the Application accesses only memory to which it has permissions. While we have already established that the Application does not write to internal state of Core instances (C2), we need to particularly consider the case where memory is transferred after its allocation between the Core and the Application. The other case, namely the Core or Application allocating memory without transferring it, is straightforward. I.e., if Core-allocated memory is never transferred to the Application then the Application cannot access it. Similarly, if Application-allocated memory is not transferred to the Core then the Application retains the corresponding permission.

Checking condition (C8) is sufficient. If Application-allocated memory is transferred to the Core, our syntactic restrictions guarantee that the Core only temporarily borrows the corresponding permissions until the Core API call returns. If the Core allocates memory and transfers it to the Application, the Core must also transfer the corresponding permissions, which we enforce via our pass-through analysis checking that this transfer happens via a return parameter as our syntactic restrictions guarantee that the postcondition specifies permissions for this return parameter. Using (C8), we prove that each memory access in the Application is to a location in either lhs or ghs (cf. Sec. 5.2). In the latter case, we need to reason about concurrent access. We assume that the Application is free from data races: if two accesses race, then the program is invalid according to the Go specification. If there are no races, then there is some total order in which the threads can atomically pull permission from ghs, perform the access, and then return permissions to ghs before the next thread needs to access the same location.

4.4. Threat Model, Assumptions, and Limitations

The DIODON methodology provides strong guarantees for large codebases, namely that a codebase satisfies the same trace-based safety properties as the abstract protocol model. Like other verification techniques, DIODON relies on assumptions about the codebase, execution environment, and the employed tools.

DIODON considers an arbitrary number of potentially concurrent protocol sessions, allowing the DY attacker to, e.g., replay messages across sessions or apply cryptographic operations thereto to construct messages of unbounded size. As is standard for symbolic cryptography, we assume cryptographic operations such as signing are perfectly secure, e.g., the attacker can create valid signatures only if it possesses the correct signing key. The attacker can obtain such keys only by observing or constructing them, never by guessing.

Our methodology allows us to prove that each implementation individually refines a particular role of an abstract protocol model. Since the security properties we prove about an abstract model are typically global, they hold only if each involved implementation refines one of the protocol roles. Next, we discuss limitations of this refinement proof, grouped by limitations of the methodology itself and limitations of our instantiation in Go.

The DIODON methodology requires a partitioning of a codebase into CORE and APPLICATION, while satisfying the syntactic restrictions for the CORE API specifications. This partitioning limits applicability, not soundness as the taint analysis checking I/O independence guides correct partitioning and fails otherwise. Additionally, DIODON requires the absence of undefined behavior in the codebase, which we prove for the CORE. However, this remains an assumption for the APPLICATION, which could be mitigated by performing an additional static analysis establishing this property. E.g., we could use ASTRÉE [36] for a subset of C and C++. Finally, we inherit the *pattern requirement* from Arquint et al. [14], which allows multiple terms to have the same byte-level representation in general, but requires a unique representation for terms corresponding to protocol messages.

Our instantiation of DIODON in Go uses several tools to discharge proof obligations, and we rely on the soundness of each tool: the abstract protocol model verifier, the auto-active program verifier, and the static analyses. The risk that any of these tools is unsound can be mitigated by choosing mature tools such as TAMARIN and GOBRA.

More specifically, the CORE's auto-active verification relies on trusted specifications for libraries, such as the I/O or cryptographic libraries that, e.g., consume I/O permissions or specify the cryptographic relations between input and output parameters. DIODON could be combined with verified libraries like EverCrypt [37] to reduce this trust assumption.

Furthermore, our taint analysis relies on the correct specification of secrets and I/O operations (we use an existing tool [35] to identify I/O operations). E.g., not treating the pre-shared key in the running example as a secret would allow us to perform I/O operations in the APPLICATION that depend on this key.

The employed static analyses assume that the entire codebase is free of data races and, thus, exhibits defined behavior only [38]. While we auto-actively prove race freedom for the CORE, this remains an assumption for the APPLICATION. Our implicit annotations clearly indicate where in the APPLICATION we rely on this assumption. Additionally, the static analyses do not soundly handle certain hard-to-analyze features such as the `unsafe` package (e.g., allowing arbitrary pointer arithmetic), `cgo` (i.e., the ability to invoke C functions), or reflection. We rely on the codebase not using them in a way that would invalidate the analysis results. DIODON could be extended by additional static analyses to reduce these assumptions, e.g., by performing a data race analysis and checking for uses of the `unsafe` and `cgo` packages and reflection. As such, these assumptions are not an inherent limitation of DIODON itself. We report case-studies-related limitations of the static analyses in Sec. 6.

5. Formalization and Soundness

We provide an overview of DIODON's soundness proof by highlighting its key steps and main ideas, and refer to the paper's extended version [1, App. A] for details. We split the soundness proof into two parts. First, we prove that we can soundly allow protocol-independent I/O operations in a codebase while assuming that we auto-actively verify an entire codebase. Second, we relax the requirement of verifying an entire codebase by showing that we can still construct a proof for the entire codebase in separation logic even though only the CORE is verified when certain side conditions are satisfied, most of which can be discharged by static analyses.

5.1. I/O Independence

As explained in Sec. 4.1, we execute a taint analysis to identify protocol-independent I/O operations in a codebase. Furthermore, we assume that we have a proof that a codebase c satisfies the Hoare triple $[\phi]\ \ c\ \ [\mathsf{true}]$, where ϕ is an I/O specification providing I/O permissions to execute protocol-*dependent* I/O operations. Protocol-independent I/O operations do not require an I/O permission and, thus, the codebase c may contain arbitrarily many protocol-independent I/O operations.

We prove in [1, App. A.1] that these independent I/O operations do not violate the security properties proven for the abstract protocol model by showing that the DY attacker can simulate these I/O operations. Hence, TAMARIN considers the existence of these I/O operations when proving security properties for a protocol model. More specifically, we extend the soundness proof by Arquint et al. [39] by an additional refinement step. This step defines a more refined protocol model by augmenting a given protocol model with rules enabling the protocol roles to perform protocol-independent I/O operations. We then prove that this refined protocol model refines the original protocol model by establishing a refinement relation that simulates protocol-independent I/O operations by actions of the DY attacker.

The key insight of the I/O independence proof is that we split the state of every protocol role instance into two parts. The first part is involved in the protocol's execution, e.g., keeping track of progress therein. The second part is involved only in the protocol-independent I/O operations. Our refinement relation leaves the former part unchanged while refining the latter to a corresponding state of the DY attacker. We map a transition in the augmented protocol model executing a protocol-independent I/O operation to a transition of the DY attacker performing the same I/O operation.

Therefore, we prove that the codebase c refines the abstract protocol model $\mathcal{R}$ even though the codebase contains *more* I/O operations than specified by the model for this protocol role, the difference being all protocol-independent I/O operations in c. We prove $(||_{rid}\, c(rid)) \;||\; \mathscr{O} \preccurlyeq_{\mathrm{t}} \mathcal{R}$ (simplified here, see [1, Thm. 3]), where we consider unboundedly many instances of the codebase c, parameterized by a run identifier rid, $\mathscr{O}$ denotes instances of all other verified protocol role implementations and the environment including the DY attacker, $||$ represents parallel composition, and $\preccurlyeq_{\mathrm{t}}$ expresses trace inclusion.

5.2. Combining Auto-Active Verification and Static Analyses

In the second part of the soundness proof [1, App. A.2], we want to prove the Hoare triple $[\phi]\ c\ [\mathsf{true}]$ for an entire codebase c, such that we can apply the soundness proof's first part, while auto-actively verifying only a small part of c, namely the CORE.

At a high-level, we prove a Hoare triple for each function f in the CORE using an auto-active verifier, i.e., $[P]\ f\ [C]$, where P and Q are f's pre- and postconditions. Since the APPLICATION calls multiple CORE functions and performs arbitrary I/O and memory operations in between, we have to prove that the APPLICATION establishes each invoked CORE function's precondition and has sufficient permissions to execute its memory operations in order that we obtain a proof in concurrent separation logic [25], [40] for the entire codebase c. While not all codebases c have such a proof, we prove its existence under certain side conditions that can be discharged using static analyses.

The proof sketch proceeds as follows. After presenting a simple programming language allowing us to focus on the main ideas, we present an algorithm inserting the implicit annotations for every statement in our language (cf. Sec. 4.3). The implicit annotations then allow us to define a program invariant that each statement in our language maintains, under some side conditions. By constructing derivation trees, we prove that each statement maintains the program invariant and make all side conditions apparent. We then prove lemmata showing that all these side conditions hold if our static analyses succeed on the codebase c. Finally, we compose the individual proof rules to construct a proof for the entire codebase c, i.e., $[\phi]\ c\ [\mathsf{true}]$, and discuss extensions of our soundness proof to support features commonly found in programming languages.

Programming language. We consider an imperative, concurrent, and heap-manipulating programming language in which the codebase c is written. To focus on the main ideas, we keep this language simple by restricting the codebase to a single run of a protocol role and omitting function boundaries, complex control flow, and callbacks; we separately discuss extensions lifting these restrictions at the end. Thus, we make calls to CORE functions first-class statements in our language. I.e., we consider $c := \mathbf{CoreAlloc}(\bar{e})$ to correspond to calling the CORE function allocating and initializing a new CORE instance (cf. `InitChannel` in the running example) and $\bar{r} := \mathbf{CoreApi_k}(c, \bar{e})$ to correspond to calling any other CORE function (indexed by k) on a CORE instance c with arguments $\bar{e}$ and results $\bar{r}$.

Implicit annotations. As explained in Sec. 4.3, we conceptually introduce ghost sets to track the permissions for heap locations owned by the APPLICATION. We recall that each thread has a set lhs for thread-local objects such as buffers and a set ihs for CORE instances. In addition, a set ghs contains heap locations that are shared between threads, and a flag used tracks whether the codebase has already used the I/O specification ϕ to create an instance of the CORE and, thus, a run of the corresponding protocol role.

While Fig. 7 demonstrates the implicit annotations to manipulate the mentioned ghost sets for a CORE API call, Fig. 9 presents the general algorithm $\mathbb{A}$ for inserting these implicit annotations for every statement in our language.

Writing to a heap location $(*x := e)$ in the case that x does not point to a thread-local heap location, i.e., $x \notin \mathsf{lhs}$, and our fork statement are of particular interest. For the former statement, the implicit annotations wrap the heap access in an atomic block to synchronize this access. This synchronization is sound as we assume that the APPLICATION is free of data races, and, thus, a linearization of accesses to a particular heap location exists (we assume that the underlying memory model is sequentially-consistent, like Go's memory model). Furthermore, we not only temporarily remove x from ghs but also move all heap locations that are transitively reachable from e from lhs to ghs because the write operation makes these heap locations become potentially accessible to other threads (via the heap location to which x points). Since our fork statement $(\mathbf{fork}\ (\bar{x})\ \{s\})$ passes the arguments $\bar{x}$ to a newly forked thread executing the statement s, we similarly move all heap locations that are transitively reachable from $\bar{x}$ from lhs to ghs. The newly forked thread then starts with its own, initially empty lhs and ihs as no thread-local heap locations and CORE instances exist yet.

Program invariant. With ghost sets and algorithm $\mathbb{A}$ defined, we can define a program invariant that holds at every original program point and that each statement maintains. More specifically, this program invariant is split into a thread-local (Π_l) and global (Π_g) part.

$$\Pi_l \triangleq (\bigstar_{l \in \mathsf{lhs}} \mathbf{acc}(l)) \star (\bigstar_{i \in \mathsf{ihs}} \mathbf{inv}(i))$$
$$\Pi_g \triangleq \mathbf{acc}(\mathsf{ghs}) \star (\bigstar_{g \in *\mathsf{ghs}} \mathbf{acc}(g)) \star \mathbf{acc}(\mathsf{used}) \star (\neg(*\mathsf{used}) \implies \phi)$$

$$
\begin{aligned}
\mathbb{A}(\textbf{skip}) &\rightsquigarrow \textbf{skip} \\
\mathbb{A}(x := \textbf{new}()) &\rightsquigarrow x := \textbf{new}();\ \mathsf{lhs} := \mathsf{lhs} \cup_{\textbf{nil}} \{x\} \\
\mathbb{A}(x := *e) &\rightsquigarrow \begin{cases} \mathsf{lhs} := \mathsf{lhs} \setminus \{e\};\ x := *e;\ \mathsf{lhs} := \mathsf{lhs} \cup_{\textbf{nil}} \{e\} & \text{if } e \in \mathsf{lhs} \\ \textbf{atomic}\ \{*\mathsf{ghs} := *\mathsf{ghs} \setminus \{e\};\ x := *e;\ *\mathsf{ghs} := *\mathsf{ghs} \cup_{\textbf{nil}} \{e\}\} & \text{otherwise} \end{cases} \\
\mathbb{A}(*x := e) &\rightsquigarrow \begin{cases} \mathsf{lhs} := \mathsf{lhs} \setminus \{x\};\ *x := e;\ \mathsf{lhs} := \mathsf{lhs} \cup_{\textbf{nil}} \{x\} & \text{if } x \in \mathsf{lhs} \\ \textbf{atomic}\ \{ *\mathsf{ghs} := *\mathsf{ghs} \setminus \{x\};\ \mathsf{lhs} := \mathsf{lhs} \setminus (\mathit{reach}(e) \cap \mathsf{lhs}); \\ \qquad\qquad *x := e;\ *\mathsf{ghs} := *\mathsf{ghs} \cup_{\textbf{nil}} \{x\} \cup (\mathit{reach}(e) \cap \mathsf{lhs})\} & \text{otherwise} \end{cases} \\
\mathbb{A}(c := \textbf{CoreAlloc}(\bar{e})) &\rightsquigarrow \textbf{atomic}\ \{*\mathsf{used} := \mathsf{true}\};\ \mathsf{lhs} := \mathsf{lhs} \setminus \bar{e};\ c := \textbf{CoreAlloc}(\bar{e});\ \mathsf{lhs} := \mathsf{lhs} \cup_{\textbf{nil}} \bar{e};\ \mathsf{ihs} := \mathsf{ihs} \cup_{\textbf{nil}} \{c\} \\
\mathbb{A}(\bar{r} := \textbf{CoreApi_k}(c, \bar{e})) &\rightsquigarrow \mathsf{ihs} := \mathsf{ihs} \setminus \{c\};\ \mathsf{lhs} := \mathsf{lhs} \setminus \bar{e};\ \bar{r} := \textbf{CoreApi_k}(c, \bar{e});\ \mathsf{lhs} := \mathsf{lhs} \cup_{\textbf{nil}} \bar{e} \cup \bar{r};\ \mathsf{ihs} := \mathsf{ihs} \cup_{\textbf{nil}} \{c\} \\
\mathbb{A}(s_1;\ s_2) &\rightsquigarrow \mathbb{A}(s_1);\ \mathbb{A}(s_2) \\
\mathbb{A}(\textbf{fork}\ (\bar{x})\ \{s\}) &\rightsquigarrow \mathsf{lhs} := \mathsf{lhs} \setminus (\mathit{reach}(\bar{x}) \cap \mathsf{lhs});\ *\mathsf{ghs} := *\mathsf{ghs} \cup_{\textbf{nil}} (\mathit{reach}(\bar{x}) \cap \mathsf{lhs});\ \textbf{fork}\ (\bar{x})\ \{\mathsf{lhs} := \emptyset;\ \mathsf{ihs} := \emptyset;\ \mathbb{A}(s)\}
\end{aligned}
$$

Figure 9. Algorithm $\mathbb{A}$ transforms a codebase by inserting ghost statements. We define this algorithm by cases, i.e., describe how $\mathbb{A}$ transforms each statement s to a statement s', written as $\mathbb{A}(s) \rightsquigarrow s'$. $\mathit{reach}(e)$ computes the set of transitively reachable heap locations from expression e. The set union operation ignores **nil**, as variables might be **nil**, i.e., $S_1 \cup_{\textbf{nil}} S_2 \triangleq (S_1 \cup S_2) \setminus \textbf{nil}$. This ensures that **nil** is never contained in any ghost set.

Since Sec. 4.3 already explains Π_l, we briefly explain Π_g here. As ghs and used are shared between threads, we treat them as pointers and Π_g specifies permissions for the corresponding heap locations. Furthermore, Π_g specifies full permissions for each heap location in $*$ghs and the I/O specification ϕ, unless ϕ has already been used to create a Core instance, i.e., $*$used is set to true. The last separating conjunct is sufficient for programs that create at most one core instance. We later discuss how this conjunct can be adapted to provide a family of I/O permissions, enabling arbitrarily many Core instances.

Proof rules. As shown in Fig. 10, we use algorithm $\mathbb{A}$ and the program invariant to define proof rules that have structurally identical conclusions, namely $\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s)\ \ [\Pi_l]$ for each statement s. This Hoare triple expresses that executing the statement s, transformed by algorithm $\mathbb{A}$, starting in a context satisfying Π_g and state satisfying Π_l, maintains this context and results in a state satisfying Π_l. We prove these rules sound in [1, App. A.2.2] by constructing a proof tree in concurrent separation logic [40]. Our proof relies on the side conditions ω (cf. Fig. 11) for statements other than sequential composition and fork. We require that the specification of Core functions satisfies our syntactic restrictions as mentioned in Sec. 4.3.

Discharging side conditions. Since the side conditions ω refer to containment of heap locations in certain ghost sets and disjointness of heap locations, we prove several lemmata in [1, App. A.2.3] to bridge the gap between these side conditions and the properties that a successful execution of our static analyses provides.

Our pointer analysis computes judgments $\mathrm{pts}(x)$ for a pointer x, where $a \in \mathrm{pts}(x)$ denotes that $*x$ may-alias any location allocated at site a. Applying our escape analysis delivers $\mathit{local}^p(x)$ guaranteeing that x points at program point p to a heap location that is accessible only by the current thread and not by any other thread. Finally, our pass-through analysis computes the judgments $\mathit{pt}^p_{\text{Core}}(a, \tau)$ and $\mathit{pt}^p_{\text{ret}}(a, \tau)$ denoting that a heap location allocated at allocation site a passed through (pt) the return argument of a $c := \textbf{CoreAlloc}(\bar{e})$ statement and through one of the return arguments $\bar{r}$ of a $\bar{r} := \textbf{CoreApi_k}(c, \bar{e})$ statement, respectively, between site a and program point p on trace τ.

More specifically, we use our static analyses to obtain the judgments for each statement in the codebase as shown in Def. 1 and prove in [1, Lemma 10] that these judgments are sufficient to discharge the side conditions ω. While the static analyses run on the original program, before applying algorithm $\mathbb{A}$, the runtime (non-ghost) behavior is identical and so the judgments apply equally to the codebase after transformation. This proof relies on our assumptions, i.e., the Application's data race freedom, the syntactic restrictions for Core functions, and the soundness of our static analyses.

Definition 1 (Static analyses for Diodon)**.** *In Diodon, we execute the static analyses on a codebase to obtain the following judgments for every statement s at label ℓ therein, denoted as $j(s^\ell)$.*

$$
\begin{aligned}
j(x := *e) &\triangleq \forall a, \tau.\, a \in \mathit{pts}(e) \implies \mathit{am}^{\text{pre-}\ell}_\tau(a) \\
j(*x := e) &\triangleq \forall a, \tau.\, a \in \mathit{pts}(x) \implies \mathit{am}^{\text{pre-}\ell}_\tau(a) \\
j(c := \textbf{CoreAlloc}(\bar{e})) &\triangleq \mathit{disjoint}_{as}(\bar{e}) \wedge \mathit{local}^\ell_{am}(\bar{e}) \\
j(\bar{r} := \textbf{CoreApi_k}(c, \bar{e})) &\triangleq \mathit{disjoint}_{as}(\bar{e}) \wedge \mathit{local}^\ell_{am}(\bar{e}) \\
&\qquad \wedge \mathit{local}^\ell_{\mathit{Core}}(c) \wedge \mathit{local}^\ell_{\mathit{ret}}(\bar{r})
\end{aligned}
$$

where

$$
\begin{aligned}
\mathit{disjoint}_{as}(\bar{e}) &\triangleq \forall i, j.\, 0 \le i < j < \mathit{len}(\bar{e}) \implies \\
&\qquad \mathit{pts}(\bar{e}[i]) \cap \mathit{pts}(\bar{e}[j]) = \emptyset \\
\mathit{local}^\ell_{am}(\bar{e}) &\triangleq \forall e, h, \tau.\, e \in \mathit{set}(\bar{e}) \wedge \\
&\qquad \mathit{as}_\tau(h) \in \mathit{pts}(e) \implies \\
&\qquad \mathit{local}^{\text{post-}\ell}(e) \wedge \mathit{am}^{\text{pre-}\ell}_\tau(h) \\
\mathit{local}^\ell_{\mathit{Core}}(c) &\triangleq \forall h, \tau.\, \mathit{as}_\tau(h) \in \mathit{pts}(c) \implies \\
&\qquad \mathit{local}^{\text{pre-}\ell}(c) \wedge \mathit{pt}^{\text{pre-}\ell}_{\mathit{Core}}(h, \tau) \\
\mathit{local}^\ell_{\mathit{ret}}(\bar{r}) &\triangleq \forall r, \tau.\, r \in \mathit{set}(\bar{r}) \implies \mathit{local}^{\text{post-}\ell}(r)
\end{aligned}
$$

pre-ℓ and post-ℓ refer to the program points immediately preceding and following the statement at label ℓ, respectively. $\mathit{as}_\tau(h)$ returns heap location h's allocation site for a particular program trace τ. $\mathit{am}^{\text{pre-}\ell}_\tau(a)$ denotes that an allocation

$$\frac{\omega(s_{\text{simple}})}{\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s_{\text{simple}})\ \ [\Pi_l]}\ (\textsc{Simple}) \qquad \frac{\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s_1)\ \ [\Pi_l] \quad \Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s_2)\ \ [\Pi_l]}{\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s_1; s_2)\ \ [\Pi_l]}\ (\textsc{Seq}) \qquad \frac{\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(s)\ \ [\Pi_l]}{\Pi_g \vdash [\Pi_l]\ \ \mathbb{A}(\textbf{fork}\ (\bar{x})\ \{s\})\ \ [\Pi_l]}\ (\textsc{Fork})$$

Figure 10. Proof rules. s_{simple} ranges over all *simple* statements; s, s_1, and s_2 range over all statements. ω denotes a statement's side conditions (cf. Fig. 11).

$$\begin{aligned}
\omega(x := *e) &\triangleq e \in \mathsf{lhs} \cup *\mathsf{ghs} \\
\omega(*x := e) &\triangleq x \in \mathsf{lhs} \cup *\mathsf{ghs} \\
\omega(c := \mathbf{CoreAlloc}(\bar{e})) &\triangleq *\,\mathsf{used} = \mathsf{false} \wedge \\
&\quad (\mathit{set}(\bar{e}) \setminus \mathbf{nil}) \subseteq \mathsf{lhs} \wedge \\
&\quad \mathit{disjoint}(\bar{e}) \\
\omega(\bar{r} := \mathbf{CoreApi_k}(c, \bar{e})) &\triangleq (\mathit{set}(\bar{e}) \setminus \mathbf{nil}) \subseteq \mathsf{lhs} \wedge \\
&\quad \mathit{disjoint}(\bar{e}) \wedge \\
&\quad (c \in \mathsf{ihs} \vee c = \mathbf{nil})
\end{aligned}$$

Figure 11. Side conditions for our statements, which are amenable to static analyses. ω evaluates to true for all statements not listed above and $\mathit{set}(l)$ returns the set of elements in list l. We implicitly refer to variables' values, e.g., $v \in S$ denotes that the value of variable v is contained in set stored in variable S as opposed to the variables' syntactical representation.

site a is Application-managed on trace τ at the program point pre-ℓ. An allocation site a is Application-managed if a is either within the Application or the Core passed the corresponding heap location as a return argument to the Application. We decide whether a is Application-managed by checking its location in the program text and running our pass-through analysis.

Proof construction. While we showed that we can compose the proof rules in Fig. 10 and discharge their side conditions ω, it remains to show that we initially establish the global context Π_g and the local program invariant Π_l, such that we obtain a proof for the entire codebase c. We close this gap in Cor. 1.

Corollary 1 (Proof construction)**.** *Successfully executing Diodon's static analyses on a codebase c and the Core's auto-active verification combined with our assumptions allow us to construct a separation logic proof for c.*

$$\begin{aligned}
&\textit{If}\ \ \forall s, k.\, s \in c \wedge j(s) \wedge \\
&\quad \Big(s = c := \mathbf{CoreAlloc}(\bar{e}) \implies \\
&\quad\ \Pi_g \vdash [P_{CoreAlloc}(\bar{e})]\ \ s\ \ [Q_{CoreAlloc}(c, \bar{e})]\Big) \wedge \\
&\quad \Big(s = \bar{r} := \mathbf{CoreApi_k}(c, \bar{e}) \implies \\
&\quad\ \Pi_g \vdash [P_{CoreApi_k}(c, \bar{e})]\ \ s\ \ [Q_{CoreApi_k}(c, \bar{e}, \bar{r})]\Big), \\
&\textit{then}\ \ \mathbf{emp} \vdash [\phi]\ \ s_{init}; \mathbb{A}(c)\ \ [\mathsf{true}]
\end{aligned}$$

where s_{init} is ghost code creating and initializing the thread-local ghost sets lhs *and* ihs *for the main thread, as well as the global ghost set* $*$ghs *and the ghost flag* $*$used.

We show that we obtain the desired proof for the entire codebase, namely that the codebase satisfies the I/O specification ϕ expressed as the Hoare triple $\mathbf{emp} \vdash [\phi]\ \ s_{\text{init}}; \mathbb{A}(c)\ \ [\mathsf{true}]$. This Hoare triple relies on s_{init} that initializes lhs, ihs, and $*$ghs to empty sets, as well as sets the ghost flag $*$used to false. s_{init} is similar in spirit to the ghost statements that algorithm $\mathbb{A}$ inserts as these statements are necessary to construct a proof for the codebase c. Cor. 1's premise states that our static analyses succeed on the codebase c, such that we obtain $j(s)$ for each statement s therein, and that we prove a Hoare triple for each Core function satisfying the syntactic restrictions.

We combine the proof for the entire codebase that we obtain from Cor. 1 with the result of Sec. 5.1 to obtain Diodon's overall soundness result. This result states that successfully executing our static analyses on codebase c and auto-actively verifying its Core suffices to prove that the traces of executing c together with other verified implementations and the environment are contained in the traces described by the abstract protocol model.

Theorem 2 (Overall soundness)**.** *Suppose Cor. 1's antecedent holds, and we established I/O independence. Then, $(||_{rid}\ c(rid)) \,||\, \mathscr{O} \preccurlyeq_{\mathrm{t}} \mathcal{R}$ (simplified) holds.*

Proof sketch. By Cor. 1 and [1, Thm. 3] (cf. Sec. 5.1). □

5.2.1. Limitations. Our formalization defines a simple programming language to focus on the main ideas of our soundness proof and to show that successfully executing our static analyses discharges all side conditions. We believe this language covers the most critical features like heap manipulations and concurrency as these features are relevant for the results of our static analyses. In addition, we abstract each function making up the Core's API to a dedicated statement in our language, and assume that the specification of each such function satisfies our syntactic restrictions [1, Asm. 4]. However, there is a slight risk that this language misses Go features that would be a threat to soundness such as function boundaries, complex control flow, and callbacks; the former two features would be straightforward to add, and we discuss in Sec. 5.2.2 how to add the latter.

To prove a Hoare triple for the entire codebase, we assume that the Application is free of crashes [1, Asm. 2] and data races [1, Asm. 3]. While our soundness proof does not make any statement in the case that the program crashes, our compositional proof informally guarantees that the trace inclusion holds for the program's prefix up to the program point at which a crash occurs, such that the crash freedom assumption could be dropped, which we leave to future work. However, data race freedom remains an assumption; more generally, we assume the absence of undefined behavior for programming languages other than Go and our formalized one. This assumption can be mitigated by performing additional static analyses.

5.2.2. Extensions. Having covered the main soundness result, we discuss two extensions to bridge the gap to realistic applications of Diodon as used in our case studies. We first lift the restriction of at most one Core instance to allow a codebase to create unboundedly many Core instances. Second, we allow the Core to invoke callbacks into the Application and discuss the side conditions that arise by this extension.

Unboundedly many Core instances. So far, our global program invariant Π_g contains the separating conjunct

$$\mathbf{acc}(\mathsf{used}) \star (\neg(*\mathsf{used}) \implies \phi).$$

As explained in [1, App. A.1], each execution of a protocol role is parameterized by a unique rid. I.e., ϕ and all I/O permissions that ϕ internally provides are parameterized by rid and, thus, are not interchangeable but specific to a particular rid. Hence, we can change the separating conjunct stated above to

$$\mathbf{acc}(\mathsf{used}) \star (\forall rid \notin *\mathsf{used} \implies \phi(rid))$$

providing a family of I/O permissions, where used points to a ghost set containing the rids that have already been used. In addition, we adapt the entire program's precondition from ϕ to $\forall rid.\, \phi(rid)$ and change the translation $\mathbb{A}(c := \mathbf{CoreAlloc}(\bar{e}))$ to, first, pick a fresh rid' such that $rid' \notin *\mathsf{used}$ and, second, adding rid' to $*\mathsf{used}$. Picking such a fresh rid' is always possible since rid ranges over $\mathbb{N}$.

Adding callbacks to the Core. So far, we have treated the statements $\mathbf{CoreAlloc}(\bar{e})$ and $\bar{r} := \mathbf{CoreApi_k}(c, \bar{e})$ as atomic statements in our language. These two statements are internally implemented as sequences of statements, which we hereafter call Core statements. As these statements constitute the Core, we auto-actively prove that a particular postcondition holds when control transfers back to the Application after fully executing these statements.

In the presence of callbacks, however, calling into the Core becomes non-atomic and control flow might transfer to the Application before reaching the post-state for which we know that the postcondition holds. We can treat callbacks as temporarily pausing the execution of these auto-actively verified Core statements to (sequentially) execute some statements belonging to the Application before eventually resuming execution of Core statements.

With respect to algorithm $\mathbb{A}$ and the ghost sets, interrupting the execution of Core statements to execute certain Application statements s_{app} means that heap locations on which the Core statements operate are missing from the ghost sets while executing s_{app} as we remove them from the ghost sets before executing Core statements and put them back only after the Core statements' postcondition holds. Missing permissions include both arguments $\bar{e}$ and the Core instance c. Therefore, we have to make sure that s_{app} neither accesses heap locations to which $\bar{e}$ points nor invokes API calls on the Core instance c as the Core invariant might not hold.

We can lift these restrictions by introducing additional proof obligations for the auto-active verification. More specifically, if we auto-actively prove that the Core statements satisfy a particular precondition for the callback, then we can update the ghost sets accordingly. E.g., such a precondition can specify permissions for heap locations passed to the callback or that the Core invariant holds.

In our SSM Agent case study (Sec. 6.1), we make use of these proof obligations for the callback delivering incoming messages to the Application as we specify that the Core transfers permission for the incoming message to the Application. Conceptually, this allows us to add the corresponding heap location to lhs before executing the statements constituting the callback because the auto-active proof guarantees that no statement in the Core thereafter accesses this heap location.

For our case studies, it was not necessary to transfer permissions from a callback back to the Core via a callback's postcondition. Extending Diodon to allow such permission transfers would require an analysis of the callback showing that the Application possesses these permissions while executing the callback and that the corresponding heap locations do not get accessed by the Application after the callback returns.

6. Case Studies

To demonstrate that Diodon scales to large codebases, we evaluate it on the AWS Systems Manager Agent (SSM Agent) [20], a 100k+ lines of code (LoC) production Go codebase. Furthermore, we apply Diodon to a small implementation of the signed Diffie–Hellman (DH) key exchange to showcase that our methodology applies to other implementations and coding styles.

6.1. AWS Systems Manager Agent

The AWS Systems Manager Agent (SSM Agent) [20] provides features for configuring, updating, and managing Amazon EC2 instances, and is widely used by AWS customers. A fork of this codebase implements a novel protocol which enables encrypted interactive shell sessions with remote host machines, similar to the Secure Shell (SSH) protocol, without needing to open inbound ports or manage SSH keys. This protocol establishes these shell sessions with a handshake protocol involving a signed elliptic-curve DH key exchange to derive sessions keys that are subsequently used in the transport phase to encrypt the shell commands and their results.

We apply Diodon by first modeling the protocol in Tamarin and proving secrecy and injective agreement (Sec. 6.1.1). Second, we partition the codebase into the code implementing the protocol (the Core) and the remaining codebase (the Application), and prove I/O independence (Sec. 6.1.2). Third, we auto-actively verify the Core using Gobra to prove that the Core refines the SSM Agent's role (Sec. 6.1.3). Finally, we apply the automatic static analyses Argot [24] to discharge the assumptions within the Application on which the auto-active proof relies (Sec. 6.1.4).

Fig. 12 overviews each tool's execution time, for which we use the 10 % Winsorized mean of the wall-clock runtime

	Tool	Proof Effort	Execution Time
Protocol Model	Tamarin	<2 pms	3.30 min
Core Refinement	Gobra	<3 pms	1.17 min
I/O Independence	Argot	<0.5 pm	0.48 min
Core Assumptions	Argot	<1.5 pms	2.12 min

Figure 12. Execution time for running each tool on the SSM Agent codebase and approximate proof effort in person-months (pms) for creating a protocol model, adding specifications, and adapting the Argot analyses, respectively.

across 10 verification runs, measured on a 2023 Apple MacBook Pro with M3 Pro processor and macOS 15.6.

6.1.1. Protocol Model. We model in Tamarin the security protocol for establishing a remote shell session between an SSM Agent running on an EC2 instance and an AWS customer. The protocol offloads all signature operations to the AWS Key Management Service (KMS) [41] such that neither protocol role has to manage their own signing keys. We model the connections to KMS as secure channels. Furthermore, the SSM Agent sends the asymmetrically-encrypted session keys to a trusted third party to monitor the transmitted shell commands should this be necessary for regulatory reasons. We provide the full description of the protocol in the extended version of this paper [1, App. B].

In Tamarin, we prove secrecy for the two symmetric session keys, i.e., the attacker does not learn these keys unless the SSM Agent's or customer's signing key or the monitor's secret key is corrupted. Additionally, we prove that the SSM Agent injectively agrees with the customer, and vice versa, on their identities and the session keys, unless one of the three aforementioned corruption cases occurs.

The abstract protocol model amounts to 319 LoC and is automatically verified by Tamarin 1.10.0 in 3.30 min using an auxiliary oracle consisting of 75 lines of Python code.

6.1.2. Proving I/O Independence. We perform a taint analysis to prove I/O independence. We configure the taint analysis to consider all generated elliptic-curve secret keys as sources of protocol secrets. We assume that only the Core uses the SSM Agent's signing keys and do not treat KMS responses as taint sources because KMS only sends us signatures and never key material. As described in Sec. 4.1, we use Capslock's capability information to automatically configure the taint analysis' sinks.

We annotated some branching operations, instructing the taint analysis to ignore that the branch condition is tainted. We identified two classes of such branching operations. The first class is justified by cryptography. E.g., we allow branching on the success of decrypting a transport message because leakage is minimal. The second class results from imprecisions of the taint analysis and corresponds to false positives, i.e., the analysis deems a branch condition tainted although it is not. To avoid another source of false positives, we configured the taint analysis to ignore taint escaping the current thread (which would otherwise always lead to errors). Such cases could be handled precisely by marking certain struct fields as potentially storing concurrently-accessed, tainted data, such that the analysis can track the taint.

The taint analysis succeeds for the SSM Agent codebase in 29.0 s, proving that there are no taint flows.

6.1.3. Core Refinement. The SSM Agent contains a Go package called `datachannel` that implements the protocol. More precisely, this package contains struct definitions that together store all necessary internal state. Additionally, this package exposes publicly accessible functions to initialize the internal state, perform a handshake, and send a payload, which internally rely on several private functions. We refer to these struct definitions and functions as the Core. For backward compatibility, the Core also implements a legacy protocol; we assume that this legacy protocol is disabled.

Implementation. Each Core instance corresponds to one run of the protocol with a particular AWS customer. During initialization of a new Core instance, the Core starts a new thread, responsible for receiving and processing incoming packets for this protocol run, similar to the running example. If an incoming packet contains a transport phase payload, this payload is delivered by a callback to the Application. Thus, the Core uses two different threads, one for sending messages and another one for receiving messages, which both operate on shared state. This shared state keeps track of the progress within the protocol and the secret data involved in the protocol, such as the elliptic-curve DH points and the resulting session keys.

Since the shared state is modified during the handshake, accesses must be synchronized to avoid data races. Hence, the Core employs Go channels, i.e., lightweight message passing, to signal a transfer of the shared state's ownership from one thread to another. During the handshake phase, exclusive ownership is transferred such that the threads have synchronized write access to the shared state. Afterwards, the shared state, which includes the established session keys, is used in a read-only way permitting both threads to concurrently read the shared state while sending and receiving transport messages.

Auto-active refinement proof. We verify the Core using Gobra, which proves that the Core refines the Tamarin model's SSM Agent role. This proof encompasses safety, i.e., we prove that the Core does not crash and has sufficient permissions for every heap access, thus, guaranteeing absence of data races. In particular, this forces us to reason precisely about the accesses to shared state that the two threads within the Core perform.

Due to the intricate interplay of these threads, the resulting safety proof is substantial and requires Gobra's expressivity. We isolate and axiomatize operations that Gobra does not yet support such as simultaneously receiving on multiple channels and functionally reasoning about serialization and deserialization. For the purpose of the proof, we treat the Core as a state machine consisting of 12 different states. This allows us to refer to these states in the Core's invariant and precisely express for each state the permissions and progress w.r.t. the abstract protocol model.

Although the entire complexity of the proof is encapsulated in the Core's invariant, function calls to the Core

must respect its state machine. To avoid exposing the state machine in these functions' preconditions and imposing additional restrictions on callers, we slightly changed the implementation to perform a dynamic check consisting of a comparison with **nil** and a single integer comparison ensuring that the state machine is in a correct state; otherwise, these CORE functions return a descriptive error. Thus, the CORE functions' specifications are similar to those of our running example, i.e., mention only the invariant and specify permissions for parameters without referring to the state machine. While most parameters are of primitive type or shallow, there are a few non-shallow input parameters, which the CORE treats as opaque. Similarly, the callback from the CORE to the APPLICATION delivers a non-shallow struct for which we ensure that the CORE passes permissions for all transitively reachable heap locations to the APPLICATION.

We prove safety and refinement of the CORE in 1.17 min for 749 lines of code requiring 3825 lines of specification and proof annotations; 1064 thereof are related to the I/O specification and generated automatically by TAMARIN.

6.1.4. Analyzing the APPLICATION. The auto-active proof for the CORE relies on callers satisfying the specified preconditions, which we establish using a combination of static analyses. We implemented automatic checks as described in Sec. 4.3 for conditions (C1)–(C4) and (C6)–(C8). Condition (C5) requires a more precise call graph than is currently available in our tool and is, thus, left as future work.

We implemented our analyses by forking and extending the existing ARGOT tool. Most of our analyses are obtained by interpreting the output of an existing analysis; e.g., the parameter alias check uses the off-the-shelf pointer analysis to show parameters do not alias one another.

For some conditions, our static analyses were not able to validate the APPLICATION due to tool limitations. For example, the escape analysis cannot reason about which fields are accessed after a struct escapes. This can cause the tool to raise alarms when a struct stores a CORE instance in a field. We found it was straightforward to rewrite the CORE and APPLICATION to eliminate these failures. For example, the struct leakage can be fixed by moving the relevant field accesses before thread creation, so that the new thread has access only to the values of those fields and not the entire struct, and by extension the CORE instance.

By running our escape analysis, we observed that CORE instances escape the thread in which they are created because the APPLICATION creates a closure that closes over an object that points to a CORE instance. This capture is incidental in that the closure does not access the captured CORE instance, which we verified by manual inspection. This capture can be eliminated by rewriting the application to reference only the state necessary in this closure, rather than the full object. This change would result in a more defensive implementation by reducing the scope of possibly concurrent accesses.

Our pass-through analysis is a prototype that succeeds on our second case study. However, for the SSM AGENT, we obtain false positives due to allocations in functions called from both CORE and APPLICATION, which could be addressed by adding calling context information.

Some CORE functions take a pointer to a logger object as a parameter, which is internally thread-safe and shared between threads. We can safely ignore escape errors due to these parameters because the CORE does not access any memory of the logger object; the pointer is just used as an opaque reference to invoke log functions that are part of the APPLICATION.

In summary, this case study demonstrates that DIODON allows one to obtain strong security guarantees for a production codebase that was not designed with formal verification in mind. The remaining limitations (manual overrides of false positives in the static analyses, checking condition (C5), extremely lightweight dynamic checks enforcing non-nilness and correct ordering of API calls, and minor code changes) are modest compared to the complexity of the overall verification challenge and we conjecture that we can lift them by employing more precise static analyses.

6.2. Signed Diffie–Hellman (DH) Key Exchange

We also apply our approach to a codebase employing *inverted I/O*, i.e., has a CORE that only produces and consumes byte arrays corresponding to protocol messages while the APPLICATION performs all I/O operations. We adapted the TAMARIN model and Go implementation of the signed DH key exchange from Arquint et al. [14] and extended both by a transport phase that uses the established session key to send and receive unboundedly many payloads. TAMARIN verifies the abstract model with 177 lines of code in 3.2 s while GOBRA verifies the CORE consisting of 178 lines of code in 14.2 s requiring 1726 lines of specification (LOS). Executing all static analyses including the taint analysis takes 9.7 s.

This case study clearly exhibits the concept of virtual I/O. The CORE performs a virtual input operation for messages that the APPLICATION received from the network and forwarded to the CORE. Similarly, we perform a virtual output operation for every message that the CORE produces before returning this message to the APPLICATION. Therefore, we prove that the TAMARIN model permits sending this message and in return, we sanitize the message from a taint analysis' perspective such that the APPLICATION can send the message without causing a false-positive taint flow.

DIODON separates the justification of sending a particular message from the actual I/O operation. This is important for tackling realistic codebases because identifying the actual send operation in a call stack is typically difficult as a message passes through several functions that, e.g., add additional protocol headers before a message is handed to the network interface controller.

6.3. Discussion

Our evaluation demonstrates that DIODON enables us to efficiently prove that an entire codebase refines a protocol model and therefore is secure. To obtain the security properties as proven in TAMARIN for a deployment of this protocol,

we have to prove the implementations of all other protocol roles analogously against the same model using Diodon.

As shown in Fig. 12, the efforts for applying Diodon to the SSM Agent is manageable. Thanks to I/O independence, the Tamarin model is concise and can focus on the relevant interactions between the protocol roles. In addition, I/O independence allows us to apply automatic static analyses at the code-level to reason about all protocol-irrelevant I/O operations. This contrasts existing approaches that would auto-actively verify the entire codebase and prove that every I/O operation is explicitly permitted by the model, which is completely impractical for this codebase.

To evaluate Diodon's effectiveness at preventing security vulnerabilities, we deliberately introduce bugs in our case studies. E.g., our taint analysis correctly fails if the Core's internal state, which includes the established session keys, is logged after the handshake. Additionally, sending the DH secret key in plaintext correctly results in Gobra failing to prove refinement w.r.t. the abstract protocol model. The tools' execution time in the presence of these bugs remains comparable to that for the secure implementations.

By applying Diodon we not only obtain security properties for the SSM Agent codebase but we also discovered and fixed bugs along the way. Tamarin allowed us to quickly locate and fix a Mallory-in-the-middle (MITM) attack in an earlier and unreleased version of the protocol, which is possible if the intended recipient's identity is omitted in the signatures (sig_x and sig_y in [1, Fig. 36]). On the code level, we identified and fixed a potential data race in an earlier and unreleased version of the Core caused by insufficient synchronization between the two threads that send and receive handshake messages. We uncovered this data race because completing the safety proof for the Core's earlier version is not possible as an additional synchronization point is necessary to transfer separation logic permissions between these threads. This demonstrates the power of applying formal methods because detecting this data race with testing techniques would require to precisely time the reception of a handshake message such that the faulty memory access occurs and, thus, can be observed.

7. Related Work

Much prior work on verifying security protocols exists and surveys [42], [43], [44] provide an extensive overview. Hence, we focus on approaches for verifying security properties for *implementations* and their applicability to large and real-world codebases. We end by comparing Diodon to approaches based on dynamic verification.

Implementation and model generation. One approach to obtain verified protocol implementations generates secure-by-construction implementations from an abstract model, e.g., [45], [46], [47], [48], [49]. However, these implementations typically show subpar performance and optimizing them by hand or integrating them into a larger codebase forfeits proven security properties. Thus, the abstract model has to cover the entire functionality (which we do not require). While OwlC [49] enables embedding a generated implementation into a codebase, they rely on the Rust type system to shield secrets from the rest of the codebase. We avoid this restriction by checking I/O independence for the Application. In addition, they use a Rust type to ensure that a previously established session key is used during a transport phase. Instead, we use the Core invariant to maintain separation logic properties between Core API calls, which is more expressive.

An alternative approach extracts an abstract model from an implementation, e.g., [5], [7], [50], [51], [52]. However, for this extraction to work, an implementation typically has to follow restrictive coding disciplines such that relevant protocol steps can be identified and extracted. To achieve isolation between a verified component and potentially malicious code, Kobeissi et al. [7] build on process isolation provided by operating systems and, thus, require verifying the entire critical process. We cannot adopt this approach because it requires changing the codebase heavily to split it into several processes and results in an, for our use case, unacceptable overhead, since each process includes its copy of the Go runtime and the Go standard library. Bhargavan et al. [5] impose substantial restrictions on the API of verified code, e.g., disallowing state preservation between API calls. Codebases do not normally satisfy these restrictions, including all our case studies. E.g., they use a session key for sending a transport message in one API call that was established during the handshake, i.e., a previous API call.

Existing implementations. Dupressoir et al. [13] embed a *trace* storing relevant protocol operations as an auxiliary data structure for proof purposes into C code implementing a security protocol. This auxiliary data structure is removed before compilation and does not incur any runtime overhead while enabling reasoning about weak secrecy and non-injective agreement. Arquint et al. [15] generalize this approach to separation logic, making it applicable to a wide range of programming languages and supporting stronger security properties such as forward secrecy and injective agreement. However, both approaches require a sufficiently strong invariant over this trace to prove security properties. To avoid such a trace invariant, Arquint et al. [14] prove security properties on the level of an abstract model using Tamarin's proof automation and prove that an implementation refines the abstract model. All three approaches require verifying the entire codebase using an auto-active verifier (which we do not). We build on the latter approach and, to the best of our knowledge, are the first to relax this requirement to verifying just the Core and reason about the Application using lightweight static analyses.

Dynamic verification. Several approaches employ dynamic checks at runtime to allow for partially verified codebases. Agten et al. [53] target single-threaded C code and generate runtime checks at the boundary between verified and unverified code to test that the verified code's specification holds. To detect violations of properties expressed in separation logic such as ownership (via permissions) and aliasing, this approach tracks the heap locations accessed by the verified codebase at runtime and computes cryptographic hashes

thereover. It remains unclear whether these checks only at the boundary remain sufficient when targeting concurrent codebases or whether the runtime overhead increases further. To avoid tracking heap locations at runtime, Ho et al. [54] copy all heap data at this boundary to rule out aliasing.

Gradual verification (e.g., [55], [56]) combines auto-active verification with dynamic checks but aims at helping the proof developer by allowing incomplete specifications. I.e., gradual verification enables incremental verification where each function's specification is extended over time to eventually obtain a fully specified and verified codebase. However, as long as a codebase is not fully specified and verified, gradual verification requires tracking heap locations at runtime, which results in noticeable runtime overhead.

SCIO* [57] is an F* transpiler that injects dynamic checks not only at the boundary between verified and unverified code but also at call sites of I/O operations. While they can enforce access policies for I/O operations, it remains unclear how this approach extends to cryptographic message payloads. To be applicable in our context, we would need to dynamically check whether a message sent by our Application is indeed protocol-irrelevant and, thus, does not contain any secrets from the Core—not even in encrypted form. Like our work, SecRef* [58] considers the problem of verifying only a subset of a codebase due to the otherwise prohibitive proof effort. While they also allow pre- and postconditions at the boundary between verified and unverified code, they rely on dynamic checks to enforce these conditions for heap locations accessible by unverified code. For a verified component like our Core, this means that they check the entire invariant at runtime for each API call (which we do not), as they treat the unverified code as potentially modifying a Core instance's entire state. By targeting a single-threaded language (F*), SecRef* does not have to consider concurrent memory accesses (which we do).

By contrast, Diodon performs only extremely lightweight dynamic checks enforcing non-nilness and correct ordering of API calls, and checks all other constraints *statically* to avoid runtime overhead while simultaneously requiring minimal code changes.

8. Conclusions

We present Diodon, a novel methodology to scale verification of security protocol implementations to large existing codebases by symbiotically combining powerful auto-active verification of a relatively small part of the codebase with static analyses that scale to the entire codebase. Since Diodon is not inherently limited to Go, future work could apply it to codebases written in, e.g., Rust and C. Adapting Diodon to Rust would remove several static analyses due to the strong type system, and C has a variety of static analyses and auto-active verifiers that can be used. Orthogonally, extending Diodon with further static analyses, such as for nilness, would allow us to pass more guarantees from the Application to the Core. We hope our work spurs both practical and theoretical understanding of how to soundly combine proof systems of different expressive power.

Acknowledgments

We thank the Werner Siemens-Stiftung (WSS) for their generous support of this project, Michael Hicks, K. Rustan M. Leino, and Margarida Ferreira for feedback on drafts of this paper, Christoph Sprenger and Joseph Lallemand for helpful discussions, and the anonymous reviewers for their insightful comments. Parts of this work were conducted by the first author during internships at AWS. Some authors are developers of Gobra or Argot, which we disclose as potential non-financial interests.

References

[1] L. Arquint, S. Kishor, J. R. Koenig, J. Dodds, D. Kroening, and P. Müller, "The secrets must not flow: Scaling security verification to large codebases (extended version)," *CoRR*, vol. abs/2507.00595, 2025.

[2] B. Schmidt, S. Meier, C. Cremers, and D. A. Basin, "Automated analysis of Diffie–Hellman protocols and advanced security properties," in *CSF*. IEEE, 2012, pp. 78–94.

[3] S. Meier, B. Schmidt, C. Cremers, and D. A. Basin, "The TAMARIN prover for the symbolic analysis of security protocols," in *CAV*, ser. LNCS, vol. 8044. Springer, 2013, pp. 696–701.

[4] B. Blanchet, "An efficient cryptographic protocol verifier based on Prolog rules," in *CSFW*. IEEE, 2001, pp. 82–96.

[5] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," in *S&P*. IEEE, 2017, pp. 483–502.

[6] D. A. Basin, R. Sasse, and J. Toro-Pozo, "The EMV standard: Break, fix, verify," in *S&P*. IEEE, 2021, pp. 1766–1781.

[7] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *EuroS&P*. IEEE, 2017, pp. 435–450.

[8] D. A. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5G authentication," in *CCS*. ACM, 2018, pp. 1383–1396.

[9] C. Cremers and M. Dehnel-Wild, "Component-based formal analysis of 5G-AKA: Channel assumptions and session confusion," in *NDSS*. The Internet Society, 2019.

[10] CVE, "CVE-2021-40823," 2021. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2021-40823

[11] ——, "CVE-2022-22805," 2022. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2022-22805

[12] ——, "CVE-2022-22806," 2022. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2022-22806

[13] F. Dupressoir, A. D. Gordon, J. Jürjens, and D. A. Naumann, "Guiding a general-purpose C verifier to prove cryptographic protocols," in *CSF*. IEEE, 2011, pp. 3–17.

[14] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. A. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations," in *S&P*. IEEE, 2023, pp. 1077–1093.

[15] L. Arquint, M. Schwerhoff, V. Mehta, and P. Müller, "A generic methodology for the modular verification of security protocol implementations," in *CCS*. ACM, 2023, pp. 1377–1391.

[16] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Trans. Inf. Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[17] CVE, "CVE-2024-47083," 2024. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-47083

[18] ——, "CVE-2023-6746," 2023. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2023-6746

[19] K. R. M. Leino and M. Moskal, "Usable auto-active verification," in *Usable Verification Workshop*, 2010.

[20] Amazon Web Services, Inc., "Working with SSM Agent," 2023. [Online]. Available: https://docs.aws.amazon.com/systems-manager/latest/userguide/ssm-agent.html

[21] L. Arquint, S. Kishor, J. R. Koenig, J. Dodds, D. Kroening, and P. Müller, "The secrets must not flow: Scaling security verification to large codebases (artifact)," Sep. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.17099763

[22] ——. (2025, Oct.) The secrets must not flow: Scaling security verification to large codebases. Artifact repository containing the protocol models, the forked SSM Agent's codebase, a DH implementation codebase, and the static analysis tools. [Online]. Available: https://github.com/viperproject/diodon-artifact

[23] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of Go programs," in *CAV*, ser. LNCS, vol. 12759. Springer, 2021, pp. 367–379.

[24] AWS Labs, "Argot," 2024. [Online]. Available: https://github.com/awslabs/ar-go-tools

[25] J. C. Reynolds, "Separation Logic: A logic for shared mutable data structures," in *LICS*. IEEE, 2002, pp. 55–74.

[26] D. A. Basin, X. Hofmeier, R. Sasse, and J. Toro-Pozo, "Getting chip card payments right," in *FM (1)*, ser. LNCS, vol. 14933. Springer, 2024, pp. 29–51.

[27] G. Girol, L. Hirschi, R. Sasse, D. Jackson, C. Cremers, and D. A. Basin, "A spectral analysis of Noise: A comprehensive, automated, formal analysis of Diffie–Hellman protocols," in *USENIX Security Symposium*. USENIX Association, 2020, pp. 1857–1874.

[28] J. Boyland, "Checking interference with fractional permissions," in *SAS*, ser. LNCS, vol. 2694. Springer, 2003, pp. 55–72.

[29] M. J. Parkinson and G. M. Bierman, "Separation Logic and abstraction," in *POPL*. ACM, 2005, pp. 247–258.

[30] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens, "VeriFast: A powerful, sound, predictable, fast verifier for C and Java," in *NASA Formal Methods*, ser. LNCS, vol. 6617. Springer, 2011, pp. 41–55.

[31] Q. Cao, L. Beringer, S. Gruetter, J. Dodds, and A. W. Appel, "VST-Floyd: A Separation Logic tool to verify correctness of C programs," *J. Autom. Reason.*, vol. 61, no. 1-4, pp. 367–422, 2018.

[32] M. Eilers and P. Müller, "Nagini: A static verifier for Python," in *CAV (1)*, ser. LNCS, vol. 10981. Springer, 2018, pp. 596–603.

[33] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging Rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 147:1–147:30, 2019.

[34] W. Penninckx, B. Jacobs, and F. Piessens, "Sound, modular and compositional verification of the input/output behavior of programs," in *ESOP*, ser. LNCS, vol. 9032. Springer, 2015, pp. 158–182.

[35] Google, "Capslock," 2024. [Online]. Available: https://github.com/google/capslock

[36] AbsInt Angewandte Informatik GmbH, "Astrée static analyzer for C and C++," 2025. [Online]. Available: https://www.absint.com/astree

[37] J. Protzenko, B. Parno, A. Fromherz, C. Hawblitzel, M. Polubelova, K. Bhargavan, B. Beurdouche, J. Choi, A. Delignat-Lavaud, C. Fournet, N. Kulatova, T. Ramananandro, A. Rastogi, N. Swamy, C. M. Wintersteiger, and S. Z. Béguelin, "EverCrypt: A fast, verified, cross-platform cryptographic provider," in *S&P*. IEEE, 2020, pp. 983–1002.

[38] Go developers, "The Go memory model," 2022. [Online]. Available: https://go.dev/ref/mem

[39] L. Arquint, F. A. Wolf, J. Lallemand, R. Sasse, C. Sprenger, S. N. Wiesner, D. A. Basin, and P. Müller, "Sound verification of security protocols: From design to interoperable implementations (extended version)," *CoRR*, vol. abs/2212.04171, 2022.

[40] V. Vafeiadis, "Concurrent Separation Logic and operational semantics," in *MFPS*, ser. Electronic Notes in Theoretical Computer Science, vol. 276. Elsevier, 2011, pp. 335–351.

[41] Amazon Web Services, Inc., "AWS Key Management Service," 2024. [Online]. Available: https://aws.amazon.com/kms/

[42] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno, "SoK: Computer-aided cryptography," in *S&P*. IEEE, 2021, pp. 777–795.

[43] M. Avalle, A. Pironti, and R. Sisto, "Formal verification of security protocol implementations: a survey," *Formal Aspects Comput.*, vol. 26, no. 1, pp. 99–123, 2014.

[44] B. Blanchet, "Security protocol verification: Symbolic and computational models," in *POST*, ser. LNCS, vol. 7215. Springer, 2012, pp. 3–29.

[45] D. Pozza, R. Sisto, and L. Durante, "Spi2Java: Automatic cryptographic protocol Java code generation from Spi calculus," in *AINA*. IEEE, 2004, pp. 400–405.

[46] D. Cadé and B. Blanchet, "From computationally-proved protocol specifications to implementations," in *ARES*. IEEE, 2012, pp. 65–74.

[47] K. Bhargavan, A. Bichhawat, Q. H. Do, P. Hosseyni, R. Küsters, G. Schmitz, and T. Würtele, "DY*: A modular symbolic verification framework for executable cryptographic protocol code," in *EuroS&P*. IEEE, 2021, pp. 523–542.

[48] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "Owl: Compositional verification of security protocols via an information-flow type system," in *S&P*. IEEE, 2023, pp. 1130–1147.

[49] P. Singh, J. Gancher, and B. Parno, "OwlC: Compiling security protocols to verified, secure, high-performance libraries," in *USENIX Security Symposium*. USENIX Association, 2025, pp. 5071–5090.

[50] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 1, pp. 5:1–5:61, 2008.

[51] N. O'Shea, "Using Elyjah to analyse Java implementations of cryptographic protocols," in *FCS-ARSPA-WITS-2008*, 2008.

[52] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational verification of C protocol implementations by symbolic execution," in *CCS*. ACM, 2012, pp. 712–723.

[53] P. Agten, B. Jacobs, and F. Piessens, "Sound modular verification of C code executing in an unverified context," in *POPL*. ACM, 2015, pp. 581–594.

[54] S. Ho, J. Protzenko, A. Bichhawat, and K. Bhargavan, "Noise*: A library of verified high-performance secure channel protocol implementations," in *S&P*. IEEE, 2022, pp. 107–124.

[55] J. Bader, J. Aldrich, and É. Tanter, "Gradual program verification," in *VMCAI*, ser. Lecture Notes in Computer Science, vol. 10747. Springer, 2018, pp. 25–46.

[56] J. Wise, J. Bader, C. Wong, J. Aldrich, É. Tanter, and J. Sunshine, "Gradual verification of recursive heap data structures," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 228:1–228:28, 2020.

[57] C. Andrici, Ş. Ciobâcă, C. Hritcu, G. Martínez, E. Rivas, É. Tanter, and T. Winterhalter, "Securing verified IO programs against unverified code in F*," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, pp. 2226–2259, 2024.

[58] C. Andrici, D. Ahman, C. Hritcu, R. Icleanu, G. Martínez, E. Rivas, and T. Winterhalter, "SecRef*: Securely sharing mutable references between verified and unverified code in F*," *CoRR*, vol. abs/2503.00404, 2025.