Abstracting Architectures: Two Techniques in Formal Hardware Security Verification

Alejandro Sanchez Ocegueda

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-169
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-169.html

August 15, 2025

Acknowledgement

I would like to express my deepest gratitude to Adwait Godbole and Sanjit Seshia, who have been guides and mentors throughout the better part of three years. Thank you for your guidance in these exciting and challenging projects, and for your unwavering encouragement.

Abstracting Architectures: Two Techniques in Formal Hardware Security Verification

by

Alejandro Sanchez Ocegueda

A technical report submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Summer 2025

Abstracting Architectures: Two Techniques in Formal Hardware Security Verification

Abstract

Abstracting Architectures: Two Techniques in Formal Hardware Security Verification

by

Alejandro Sanchez Ocegueda

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Sanjit A. Seshia, Chair

Recent years have seen a dramatic increase in cyberattacks that target vulnerable hardware. One of the most effective ways to mitigate such vulnerabilities is to formally verify that hardware designs adhere to security standards. However, in a world with an insatiable need for performance, tight development cycles, and ever-evolving technical demands, verification efforts often fail to keep pace with hardware development. In this work, we present two approaches to making formal verification more efficient.

Our first contribution is a formal model for pointer-encryption schemes, U^2. This model serves as a formal foundation for any system that enforces memory safety by using cryptography to protect pointers and data. We implement our model in UCLID5, and prove that it satisfies all desired security properties.

Our second contribution is BtorSec, a security-aware extension of the popular Btor2 format. By adding new cryptographic instructions, BtorSec enables formal reasoning about cryptography in real-world RTL designs. Additionally, we implement a compiler and a proof-of-concept solver for security queries on BtorSec programs.

Together, these contributions advance the state of the art in hardware security verification, enabling earlier detection of vulnerabilities and lowering the barrier to adopting formal methods in modern hardware design.

To everyone who has been a part of this journey.
Maybe the real thesis was the friends we made along the way.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my M.S. advisor, Professor Sanjit Seshia, initially for allowing me to join his Formal Methods class back in Spring 2023, and for later introducing me to the Learn&Verify group. Thank you for the continued mentorship and guidance throughout these years–I have learned (and verified) a lot in my time in the group. Thank you for taking the time to read and give feedback on my drafts.

I would also like to thank my second reader, Professor Christopher Fletcher, for the continued support and advice throughout the Spring 2025 semester. Your comments, suggestions, and encouragements in the CS 252A discussions have helped shaped the work in this report tremendously.

The next round of thank-yous goes to the Learn&Verify group members, for sharing your wisdom and for the lovely group lunches every Friday. A special thank you goes to the UCLID folks, who have been following the development of this project for a very long time. Your feedback and suggestions have greatly influenced the development of SSV, and I cannot thank you enough. And, of course, a big shoutout to my fellow 5th Year M.S. students, Anirudh and Kai.

I also want to take the time to thank the friends I made at UC Berkeley, who have made my time as a student here both a pleasure and a privilege. Thank you to all my friends at I-House, especially Bryant, Natalie, Melanie, Tatsu, Serena, and the Dwellers. Thank you to the Smash at Berkeley people for giving me the opportunity to grow and develop a passion outside of academics. And thank you to the all the HKNers for always having my back in all of my EECS classes.

I want to extend my deepest gratitude to Adwait Godbole, who has been my mentor and closest collaborator for the past two years. Thank you for the guidance. Thank you for the patience as I stumbled my way across the vast sea that is research. Thank you for the long debugging sessions. This project quite literally would not have been possible without your help.

Por último, quiero agradecer a mi familia, quienes han estado conmigo desde hace 24 años. Gracias Baba, gracias Babi, y gracias Yoyi por siempre estar a mi lado, estén en donde estén. Juntos hemos cruzado fronteras, mares, y ahora, la meta final.
¡Lo conseguimos, lo conseguimos!

Chapter 1

Introduction

With the advent of technologies like generative artificial intelligence, cryptocurrencies, and cloud-based services, the demand for computing power around the world has never been higher. Models must be trained, transactions must be committed, requests must be fulfilled. In this age of unprecedented competition and innovation, there is not a second to waste.

To satiate the industry's ever-growing need for speed, engineers work across the stack and around the clock, looking to squeeze as much performance out of their systems as possible. One particularly effective method of improving performance that has gained much traction recently is *hardware acceleration*, a technique that involves directly implementing important functionality in custom hardware circuits. These tailor-made circuits often provide massive increases in performance, easily outperforming software-based optimizations by large margins.

One particular class of hardware accelerators has received much attention in recent years: cryptographic accelerators. Cryptography is often framed as a necessary evil—components of the system that hinder performance, but prevent much more dire consequences, like the leakage of sensitive data. Indeed, cryptography presents itself as the perfect candidate for hardware acceleration: it is a necessary operation that could lead potentially be a bottleneck for performance. By implementing custom circuits to provide cryptographic functionality, we can offset this performance penalty, achieving the dream of having both security and performance.

Unfortunately, the solution to secure and efficient hardware designs is not that straightforward. Well-designed and properly implemented cryptographic accelerators are not enough. Much like in the software realm, if cryptographic primitives are not used appropriately, the security of a hardware design can still be compromised, even if the primitives work exactly as intended. Thus, to fully achieve the vision of secure and efficient hardware, we must also ensure the correct integration of cryptographic primitives into the larger design. It is not enough to have all the right pieces of the puzzle—we must also ensure that it is assembled correctly.

Challenges

Guaranteeing the security of hardware designs in the modern age is no small feat, and doing so poses a number of challenges, which we outline below:

(I) How can we develop formal models to reason about the high-level security properties of our designs? What kinds of abstractions allow us to reason about the security of our hardware?

(II) Once we implement a design in RTL, how can we verify its security properties?

(III) Lastly, how can we develop tools that are easy to use and provide meaningful feedback on the security of hardware designs, in an era where hardware implementations can change from one day to another?

This Report

The work presented in this text represents our approach to addressing the challenges laid out above. Our vision is that by combining techniques in the areas of formal methods, security, and computer architecture, we can make some progress towards developing effective solutions for verifying security of hardware designs.

1.1 Contributions and Organization

The work presented in this report is the culmination of a collaborative effort. The work for Chapter 2 started as a class project for the Spring 2023 iteration of EECS 219C at UC Berkeley. The initial work was done with my classmates Tommy Joseph and Nigel Chen, with guidance from Adwait Godbole and Sanjit A. Seshia. The work that followed, and that constitutes the majority of said chapter, was done in collaboration with Adwait Godbole and Sanjit A. Seshia. The work on Chapter 3 was done in collaboration with Adwait Godbole, with input and guidance from Sanjit A. Seshia, Tianrui Wei, and Christopher W. Fletcher. Below we present a brief overview of this report, along with the primary contributions of each piece of work:

Chapter 2: U^2: A Formal Model of Pointer Encryption Schemes. This chapter provides our approach to modeling and verifying a complex hardware cryptosystem with the UCLID5 toolkit [1], [2]. This allows us to reason about the correctness and memory safety guarantees of the C^3 cryptosystem at a high level. Additionally, the modeling techniques utilized therein can be used to tackle similar high-level system verification problems.
Chapter 2 is organized as follows:

- Section 2.4 presents our formal model of the U^2 cryptosystem. This section also specifies the main security properties that our model provides.

- In Section 2.5, we present our UCLID5 implementation of U^2. We highlight key abstractions and design decisions that went into the implementation.

Chapter 3: BtorSec: A Model-Checking Format for Hardware Security. This chapter presents BtorSec, our novel extension of the popular Btor2 [3] format. Our extension, which consists of abstract cryptographic instructions, allows us to reason about the correct usage of cryptographic primitives in RTL design. This chapter also provides proof-of-concept tools for (almost) automatically compiling and verifying BtorSec designs, thus minimizing overheads and effort required on the engineering side.
Chapter 3 is organized as follows:

- Section 3.4 describes BtorSec, our security-centric extension of the popular Btor2 model-checking format. Specifically, Section 3.4.1 presents the new abstract cryptographic instructions of the format, while Section 3.4.2 introduces the SecSpec format, which is used to annotate a Verilog design with security-relevant metadata.

- In Section 3.5, we explain the BtorSec compilation pipeline, which allows users to create BtorSec files from existing Verilog designs and an associated SecSpec.

- Section 3.6 dives into the implementation of SSV, our prototype bounded model checker for BtorSec programs.

Chapter 4: Conclusion. This final chapter summarizes this report and provides suggestions for future research stemming from our work.

The chapters of this report are self-contained. Each chapter includes the necessary background required to understand it, as well as the related work most relevant to its focus. We assume only a basic familiarity with first-order logic (FOL), including the negation operator ($\neg$), the logical connectives—conjunction ($\wedge$), disjunction ($\vee$), conditional ($\Rightarrow$), and biconditional ($\Leftrightarrow$)—and a basic understanding of the existential ($\exists$) and universal ($\forall$) quantifiers.

Chapter 2

U^2: A Formal Model of Pointer Encryption Schemes

Are we building the right thing?

Cryptographic systems claim to provide a number of security guarantees, but how can we be sure this is indeed the case? Formal models allow us to rigorously reason about the security properties of such systems. We provide a formal model U^2 that provides cryptographic memory safety guarantees. We also present an implementation of U^2 in the UCLID5 language. This chapter addresses Challenges (I).

2.1 Introduction

Certain low-level languages, such as C and C++, allow programmers to directly manipulate a program's memory. This flexibility lets programmers take full advantage of the underlying hardware, often enabling high performance. However, the burden of correct memory management falls entirely on the programmer, creating opportunities for a wide range of memory-related bugs.

This class of memory corruption bugs—also known as memory safety vulnerabilities—often leads to undefined and potentially dangerous behavior. Furthermore, malicious attackers can take advantage of these vulnerabilities to perform a number of attacks. For instance, an attacker may index beyond the bounds of an array to access other parts of memory, or they may use a pointer that was previously freed to manipulate the data inside the new allocation. These attacks can have serious consequences, ranging from leaking sensitive data to allowing an attacker to take control of the machine running the vulnerable code.

Several solutions have been proposed to deal with the threat of memory corruption. Many propose rewriting code written in vulnerable languages. Most of the effort on this front involves translating code into Rust, as this ideally would provide both performance and safety [4]. Other solutions involve statically analyzing the source code for vulnerabilities

[5]–[7] or monitoring the code during runtime to detect any vulnerabilities when they happen [8], [9].

Recently, researchers have introduced mitigations that operate at the hardware level. These solutions introduce specialized hardware that performs the necessary checks required to enforce memory safety, ensuring that software that runs on these machines is secure. The philosophy behind this approach is that by utilizing dedicated hardware, users will enjoy security guarantees and minimal performance overheads. A classic example of hardware-based memory safety enforcement is the use of hardware-based pointer tagging mechanisms, like ARM's Memory Tagging Extension (MTE) [10] and SPARC's Application Data Integrity (ADI) [11]. These solutions use tags to enforce that pointers are not improperly modified. Another prominent example is capability architectures, such as CHERI [12]. Capability architectures replace pointers with *capabilities*, which are objects that store additional information like permissions, bounds, and more. This additional information is used in hardware-based checks that enforce correct memory management.

Intel recently proposed a hardware-based architecture for enforcing memory safety: the Cryptographic Capability Computing microarchitecture, stylized as C^3 [13]. C^3 is a capability architecture that defends against memory-based attacks through cryptography. In a nutshell, C^3 protects the heap address space by encrypting pointers to memory, and then uses these encrypted pointers as part of the key used to access the data. The authors claim that this scheme is sufficient to provide confidentiality and integrity of all data in the heap, effectively mitigating vulnerabilities that target the heap altogether.

It is natural to have some skepticism at such bold claims. The authors show C^3's effective protection on a comprehensive suite of vulnerable code, but how can we be sure that it was not just good luck? Furthermore, if the C^3 scheme is indeed secure, we would like a general model or framework that future work can build upon.

To move beyond empirical evaluation and provide principled security guarantees, we develop a formal model for C3-like pointer-encryption schemes. We name our model Unobservably Unbreakable, or U^2, for short. Formal models allow us to reason about systems and their properties soundly. Additionally, they enable—and indeed require—us to precisely specify properties that are often ambiguous, such as "memory safety," "integrity," and "confidentiality." Given that hardware-based memory safety mitigations like C^3 involve the interaction of hardware and software, we employ the UCLID5 language [2]. The language's multi-modal modeling capabilities allow us to specify and verify complex systems like C^3 seamlessly.

2.2 Contributions

Concretely, the contributions of this chapter can be summarized as follows:

1. We formulate U^2, an abstract model for C^3-like pointer encryption systems.

2. We give a characterization of memory safety properties for pointer encryption schemes.

3. We implement and verify a model of U^2 in UCLID5, showing that our model satisfies our memory safety properties.

2.3 Background

This section provides the necessary background information for this chapter. We begin with a quick treatment of memory safety. Then, we proceed to explain the design of C^3. Finally, we give a short primer on UCLID5.

2.3.1 Memory Safety

Memory safety is the property of a program or system that ensures all memory accesses are valid—meaning the program only reads from and writes to memory it has been allocated, and only while that memory is still in scope. Violations of memory safety, such as buffer overflows, use-after-free errors, or accessing uninitialized memory, can lead to crashes, data corruption, or exploitable security vulnerabilities. Enforcing memory safety can be done through safe programming languages, runtime checks, or hardware-based protections. We refer interested readers to [14] for a more detailed survey of this topic.

2.3.2 The C^3 Microarchitecture

As stated in the introduction to this chapter, Cryptographic Capability Computing (C^3) [13] is a capability architecture developed by Intel to provide memory safety in the heap. C^3's novelty stems from the fact that it relies entirely on cryptography to provide its memory safety properties. This allows the architecture to remain efficient, and it has the added benefit of not needing to keep track of any additional state, unlike other hardware-based mitigations. The most salient aspect of C^3 is the way in which the architecture encrypts pointers, and then utilizes these encrypted pointers as part of the keystream used to access the associated data. This has the effect of 'entangling' the pointer with the data that it points to in the heap.

In what follows, we give a brief introduction to how C^3 works. The diagrams presented in this section have been adapted from [13] and [15].

Pointer Encryption

One of the innovations of C^3 is its use of cryptographic addresses (CAs). CAs are a special format for pointers that only encrypt a specific portion of a plaintext linear address (LA). They are created at the time of memory allocation (i.e. after a call to `malloc`). The format as specified in the original C^3 paper can be seen in Figure 2.1.

The fields of the format are the following:

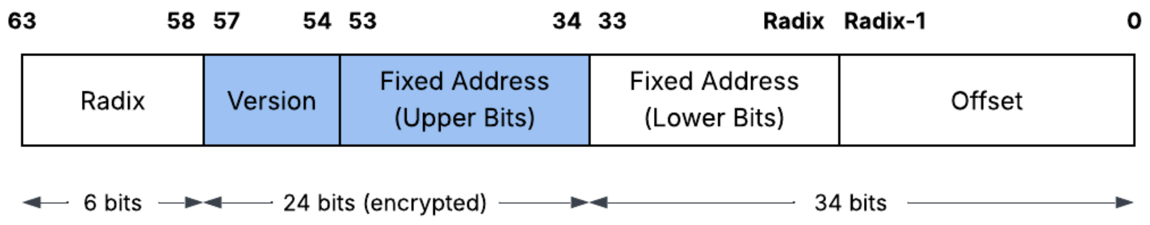

Figure 2.1: C^3's Cryptographic Address Format

- **Radix:** This 6-bit field specifies the size of the allocation[1], in power of two.
- **Version & upper fixed address:** This 24-bit range consists of the upper 20 bits of the original pointer and the 4 bits of the Version field put together. The Version field is used to avoid temporal safety issues stemming from the underlying LA being the same for two CAs that were allocated at different times.
- **Lower fixed address & offset:** This 34-bit range is left unchanged from the original LA. This is done so that software can perform pointer arithmetic without requiring cryptographic intervention.

CAs can be created from LAs by encrypting the corresponding fields. C^3 achieves this by using a tweakable block cipher named “K-cipher” [16]. The flow for generating a CA from an LA is shown in Figure 2.2.

Data Encryption

The next important component of C^3 is data encryption and decryption. Encryption happens at the time data is stored into a memory slot allocated by C^3. Decryption happens whenever data is loaded from a memory slot. One key aspect of how these operations are performed is that *the pointer to the data itself is part of the key.* The flow of how data encryption and decryption happens during load and store operations can be seen in Figure 2.3.
,

2.3.3 UCLID5 Primer

UCLID5 is a software toolkit for the formal modeling, specification, verification, and synthesis of computational systems. In this section, we briefly familiarize the reader with UCLID5. For a more thorough discussion of UCLID5, we refer the reader to the tutorial[2] or to the original UCLID5 papers [1], [2].

[1]In the C^3 scheme, the size of all allocations is a power of two, by design.
[2]https://github.com/uclid-org/uclid/blob/master/tutorial/tutorial.pdf

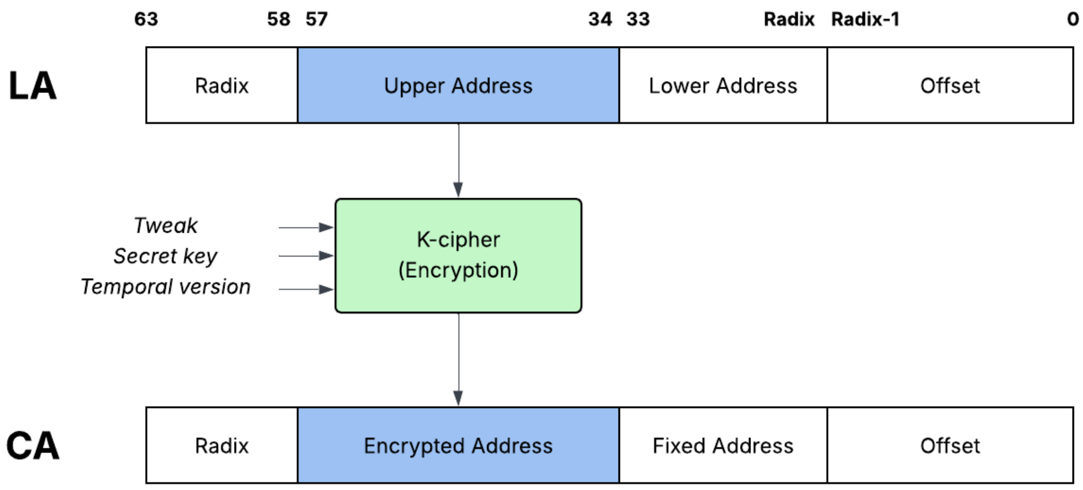

Figure 2.2: Flow of C^3 Pointer Encryption

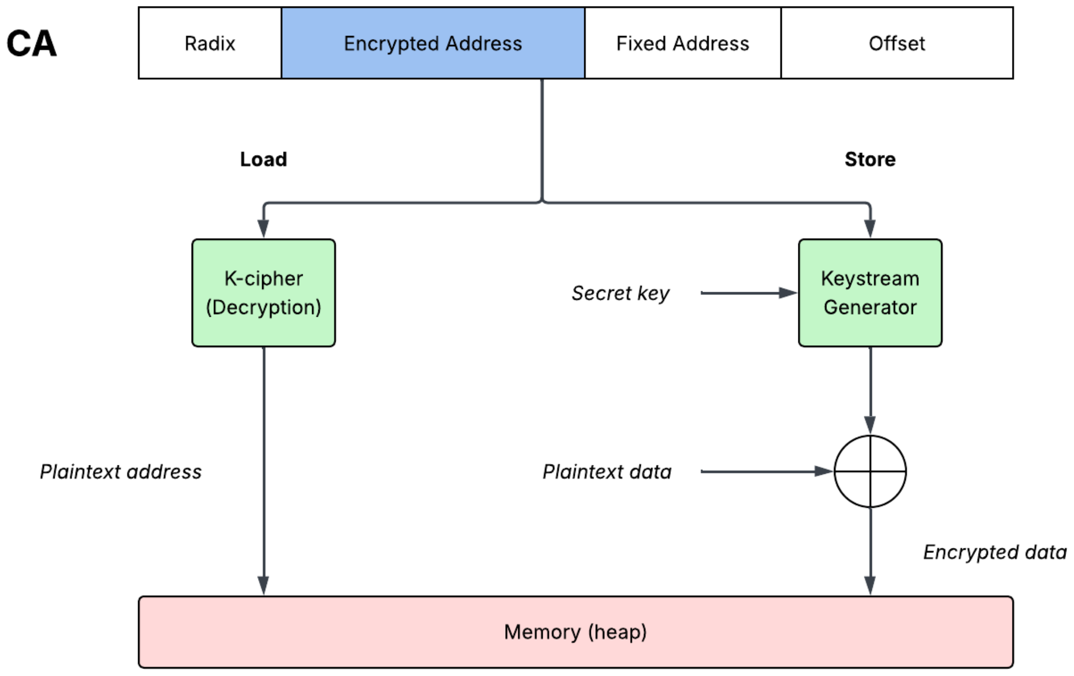

Figure 2.3: Flow of C^3 Data Encryption and Decryption

We now give two examples of UCLID5 in action. First, we show how we can model state transition systems by modeling the Fibonacci sequence. Second, we show an example of how we can construct and use algebraic data types in UCLID5 .

State Transition Systems

The UCLID5 language allows us to seamlessly model, specify, and verify state transition systems like the one shown in Figure 2.4. This example module of the Fibonacci sequence, while simple, is enough to show the power of UCLID5.

```
 1 module main {
 2 
 3     // Part 1: System description
 4     var x, y : integer;
 5 
 6     init {
 7         x = 0;
 8         y = 1;
 9     }
10 
11     next {
12         x' = y;
13         y' = x + y;
14     }
15 
16     // Part 2: System specification
17     property y_le_x : y <= x;
18     property ind_strengthen : x >= 0 && y >= 0;
19 
20 
21     // Part 3: Proof script
22     control {
23         v = induction;
24         check;
25         print_results;
26     }
27 }
```

Figure 2.4: A simple UCLID5 module of the Fibonacci sequence

Part 1: System Description. We can describe the system's behavior, as the module does in lines 4-14. Intuitively, we are describing *what the system does.* First, we declare the integer variables `x` and `y`. Then, in the `init` block on lines 6-9, we declare the initial values of `x` and `y` (`0` and `1`, respectively). Below that, in lines 11-14, the `next` block specifies how the variables will change after one step of the transition relation is executed. This is denoted with the primed variable notation: `x' = y` indicates that the next value of `x` in will be the current value of `y`. Similarly, the notation `y' = x + y` indicates that the next value of `y` will be the sum of the current values of `x` and `y`. This aligns with the behavior of the Fibonacci sequence.

Part 2: System specification. UCLID5 also allows us to create specifications for our models. Intuitively, the specification allows us to clearly define *what the system is supposed* *to do.* In this example, we have two properties that describe the correctness of our system: `y_le_x` on line 17 and `ind_strengthen` on line 18. `y_le_x` is the property we wish to prove, whereas `ind_strengthen` is a property that is used to strengthen the induction and avoid spurious counterexamples.

Part 3: Proof script. Finally, the we have the `control` block on lines 22-26. This block consists of a sequence of commands that are used to verify the properties we specified earlier. In this model, we use induction to verify our properties. The `check` command checks whether the proof obligations are satisfied and the `print_results` command prints out the results of these checks.

Algebraic Data Types

We now proceed to show an example of how we can construct and utilize algebraic data types (ADTs) in our systems. Consider the example shown in Figure 2.5.

```
 1 module main {
 2
 3     // Part 1: System description
 4     datatype list = cons(head: integer, tail: list) | nil() ;
 5
 6     var l : list;
 7
 8     init {
 9         l = nil();
10     }
11
12     next {
13         l' = cons(1, l);
14     }
15
16
17     // Part 2: System specification
18     invariant test : l.head == 1;
19
20
21     // Part 3: Proof script
22     control {
23         induction;
24         check;
25         print_results;
26     }
27 }
```

Figure 2.5: A UCLID5 module of a linked list

Part 1: System description. This module consists of a single variable `l` of type `list` (line 6). The `list` type is a user-specified algebraic data type. Users can create their own ADTs by using the `datatype` keyword, followed by the type name, as is done on line 4. Then, users can specify a grammar of *constructors*. In our example, we have two constructor: the 0-ary `nil`, which represents a null value and the binary `cons` constructor. `cons` recursively builds a `list` by creating a `head` node from an `integer` value and appending it to the `tail list`.

Initially, we create an empty list by declaring `l = nil()` in the `init` block (lines 8-10). Then, at every state transition, we grow the list by appending a `1` to the head

Part 2: System specification. The only property we wish to verify is that in this system, the `head` of our `list l` always has a value of `1`.

Part 3: Proof script. Much like the previous example, we verify the correctness of this system by `induction`. We then use the `check` command to check that our module meets the proof obligations. Finally, we print out the results with the `print_results` command.

2.4 Formal Model

In this section, we present our formal model of the abstract U^2 system. We define our model as the triple $\mathrm{U}^2 = (\Sigma, T, init)$. Σ is a set of states, or a valuation of the state variables of the model. $T \subseteq \Sigma \times \Sigma$ is the transition relation, subject to the operations described later in this section. Finally, $init \subseteq \Sigma$ is the set of allowed initial states.

We begin by introducing the U^2 model, followed by a description of the U^2 adversary. To end the section, we outline the key security properties that we require the U^2 model to satisfy.

2.4.1 State Variables

We now present the state variables of U^2. These give a characterization of our system's states, as any $\sigma \in \Sigma$ is merely a valuation of these variables. The state variables of U^2 can be found in Table 2.1. A detailed description of all types can be found in Appendix A.

Initial States

We model our initial state as a fresh start in which no allocations have been made. In other words, any initial state $\sigma_i \in init$ must satisfy the following condition:

$$\forall a.\ \texttt{alloc\_map}[a] = \texttt{false}.$$

2.4.2 Operations

We now describe the operations that our U^2 model allows. We model these as an API that all processes running in the system must use to manipulate memory. These give a characterization of the transition relation T.

State Var.	Type	Description
`msv_curr`	$\mathbb{B}$	Variable denoting whether a memory safety vulnerability has occurred in the current step.
`alloc_map`	$W \to \mathbb{B}$	Map indicating whether a given memory address has been allocated.
`mem`	$W \to W$	Physical memory.
`ghost_mem`	$W \to \mathcal{G}$	Memory augmented with ghost metadata.
`lmem`	$I \to W$	Local memory (per-process registers).
`ghost_lmem`	$I \to \mathcal{G}$	Local memory augmented with ghost metadata.
`observable`	$\mathcal{O}$	Microarchitectural observable variable.
`opcode`	O	Opcode for non-memory instructions.
`l_i`	I	Local memory indices.
`action`	A	The current operation to be executed.

Table 2.1: U^2 State Variables

Operation	Description
`malloc`(l_{dest}, k_p, k_d)	Creates a new allocation in memory, ensuring pointer is encrypted under k_p and data is encrypted under k_d. Stores resulting pointer in l_d.
`load`$(l_{dest}, l_{addr}, k_p, k_d)$	Uses pointer at register l_{addr} and k_p to load value from memory, decrypted under key k_d. Stores the result in l_{dest}.
`store`$(l_{addr}, l_{val}, k_p, k_d)$	Encrypts value at register l_{val} under k_d. Then, stores the encrypted value at the address computed using l_{addr} and k_p.
`free`(l_{addr}, k_p)	Computes physical address a using l_{addr} and k_p. Then, frees the memory by setting `alloc_map[a] == false`.
`hanop`$(op, l_{dest}, l_{src_1}, l_{src_2})$	Performs pointer arithmetic using the values at register indices l_{src_1} and l_{src_2}. Stores the result in l_{dest}.
`arithop`$(op, l_{dest}, l_{src_1}, l_{src_2})$	Performs an abstract arithmetic operation and returns a non-pointer value.

Table 2.2: The U^2 API

Memory Safety Vulnerabilities

In addition to the behaviors described in Table 2.2, the `load` and `store` operations can potentially trigger a memory safety vulnerability by using an invalid pointer. This has the effect of setting the `msv_curr` to `true` during that step.

2.4.3 The U^2 Adversary

The U^2 adversary $\mathcal{A}$ is allowed to start any number of processes and perform arbitrary operations with them. Additionally, $\mathcal{A}$ comes equipped with an observation function *Obs*, which enables them to observe certain parts of the state. To model an adversary similar to the one described in the C^3 paper, we let $\mathcal{A}$'s *Obs* function be such that they are allowed to observe any values in their process, in addition to the `observable` system variable. Intuitively, this means that the adversary has access to any of the values seen by their process, in addition to a microarchitectural side channel that leaks information.

2.4.4 Security Properties

We aim to prove two key security properties: *confidentiality* and *integrity*. Informally, *confidentiality* means that the adversary is unable to read the contents of a valid allocation in plaintext through a memory safety vulnerability. Meanwhile, we define *integrity* to mean that any values stored through a memory safety vulnerability cannot be decrypted back as plaintext, ensuring that $\mathcal{A}$ is not able to inject arbitrary values or code into other users' memories.

Definition 2.1. Confidentiality:

$$\begin{gathered}\texttt{msv\_curr} = \texttt{true} \Rightarrow \\ (\texttt{observable.valid} = \texttt{false} \lor \texttt{observable.g\_data.enc\_state} \neq \mathit{Plain})\end{gathered}$$

The formula above says that whenever a memory safety vulnerability is triggered, either the data observed through the `observable` channel is invalid, or the data must not be in plaintext.

Definition 2.2. Integrity:

$$\begin{aligned}\forall a.\ & \texttt{alloc\_map[}a\texttt{]} = \texttt{false} & \lor \\ & \big(\mathit{isDenc}(\texttt{ghost\_mem[}a\texttt{].enc\_state}) & \land \\ & (\texttt{ghost\_mem[}a\texttt{].enc\_state.DEnc\_nonce} = \texttt{ghost\_mem[}a\texttt{].nonce\_true})\big) & \lor \\ & \mathit{isGarbled}(\texttt{ghost\_mem[}a\texttt{].enc\_state}) &\end{aligned}$$

This formula specifies that at all time steps, any valid allocation must either be decrypted with the correct nonce, or it must be garbled.

2.5 UCLID5 Model

In this section, we provide an overview of implementation of U^2 in the UCLID5 language. We begin with a brief treatment of the different modules in Section 2.5.1. Then, we move on to the verification results in Section 2.5.2. For the sake of brevity, we highlight only the most important aspect of each module. We direct readers interested in learning more to either Appendix B or our GitHub repository [17].

2.5.1 Modules

The Common Module

We begin our description of our UCLID5 model with a description of the common module. This module contains basic type declarations that the other modules rely on.

```
module common {

    type word_t = bv2;
    type size_t = word_t;

    type opcode_t;

    type lmapind_t = bv3;

    // Attacker, and Victim keys
```

We note that we model the word type as a 2-bit bit-vector, and the `lmap_ind` type as a 3-bit bit-vector. This is done mainly for the sake of simplicity, as larger bit-vectors would likely have no effect on the verification, and would likely only cause the proof to take much longer to finish.

```
    // Attacker, and Victim keys
    type key_t = enum { AtKey, ViKey };
```

The `key_t` type is used to model the keys for each process. Implicitly, this also specifies the number of processes in the system. In our case, we only model the adversary and the victim processes. We argue that there is no need for modeling more processes, as there are no new behaviors introduced with the presence of extra processes. This, again, would only lead to verification being less tractable, which is something we want to avoid at all costs, given that we are modeling a system as complex as U^2.

```
    datatype enc_state_t = Pln()
        | DEnc(DEnc_val: enc_state_t, DEnc_key: key_t, DEnc_nonce: nonce_t
    ↪ , intcheck: boolean)
        | DDec(DDec_val: enc_state_t, DDec_key: key_t, DDec_nonce: nonce_t
    ↪ )
        | PEnc(PEnc_val: enc_state_t, PEnc_key: key_t)
        | PDec(PDec_val: enc_state_t, PDec_key: key_t);
```

The next type declaration we highlight is the algebraic data type used to model cryptography. This consists of a base `Pln()` constructor, and appropriate constructors for the other states of encryption, for both data and pointers.

Note that we do not model the `free` operation, as this was difficult to model and caused issues with proof convergence[3].

```
    type ghost_data_t = record {
        // Raw value
        value: word_t,

        // Value state (i.e. is this a handle or a raw value?)
        vtype: value_t,

        // Handle elements
        // Base address (inclusive)
        h_base: word_t,
        // This is the current handle offset
        h_offset: word_t,
        // Allocation is [h_base, h_base+h_length)
        h_length: word_t,

        // Only modified by the encryption function
        // Encryption state
        enc_state: enc_state_t,
        // True nonce for that allocation (used for authentication)
        ca_nonce_true: nonce_t
    };
```

The final type declaration we highlight is the `ghost_data_t` type. This type contains not only the real value stored in physical memory, but also several metadata fields that are necessary for verification:

- The `vtype` field indicates whether this allocation contains a pointer or raw data.
- The `h_base`, `h_length`, and `h_offset` comprise the allocation's handle. This handle is used to check memory bounds and detect memory safety violations during verification.
- `enc_state` tells us which state of encryption this allocation is in. Symbolically, we should know whether the value is in plaintext, encrypted, or decrypted, reagardless of the real value that memory may hold.
- `ca_nonce_true` indicates the true nonce that was created when this allocation was made (by making a call to `malloc`). This corresponds to keeping track of which CA was made to create the allocation, and ensure that it is that same CA that is used by ensuing memory access in the original C^3 paper.

The U^2 Module

We now describe the `u2` module. In this module, we create all the necessary state variables, and we also specify the behavior of the `init` and `next` blocks. This gives a full description

[3]Moreover, the behavior of `free` is not defined in the original C^3 paper.

of our U^2 system, as the state variables characterize Σ, the `init` block gives us the set of initial values (*init*), and the `next` block gives us our transition relation T.

```
    // Ghost state
    var ghost_mem    : ghost_mem_t;
    var ghost_lmap   : ghost_lmap_t;
    // Allocated map
    var alloc_map: alloc_map_t;
```

The `ghost_mem` and `ghost_lmap` variables shown above correspond to the main memory and local memory variables, respectively (augmented with extra data). The `alloc_map` variable is used to keep track of which sections of memory have been allocated through calls to `malloc`.

```
    // Current operation was a memory safety vulnerability
    var msv_curr : boolean;
    // Architectural observable
    var observable : observable_t;
```

These `msv_curr` variable describes whether a process has triggered a memory safety vulnerability by performing an illegal memory access. The `observable` variable is used to model the adversary's ability to observe certain parts of memory through other side channels.

```
    init {
        // Initially all cells are unallocated
        assume (forall (a: word_t) :: (!alloc_map[a] && !shadow_mem[a].
   ↪ alloc));

        // Initially all values (in lmap and memory) hold raw data (not
   ↪ pointers)
```

These variables describe a process's local map. This allows users to keep a private collection of values at any given time.

```
        assume (forall (a: lmapind_t) :: ghost_lmap[a].vtype == RAW);
        assume (forall (a: word_t) :: ghost_mem[a].vtype == RAW);

        observable.valid = false;
        msv_curr = false;
    }

    axiom slot_to_nonce_disjointness :
        (forall (i1: word_t, i2: word_t, i3: word_t, i4: word_t) :: (
            non_overlapping_ranges(i1, i2, i3, i4) ==> (slot_to_nonce(i1,
   ↪ i2) != slot_to_nonce(i3, i4))
        ))
    ;
```

The init block specifies the initial state of our system. We have three assumptions in the code above:

(1) Initially, all cells are unallocated.

(2) Initially, all values in the local map are not pointers.

(3) Initially, all values in main memory are not pointers.

Assumption (1) is used to model a system starting fresh and with all memory available. This aligns with our definitions in Section 2.4. Assumptions (2) and (3) are mostly a matter of simplifying the design. We believe that constraining the program such that all values in main and local memory are initially `RAW` (i.e. not used as pointers to access memory) is a fairly reasonable assumption to make. Moreover, this drastically reduces the initial state space, and speeds up the verification effort.

Moreover, they simplify verification drastically by constraining the possible initial states of our system.

This block also specifies that the `observable` variable holds invalid data, and that no memory safety vulnerability has occurred.

```
        havoc action;

        case
            (action == LOAD) : { call load(l1, l2, ViKey, ViKey); }
            (action == STORE) : { call store(l1, l2, ViKey, ViKey); }
            (action == MALLOC) : { call malloc(l1, ViKey, ViKey); }
            (action == HANOP) : { call hanop(opcode, l1, l2, l3); }
            (action == ARITHOP) : { call arithop(opcode, l1, l2, l3); }
        esac
    }

}
```

Finally, the `next` block specifies how our C^3 model must transition from one state to the next. In our UCLID5 model, this means that the victim can choose to arbitrarily perform a standard `load`, `store`, `malloc` operation, by following the C^3 scheme. Alternatively, they can perform pointer arithmetic via the `hanop` operation to create new pointers from data they hold. Finally, we also allow the user to perform some computation on the values in their local map, with the restriction that the result is not used as a pointer later. This last restriction is included to simplify the modeling, but does not change the expressivity of our system.

Other Modules

We provide a brief overview of the other modules in our UCLID model.

- The `main` module contains all the formal properties, and is responsible for performing verification.
- The `operations` module defines the behavior of the `load`, `store`, `malloc`, `hanop`, and `arith` procedures used in the `next` block of the `u2` module.

- The `cryptography` module defines procedures that implement the behavior of the cryptography using our ADT encoding.
- The `contracts` module defines a number of macros that are useful for the procedures in the `operations` and `cryptography` modules.
- Finally, the `shadow_common`, `shadow_u2`, and `shadow_operations` modules all define the shadow memory, which is a structure that independently keeps track of the allocations of the system, which allocations each address belongs to, and which nonce has been used to encrypt it (if any). The shadow memory serves mostly as an aide in verification.

2.5.2 Verification Results

We proved a total of 24 properties to the correctness of our UCLID5 model. Notably, this includes our confidentiality and integrity properties from Section 2.4.4. Our model was able to prove the properties inductively for the two-process case in a total of 11.64 seconds. The full model, along with instructions of how to run the proofs, can be found in our GitHub repository [17].

2.6 Related Work

Security Analyses of C^3

C^3 has been the subject of much scrutiny in the security community. Mahzoun, Kraleva, Posteuca, and Ashur show that the K-cipher–the main component used to encrypt pointers in the real C^3 system–is susceptible to differential cryptanalysis attacks [18]. Additionally, Hassan presents four different attacks on the C^3 system in their PhD dissertation [15]:

1. They exploit the use of XOR as an encryption primitive to leak confidential values for allocations with known initial values.
2. They leverage the lack of bounds-checking during pointer arithmetic operation to forge CAs.
3. They take advantage of the low entropy in the Version field of the CA format to violate temporal memory safety.
4. They show how the system-level design decision to only protect the heap can be abused to break C^3's memory safety guarantees.

We note that these attacks work at a much finer level of detail than what our formal model considers in scope. This only reinforces the vision of our report that security verification must be present at all points of the design process: from the ideation and system specification all the way to the implementation.

UCLID5 for Hardware Security

The work presented in this project builds off prior work in verifying hardware security properties using UCLID5. Prior endeavors in UCLID5 modeling include but are not limited to:

- Creating verifiable models of secure hardware enclaves [19], [20], building off the formal groundwork laid out by Subramanyan et al. in their seminal paper on the Secure Remote Execution of Enclaves [21].
- Creating models of programs and simulating their execution on a speculative processor while exposing certain state to the adversary's observations. [22], [23].
- Formally verifying an open-source hardware implementation of physical memory protection (PMP) in RISC-V [24].

Our work is yet another testament to the versatility of the UCLID5 toolkit to model and verify complex systems where hardware and software interact.

Dedicated Cryptographic Verifiers

We would be remiss to overlook the work surrounding other dedicated cryptographic verifiers in recent years. We give a brief overview of some of the most popular verifiers in the literature. We refer the reader to Barbosa et al.'s SoK paper on computer aided cryptography [25] for a more detailed overview into this line of research.

Tamarin: A rather popular cryptographic verifier is Tamarin [26]. This prover operates by using multiset rewrite rules and backwards reasoning to automatically find attacks for an unbounded number of sessions. Tamarin has been used in a number of projects to model and verify the security of protocols such as TLS 1.3 [27], 5G Authentication [28], and more.

ProVerif & CryptoVerif: ProVerif [29] and CryptoVerif [30] both developed at INRIA, are other examples of cryptographic verifiers. These tools leverage the applied pi calculus [31] to allow users to verify cryptographic protocols in the symbolic and computational models of cryptography, respectively. ProVerif has been used to verify election protocols [32], secure messaging protocols [33], and to automatically find attacks based on hash function weaknesses [34]. CryptoVerif, on the other hand, has been useful for analyzing the soundness of post-quantum security protocols [35] and a number of protocols involving dynamic key compromise [36].

The tools outlined above–being primarily designed with security verification in mind–support cryptographic primitives and constructs natively. This functionality allows users to use these tools and focus on the modeling and verification of cryptographic protocols. This is an excellent choice when reasoning about the use of cryptography in isolation. We could very well have used any one of these tools to create our formal, abstract model of C^3. Ultimately, however, we were more interested in modeling C^3 as a hardware-software system. These small details are why we opted to use UCLID5 with ADTs instead.

2.7 Conclusion

This chapter introduced U^2, an abstract system model for C^3-like systems, modeled in the UCLID5 language. We also proved that our model satisfies our security properties. This work provides a rigorous framework to reason about C^3-like systems. It is our hope that this project enables architects of C^3-like systems to think about the security of their designs, or that they are inspired to integrate modeling in UCLID5 or other similar tools into their workflows.

2.7.1 Future Work

Modeling

One way in which the work presented in this chapter could be extended is by creating a lower-level model that accurately reflects the behavior of the actual C^3 system more closely. This could include microarchitectural details, such as registers, TLBs, caches, and more. Then, we could show that this lower-level C^3 model refines our U^2 model, which would provide a high degree of confidence on the soundness of the C^3 scheme.

UCLID5

Our decision to model cryptographic primitives as ADTs was in part due to necessity. Indeed, UCLID5 does not natively support the kind of probabilistic or computational reasoning that is required to verify a cryptographic system at such a fine level of detail. The best we can do at the time of writing is to model cryptography symbolically. Consequently, one interesting line of research would be to equip UCLID5 with computational and probabilistic reasoning capabilities, which would allow users to make more accurate models and have more precise security guarantees.

Chapter 3

BtorSec: A Model-Checking Format for Hardware Security

Are we building the thing right?
After validating a high-level system design with a formal model—much like we did with U^2 in Chapter 2—the natural next step is to implement it in RTL. But implementations often introduce bugs that cannot be caught by higher-level specifications. This chapter aims to provide microarchitects and verification engineers with tools to check their implementations for security vulnerabilities with minimal overhead. This chapter addresses Challenges (II) and (III).

3.1 Introduction

The previous chapter focused on the specification and verification of security properties in high-level system designs. Using the UCLID5 language, we showed that we can reason about our design by omitting several implementation details and focusing only on the core properties that we wish to satisfy. But while having high-level assurance that our abstract models are secure is useful, the ultimate goal is to ensure that the *implementations* of the models are secure, too. Many things can go wrong when moving from a high-level specification to the real implementation. The devil, as usual, is in the details.

Addressing this implementation-to-specification gap requires new verification approaches that combine the precision of RTL analysis with the abstraction of cryptographic reasoning. To this end, this chapter introduces BtorSec, a word-level model checking format for hardware designs that employ cryptographic modules. By nature of being a language extension, BtorSec enjoys many of the properties of Btor2, such as its simplicity and ease of parsing. Additionally, the new instructions introduced in BtorSec allow us to express and reason about security-related properties of an RTL design. Notably, these sorts of security properties are not expressible in the base BTOR2 language. This design philosophy—abstracting cryptographic details while preserving RTL precision—enables a new class of security verification that was previously impractical.

This chapter details both the technical design of BTORSEC and its practical application to hardware verification. Section 3.4 provides an overview of the new instructions added to BTORSEC, explaining our extension of the BTOR2 format in more detail, and what the purpose of each new instruction is. Next, Section 3.5 gives a detailed explanation of the compilation pipeline, from the initial (System)Verilog design to the final BTORSEC file. Section 3.6 introduces SSV, our prototype model checker for BTORSEC security properties. Section 3.7 compares and contrasts our approach to hardware security verification to existing approaches in the literature. Finally, we conclude the chapter in Section 3.8 by summarizing our work and providing directions for future research.

3.2 Contributions

In summary, this chapter makes the following contributions:

1. We propose BTORSEC, an extension of the BTOR2 model-checking format that adds abstract cryptographic instructions.

2. We introduce a compilation pipeline that allows users to create BTORSEC files from existing Verilog designs with minimal engineering overhead.

3. We implement a bounded model checking tool, SSV, for checking confidentiality properties of BTORSEC programs.

3.3 Background

This section provides some relevant technical background that is useful in understanding the approach to hardware verification we present in this chapter. We begin with a quick introduction to the Verilog and BTOR2 languages. We emphasize the usefulness of the BTOR2 format as an intermediate representation of hardware designs, motivating our decision to create its BTORSEC extension, presented in Section 3.4. We then proceed to cover some basic formal verification techniques that will be useful in describing our approach to verifying BTORSEC programs in Section 3.6.

3.3.1 Hardware Design

This subsection provides a brief introduction to the Verilog, SystemVerilog, and BTOR2 languages, focusing on the aspects most relevant to hardware modeling and formal verification. We give a simple introduction of the popular Verilog and SystemVerilog hardware description languages. We also stress the BTOR2 format's role as an intermediate representation for model checking. This background will equip the reader with the necessary context to understand the modeling choices and verification workflows presented in subsequent sections.

Verilog and SystemVerilog

Verilog [37] is a hardware description language (HDL) originally developed in the 1980s for modeling and simulating digital circuits. It provides a C-like syntax for describing hardware behavior at multiple levels of abstraction, from gate-level implementations to high-level behavioral descriptions. Verilog became an IEEE standard (IEEE 1364) and remains one of the most widely used HDLs in the semiconductor industry for designing everything from simple logic circuits to complex processors and system-on-chips (SoCs).

SystemVerilog [38], introduced in the early 2000s, is a significant extension and enhancement of Verilog that addresses many of the original language's limitations. Beyond traditional hardware description capabilities, SystemVerilog adds powerful verification features including object-oriented programming constructs, constrained random testing, assertions, coverage metrics, and interfaces.

Given the similarity of these two languages, we will use the terms "Verilog" and "SystemVerilog" interchangeably. Below is a simple example of a (System)Verilog module describing a flip-flop register:

```
 1 module top (
 2     input clk,
 3     input [31:0] in,
 4     output [31:0] out
 5 );
 6
 7     reg [31:0] ff;
 8
 9     // Sequential logic
10     always @(posedge clk) begin
11         ff <= in;
12     end
13
14     assign out = ff;
15
16 endmodule
```

Figure 3.1: Flip-Flop Verilog Module

Lines `1-5` declare the module. Since this is the top-level module, it is named `top`. These lines also declare the input and output ports, in this case:

- a 1-bit input port, `clk`, corresponding to the clock,
- a 32-bit input port, `in`, corresponding to the input data, and
- a 32-bit output port, `out`, corresponding to the output data.

Line `7` declares a 32-bit register named `ff`. Then, lines `10-12` describe the behavior of the module:

- Line `10` indicates that the behavior in the block should happen at each rising edge of the `clk` signal (`always @posedge clk`).
- Line `11` indicates that the register `ff` should store the value `in`.

Finally, line `14` declares that the `out` port should have the same value as the `ff` register.

BTOR2

BTOR2 [3] is a word-level model checking format for capturing models of hardware in a bit-precise manner. It is largely similar to its predecessor, BTOR, with the main difference between the two being the addition of explicit sort declarations. By design, the BTOR2 format is minimalist and line-based. Consequently, it is also easy to parse, making it well-suited to interface with solvers and other verification tools. Let us continue our simple flip-flop module example. The BTOR2 representation of the Verilog module is shown in Figure 3.2.

```
; BTOR description generated by Yosys 0.50+56 (git sha1 176131b50, aarch64-apple-darwin23.5-clang++ 18.1.8 -fPIC -O3) for module top.
1 sort bitvec 1
2 input 1 clk ; flip_flop1/top.v:2.11-2.14
3 sort bitvec 32
4 input 3 in ; flip_flop1/top.v:3.18-3.20
5 state 3 ff
6 output 5 out ; flip_flop1/top.v:4.19-4.22
7 next 3 5 4
; end of yosys output
```

Figure 3.2: BTOR2 Code for Flip-Flop Module

Let us break down this simple BTOR2 program line by line. Each line of a BTOR2 program consists of a line identifier (which we will refer to as `lid`), followed by the name of the instruction, followed by the arguments of the instruction. These line identifiers are then used as references for other instructions. Comments are denoted with a '`;`' and any text following the '`;`' is ignored[1]. In general, BTOR2 instructions look something like this:

```
<lid> inst <params>*
```

[1]The comment at the top of the file is simply output created by the Yosys at the time of compiling the program.

The first line of the program is `1 sort bitvec 1`, which means that any subsequent instruction that uses this sort will be a 1-bit bit-victor.
The second line, `2 input 1 clk` indicates that the `lid 2` is associated with an `input` of sort `1` (i.e. a 1-bit bit-vector) named `clk`.
Similarly, the third line of the program, `3 sort bitvec 32`, declares the `lid 3` as a 32-bit bit-vector.
Line `4`, much like line `2`, indicates that `lid 4` now refers to an `input` of sort `3` (a 32-bit bit-vector) named `in`.
Line `5` declares a `state` variable (i.e. a `reg` in Verilog terms) of sort `3` named `ff`.
Line `6` declares that the value of `lid 5` is an output named `out`.
Finally, line `7` states that, at each transition, the state variable at `5` of sort `3` will take on the value of `4`.

Despite BTOR2's simple syntax, it is rather difficult to glean the purpose of a BTOR2 program by simply looking at one. Thankfully, BTOR2 files are not meant to be written or read by humans in the way Verilog files are. For the purposes of this project, the role of BTOR2 is mostly to act as an intermediate representation of Verilog modules that is easier to work with.

3.3.2 Formal Verification Techniques

We now provide a brief overview of some formal verification techniques referenced in this chapter. Of particular relevance are the symbolic simulation and fixed-point computation techniques. This is because our prototype checker for BTORSEC, SSV, employs a combination of these methods. While SAT and SMT solving serve as the underlying engines that implement these techniques, they are not the primary focus of this work; nonetheless, a basic understanding of them is beneficial for context.

Boolean Satisfiability

We provide a brief overview of the Boolean Satisfiability problem, better known as SAT. The SAT problem consists of asking the following question: given a Boolean formula[2] ϕ with variables $v_1, v_2, v_3, ...$ that are all either $True$ or $False$, is there some assignment of *True* and *False* to these variables such that the overall value of the formula ϕ is $True$? As an example, consider the following Boolean formula:

$$\phi := (v_1) \wedge (\neg v_1 \vee \neg v_2).$$

Then if we assign $v_1 = True$ and $v_2 = False$, we have that (v_1) evaluates to $True$ and $(\neg v_1 \vee \neg v_2)$ evaluates to $True$ as well. Therefore, $(v_1) \wedge (\neg v_1 \vee \neg v_2)$ also evaluates to $True$. Since there exists some assignment of $True$ and $False$ to v_1 and v_2 that makes ϕ evaluate

[2]That is, a formula that can only be evaluated to $True$ or $False$.

to $True$, we say that ϕ is *satisfiable* (or SAT, for short). We also say that $v_1 = True$ and $v_2 = False$ is a *satisfying assignment*.

The counterpart of a *satisfiable* formula is an *unsatisfiable* (UNSAT) formula, in which *no* assignment of $True$ and $False$ can make the overall formula $True$. For instance, the formula

$$\psi := (v_1) \wedge (\neg v_1)$$

is very clearly unsatisfiable. If we assign $v_1 = True$, then $\neg v_1 = False$; similarly, if we choose to make $\neg v_1 = True$, then $v_1 = False$. In any case, ψ will always be $False$.

SAT solvers essentially check *every possible assignment* of variables until they find a satisfying assignment. If at any point such an assignment is found, then the solver declares the formula in question to be SAT and returns the satisfying assignment to the user. On the other hand, if the solver exhausts all the assignments and does not manage to make the formula SAT, then it terminates and declares the formula to be UNSAT.

SAT technology has proven exceptionally useful in practice. It has been used to solve longstanding problems in mathematics, such as the Pythagorean Triples problem [39], and it is also a fundamental tool used in hardware verification in industry [40], [41]. Readers interested in learning more about SAT solving are directed to the wonderful papers *The Science of Brute Force* [42] by Heule and Kullmann and *Boolean Satisfiability: From Theoretical Hardness to Practical Success* [43] by Malik and Zhang.

Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) is the natural extension of SAT to more expressive theories. We now provide a formal definition of SMT, as well as some theories of interest for the present study. We have obtained these definitions from the *Handbook of Satisfiability* [44] and from Marker's *Model Theory: An Introduction* [45]. We refer the reader to those texts for a more thorough treatment on SMT and model theory, respectively.
Problem Statement: Let us define the SMT problem more rigorously. A signature Σ is a set of *predicate* and *function* symbols, each with an associated *arity*. For clarity, we let $\Sigma^P \subseteq \Sigma$ be the set of predicate symbols and $\Sigma^F \subseteq \Sigma$ be the set of function symbols. We call the 0-arity function symbols *constant* symbols and the 0-arity predicate symbols *propositional* symbols. A Σ-structure $\mathcal{M}$ is a pair consisting of a set M, called the underlying universe, and a mapping $(\cdot)^{\mathcal{M}}$ that assigns

1. to each constant symbol $c \in \Sigma^F$ an element $c^{\mathcal{M}} \in M$,
2. to each function symbol $f \in \Sigma^F$ of arity $n > 0$ a total function $f^{\mathcal{M}} : M^n \to M$,
3. to each propositional symbol $B \in \Sigma^P$ a value $B^{\mathcal{M}} \in \{True, False\}$, and
4. to each predicate symbol $p \in \Sigma^P$ of arity $n > 0$ a function $p^{\mathcal{M}} : M^n \to \{True, False\}$.

Furthermore, we denote the set of Σ-terms as the smallest set $\mathcal{T}$ such that

(i) $c \in \mathcal{T}$ for every constant symbol $c \in \Sigma^F$,

(ii) each variable symbol $c_i \in \mathcal{T}$ for $i = 1, 2, ...,$ and

(iii) if $t_1, ..., t_{n_f} \in \mathcal{T}$ and $f \in \Sigma^F$ with arity n_f, then $f(t_1, ..., t_{n_f}) \in \mathcal{T}$.

Conceptually, the set of terms $\mathcal{T}$ contains all the constant symbols, the variable symbols, and all applications of functions to other terms.

Let ϕ be a formula with free variables from $\overline{v} = (v_{i_1}, v_{i_2}, ..., v_{i_m})$, and let $\overline{a} = (a_{i_1}, a_{i_2}, ..., a_{i_m}) \in M^m$. We inductively define $\mathcal{M} \vDash \phi(\overline{a})$ as follows:

(i) If ϕ is $t_1 = t_2$, where $t_1, t_2 \in \mathcal{T}$, then $\mathcal{M} \vDash \phi(\overline{a})$ if $t_1^{\mathcal{M}}(\overline{a}) = t_2^{\mathcal{M}}(\overline{a})$.

(ii) If ϕ is $B \in \Sigma^P$, then $\mathcal{M} \vDash \phi(\overline{a})$ if $B^{\mathcal{M}} = True$.

(iii) If ϕ is $p(t_1, ..., t_{n_p})$ where $p \in \Sigma^P$, then $\mathcal{M} \vDash \phi(\overline{a})$ if $p(t_1^{\mathcal{M}}(\overline{a}), ..., t_{n_p}(\overline{a})^{\mathcal{M}}) = True$.

(iv) If ϕ is $\neg\psi$, then $\mathcal{M} \vDash \phi(\overline{a})$ if $\mathcal{M} \nvDash \psi(\overline{a})$.

(v) If ϕ is $(\psi \wedge \theta)$, then $\mathcal{M} \vDash \phi(\overline{a})$ if $\mathcal{M} \vDash \psi(\overline{a})$ and $\mathcal{M} \vDash \theta(\overline{a})$.

(vi) If ϕ is $(\psi \vee \theta)$, then $\mathcal{M} \vDash \phi(\overline{a})$ if $\mathcal{M} \vDash \psi(\overline{a})$ or $\mathcal{M} \vDash \theta(\overline{a})$.

(vii) If ϕ is $\exists v_j\ \psi(\overline{v}, v_j)$, then $\mathcal{M} \vDash \phi(\overline{a})$ if there is some $b \in M$ such that $\mathcal{M} \vDash \psi(\overline{a}, b)$.

(viii) If ϕ is $\forall v_j\ \psi(\overline{v}, v_j)$, then $\mathcal{M} \vDash \phi(\overline{a})$ if for every $b \in M$, $\mathcal{M} \vDash \psi(\overline{a}, b)$.

If $\mathcal{M} \vDash \phi(\overline{a})$, then we say that $\mathcal{M}$ *satisfies* $\phi(\overline{a})$. Naturally, we say that for any formula $\phi(\overline{v})$, if there exists some Σ-structure $\mathcal{M}$ and $\overline{a} \in M^n$ such that $\mathcal{M} \vDash \phi(\overline{a})$, we say that ϕ is *satisfiable*.

A *sentence* is a formula with no free variables. A Σ-*theory*[3] T is a set of sentences that uses only symbols from Σ. We say that a Σ-structure $\mathcal{M}$ is a *model* of a theory T if $\mathcal{M} \vDash \phi$ for every $\phi \in T$. We denote this as $\mathcal{M} \vDash T$.

Therefore, the problem of Satisfiability Modulo Theories can be stated as follows: given a signature Σ, a formula $\phi(\overline{v})$, does there exist some Σ-structure $\mathcal{M}$ and $\overline{a} \in M^n$ such that $\mathcal{M} \vDash T$ (i.e. $\mathcal{M}$ is a model of T) and $\mathcal{M} \vDash \phi(\overline{a})$ (i.e. $\phi(\overline{v})$ is satisfied in $\mathcal{M}$)?

We now give a brief description of some theories of interest, courtesy of [44].

Fixed-Width Bit-Vectors: Given that we are working with digital circuits, the theory of fixed-width bit-vectors is of great importance to us. There are many formulations for the theory of fixed-width bit-vectors. Generally, they all share the characteristic that constant symbols are used to represent bit-vectors and that each bit-vector has an associated *width* (i.e. the number of bits it contains). In this work, we consider the theory as specified by the SMT-LIB standard [46], which is also the theory that is used by Z3 [47], the SMT solver that SSV uses for its backend.

[3]Oftentimes abbreviated to just *theory* if the signature Σ in question is clear from context.

What differs among the different theories are the predicate and function symbols, which may include extraction, concatenation, bit-wise Boolean operations, and arithmetic operations. Reasoning at the level of bit-vectors offers a more compact representation of the problem, which lends to more efficient verification than using bit-level reasoning.

Arrays: This theory captures our intuitive notion of how arrays behave by characterizing and axiomatizing the *read* and *write* operations. Formally, we let Σ_A be the signature $(read, write)$. We consider the following sentences as the axioms that comprise the theory of arrays (with extensionality), T_A:

$$\forall a \forall i \forall v.\ (read(write(a,i,v),i) = v)$$
$$\forall a \forall i \forall j \forall v.\ (i \neq j \Rightarrow read(write(a,i,v),j) = read(a,j))$$
$$\forall a \forall b.(\forall i.(read(a,i) = read(b,i)) \Rightarrow a = b)$$

Model Checking

Model checking is a verification technique that provides an algorithmic means of determining whether an abstract model–representing, for example, a hardware or software design–satisfies a formal specification expressed as a temporal logic formula [48]. It systematically explores all possible states of a system to check properties like safety ("nothing bad happens") and liveness ("something good eventually happens"). If the property fails, model checking provides a counterexample showing exactly how the system can reach an undesired state. It is widely used in hardware verification, protocol analysis, and concurrent software systems because it can catch subtle bugs that are hard to detect through testing alone.

Bounded Model Checking (BMC) is a variant of model checking where, instead of exploring all possible states, BMC unrolls the system's transition relation for a limited number of steps and encodes it as a logical formula, typically solved with a SAT or SMT solver. If the formula is satisfiable, a counterexample of that length exists; if not, the property holds within the bound. BMC is especially useful for finding bugs quickly in hardware designs and software with loops, though it cannot prove correctness beyond the chosen bound.

Symbolic Simulation

Symbolic simulation is a verification technique where, instead of running a system with concrete input values, the system is executed using symbolic inputs that represent many possible values at once [49]. The outputs are expressed as symbolic expressions, allowing the analysis of multiple execution paths simultaneously. This approach is useful for detecting errors in hardware or software designs, exploring corner cases efficiently, and generating test cases.

Fixed Point Computation

Fixed-point computation is a method used in program analysis and formal verification to find a stable state of a system or function—one where applying the function again produces the same result. In verification, it's often used to compute properties like reachable states, invariants, or loop behaviors by iteratively applying a transition relation until no new states are discovered. This technique is fundamental for reasoning about recursive systems, loops, and other repetitive behaviors in both hardware and software models. This kind of reasoning is often implemented by using mathematical objects called constrained Horn clauses, and it has shown to have a number of applications to program verification [50], [51].

3.4 Language Extension

When it comes to security verification, BTOR2 programs face several fundamental limitations. For one, the format has no support for cryptographic primitives. But even if we were to add native cryptographic instructions to BTOR2, how would we even get the security-relevant information in the first place? In this section, we describe the BTORSEC, our extension of the BTOR2 language, in more detail. Section 3.4.1 provides an overview of the new abstract cryptographic instructions of the language. Section 3.4.2 introduces the SECSPECS, a construct that allows users to easily specify security-relevant metadata about a hardware design.

3.4.1 Cryptographic Instructions

BTORSEC adds 15 new instructions to the base BTOR2 language. These include a new sort, `acsort`, which represents abstract cryptographic objects like messages and keys. It also includes primitives for key generation, encryption, and more. A summary of all the new instructions can be found in Table 3.1. The rest of this subsection will be dedicated to explaining the functionality of each of these instructions in further detail. Readers who desire a more formal description of the syntax of BTORSEC are directed to Section E.1.

General

These instructions are the core foundations of any BTORSEC design. They include a new sort declaration, `acsort`, as well as the `acnondet`, `public`, and `confidential` instructions. Below, we provide a detailed breakdown of this class of cryptographic instructions.

```
<lid> acsort <width>
```

The first instruction we cover is the `acsort`, which stands for Abstract Cryptography Sort. This sort declaration instruction is meant to represent abstract cryptographic types such as messages and keys. This is a useful construct when reasoning about cryptography at a higher level, where details such as bit-vector widths are not too relevant.

general	
`acsort`	abstract cryptography sort declaration
`acnondet`	nondeterministic acsort
`public`	public signal declaration
`confidential`	confidential signal declaration
key generation	
`keygen`	key generation
`keydrv`	key derivation
symmetric cryptography	
`symenc`	symmetric encryption
`symdec`	symmetric decryption
asymmetric cryptography	
`asymenc`	asymmetric encryption
`asymdec`	asymmetric decryption
message authentication codes	
mac	create message authentication code
vfmac	verify message authentication code
digital signatures	
sign	create digital signature
vfsign	verify digital signature
other	
`hash`	compute hash of a value

Table 3.1: Cryptographic Instructions Supported by BTORSEC.

```
<lid> acnondet <sid>
```

The `acnondet` (Abstract Cryptography Nondeterministic Value) instruction is a special instruction that returns a nondeterministic ACSort value. It is used to over-approximate an ACSort value that is operated on by non-cryptographic operations. The reason we have this over-approximation is that we cannot make any assumptions on the values produced by primitives like `symenc` and `symdec`.

`<lid> public <sid> <inst>`

The `public` instruction is used to declare a signal as public. The `inst` parameter must be the `lid` of another instruction, which will be made public–in other words, observable to an adversary. While the threat model of BtorSec is left unspecified, this is meant to capture the notion of an adversary being able to observe only a subset of the signals in a given hardware design.

`<lid> confidential <sid> <inst>`

The `confidential` instruction is used to declare a signal as confidential. A confidential signal is a signal whose value an adversary should not be able to infer. The `inst` parameter must be the `lid` of another instruction, which will be marked as confidential.

Key Generation

Next, we discuss the key generation primitives supported by BtorSec. They include the `keygen` and `keydrv` functions, which represent key generation and key derivation functions, respectively.

`<lid> keygen <sid>`

The `keygen` instruction represents a generic key generation function or algorithm. It generates a single key of the sort specified by `sid`.

`<lid> keydrv <sid> <source>+`

The `keydrv` instruction represents a function that derives a new key from one or more `source` signals. It generates a single key of the sort specified by `sid`.

The purpose of this instruction is to model protocols where the generated value of a key depends on other values, such as other keys. For instance, in asymmetric cryptography protocols, the public key is often a function of the private key. Therefore, we can use the `keygen`, `keydrv` and `public` instructions in conjunction to model abstract public/private key pairs.

Symmetric Cryptography

These instructions are used to represent symmetric encryption primitives, such as encryption and decryption under a shared key. They include the instructions `symenc` and `symdec`, made to abstract symmetric encryption and decryption functions, respectively.

```
<lid> symenc <sid> <msg> <key>
```

The `symenc` instruction represents an abstract symmetric encryption operation. The `msg` parameter represents the ciphertext to be decrypted. The `key` parameter represents the shared key.

```
<lid> symdec <sid> <ctxt> <key>
```

The `symdec` is the counterpart to `symenc`. The structure of the two instructions is essentially the same. The `lid` and `sid` parameters remain unchanged. The parameters `ctxt` and `key` represent the ciphertext and the key, respectively.

Asymmetric Cryptography

These instructions are meant to represent asymmetric cryptographic primitives. These include the `asymenc` and `asymdec` instructions, which represents asymmetric encryption and decryption. While very similar to the `symenc` and `asymdec` instructions, these operate under a different model of cryptography altogether. Instead of using a single shared key for encryption and decryption, `asymenc` and `asymdec` use a public/private key pair. These instructions are meant to be used in conjunction with the `keygen`, `keydrv`, and `public` instructions outlined previously.

```
<lid> asymenc <sid> <msg> <pubkey>
```

The `asymenc` instruction represents an asymmetric encryption function. The `msg` parameter represents the ciphertext to be decrypted. The `pubkey` parameter represents the recipient's public key.

```
<lid> asymdec <sid> <ctxt> <privkey>
```

The `asymdec` instruction represents an asymmetric decryption function. The parameters `ctxt` and `privkey` represent the ciphertext and the private key, respectively.

Message Authentication Codes

We now describe the instructions that BTORSEC includes to support modeling message authentication codes (MACs). This category includes the instructions `mac` and `vfmac`, which are used to create and verify MAC tags, respectively.

```
<lid> mac <sid> <msg> <key>
```

The `mac` instruction is used to create a tag on a given message by using a shared key, which can be verified by other users who have access to the key with the `vfmac` instruction. The instruction takes in a `msg` and a `key`. Then, it outputs a tag that is tied to the message.

```
<lid> vfmac <sid> <msg> <tag> <key>
```

The `vfmac` instruction is used to verify tags created by the `mac` instruction. It takes in a `msg`, `tag`, and a `key`. By convention, `tag` should be the `lid` of some `mac` instruction. This instruction should output a 1-bit bit-vector, where 1 indicates that the (`msg`, `tag`) was successfully verified, and 0 indicates that the verification failed.

Digital Signatures

We now describe the instructions used to model digital signatures. These instructions closely resemble MACs. The main difference is that digital signatures use a signing/verifying key pair instead of a symmetric key.

```
<lid> sign <sid> <msg> <skey>
```

The `sign` instruction is used to create a digital signature of a given message. The `msg` parameter is the message to be signed, and the `skey` parameter corresponds to a user's secret signing key.

```
<lid> vfsign <sid> <msg> <signature> <vkey>
```

The `vfsign` instruction is used to verify a signature produced by the `sign` instruction. The `msg` parameter corresponds to the message to be verified. The `signature` parameter corresponds to the digital signature on the message, signed with the secret signing key. The `vkey` parameter corresponds to the public verification key that can be used to verify the authenticity of the message.

Other

Finally, we describe the other kinds of primitives that are typically used in protocols. In this category, we have the `hash` instruction, intended to represent generic hash functions.

```
<lid> hash <sort> <source>+
```

The `hash` instruction models hash functions, which are commonplace in cryptographic protocols. It takes one or more `source` parameters, and computes the hash of these values, of sort `sort`.

3.4.2 SECSPEC Files

While users could theoretically write their own BTORSEC programs, this would soon prove to be a grueling and tedious process. The BTORSEC language is primarily intended to be an intermediate representation for model-checking software, much like BTOR2. Currently, there exist tools that can compile a Verilog file into an equivalent BTOR2 representation, such as the Yosys Open Synthesis Suite [52]. However, no tool could realistically compile a BTORSEC file in the same manner. This is due to the fact that Verilog files lack any explicit security metadata that could be used to create an accurate BTORSEC representation. One could potentially add security metadata to the Verilog file directly, but such a paradigm would likely be difficult to maintain in the long run, especially given the ever-changing nature of hardware designs as power, performance, and area requirements becoming more strict over time.

Our dilemma above calls for a solution that allows us to annotate Verilog files with security-relevant metadata, yet also maintains the original Verilog design intact. Additionally, we would like our solution to be easy to read, maintain, and update as hardware designs evolve. To this end, we introduce the BTORSEC Security Specification file format (stylized as SECSPEC). SECSPECS, allow users users to define security-related metadata, such as which modules represent cryptographic primitives, which signals are public, and more. When provided with a Verilog file and its associated SECSPEC, the BTORSEC compiler can automatically create a BTORSEC file that represents the RTL with the abstractions on the cryptography.

As an illustrative example, we will be using the Verilog design found in Figure 3.3. This design consists of a simple combinational module that takes in some input data (`din`), two keys (`k1` and `k2`), and outputs the encryption of the input data under both keys. To encrypt the data, the design instantiates two `encryptor` modules, whose details are omitted for brevity.

The actual implementation of the `encryptor` module could be very large and complicated, making verification very difficult. However, just like any other encryption primitive, `encryptor` is simply computing $Enc(din, k)$. Therefore, we would like to abstract away the details and replace the module instance with a `symenc` instruction instead.

A reasonable notion of security for such a circuit is that an adversary should not be able to deduce the value of the input data by only observing the outputs. In other words, if we allow an adversary to only observe the values of `dout`, they should not be able to infer the value of `din`, assuming that they do not also know `k1` and `k2`. We will now show how we can capture this informal idea of security by writing a SECSPEC to go along with this design.

Structure of SECSPECS

We now describe the general structure of SECSPECS, with a focus on our illustrative example above. At its core, a SECSPEC is a structured JSON file consisting of an array of module descriptions. A module description either provides information about the top-level module or about cryptographic modules.

A top-level module description provides information about the design at large. This description first specifies which signals should be made publicly observable to an adversary. This signals to the BTORSEC compiler that a `public` instruction should be created for each of the signals. Figure 3.4 shows the top-level module description corresponding to our example design.

```
/* A simple combinational module that uses an encryptor */
module example (
    input [31:0] din,
    input [31:0] k1,
    input [31:0] k2,
    output [31:0] dout
);

    wire [31:0] intermediate;

    // First encryption of the input data
    encryptor enc1 (
        .din(din),
        .k(k1),
        .dout(intermediate)
    );

    // Second encryption of the intermediate data
    encryptor enc2 (
        .din(intermediate),
        .k(k2),
        .dout(dout)
    );

endmodule

module encryptor (
    input [31:0] din,
    input [31:0] k,
    output [31:0] dout
);

    /* Implementation details omitted */

endmodule
```

Figure 3.3: BTORSEC Program of the Double Encryptor

```
    {
        "name":"example",
        "modtype":"top",
        "query":["din"],
        "public":["dout"]
    },
```

Figure 3.4: A Sample Top-Level Module

The `name` field is a string that corresponds to the name of the top-level module. In our case, this corresponds to the `example` module, so we populate this field with the value `"example"`. Next, the `modtype` field is a string that specifies the kind of module we are working with. To specify that `example` is a top-level module, we populate this field with they keyword `"top"`. The `confidential` field is an array of strings that corresponds to the signal values that should remain secret throughout the execution of the hardware. In our example, we do not want the value of the plaintext input data to be leaked to an observer, so we make an array with a single value of `"din"`. Finally, the `public` field is another array of strings used to specify which signals should be made observable to an outsider. To match our informal security specification, we will make only the `dout` signal public.

A cryptographic module description, in contrast to a top-level module description, provides information about module instances that are to be abstracted. Each cryptographic module description specifies a module instance that should be abstracted. At the very least, it must specify the name of the instance that is to be abstracted, the type of primitive it should be abstracted to, and the relevant signals. Figure 3.5 shows the cryptographic module description corresponding to the `enc1` instance of the `encryptor` module.

```
    {
        "name":"enc1",
        "modtype":"symenc",
        "delay":0,
        "plaintext":"din",
        "key":"k",
        "ciphertext":"dout"
    },
```

Figure 3.5: A Sample Cryptographic Module Description

Every cryptographic module description shares the `name`, `modtype`, and `delay` fields. The `name` field must correspond to the name of the module instance. Since we are specifically trying to abstract the `enc1` module instance, we should make sure that this field has the value `enc1`. Next, the `modtype` of this module should be `symenc`, as the encryptor is ultimately performing a symmetric encryption operation (and thus should be represented by a BtorSec `symenc` instruction). The `delay` field is optional, and it is used to specify the delay (in cycles) that this module takes. If omitted, the compiler will assume that the module is combinational (and thus has 0 cycles of delay). In our case, the `encryptor` module is indeed combinational, so we assign this field the value 0.

Now, we turn our attention to the modtype-specific fields of the description. These latter fields will vary depending on the modtype. The `plaintext`, `key`, and `ciphertext` fields are all specific to the `symenc` modtype. In our example, the plaintext that is being encrypted by `enc1` corresponds to the `din` signal, while `k1` is the key and the output (ciphertext) of the module corresponds to the `intermediate` signal. Intuitively, we are telling the BtorSec compiler how to interpret each of the ports of `enc1`.

Ultimately, the SECSPEC that fully describes our design is shown in Figure 3.6.

```
[
    {
        "name":"example",
        "modtype":"top",
        "query":["din"],
        "public":["dout"]
    },
    {
        "name":"enc1",
        "modtype":"symenc",
        "delay":0,
        "plaintext":"din",
        "key":"k",
        "ciphertext":"dout"
    },
    {
        "name":"enc2",
        "modtype":"symenc",
        "delay":0,
        "plaintext":"din",
        "key":"k",
        "ciphertext":"dout"
    }
]
```

Figure 3.6: A Complete SECSPEC

For the interested reader, we provide the full JSON Schema that all SECSPECs must adhere to in Appendix E.2.

3.5 Compilation

We now shift our focus to explaining the BTORSEC compilation workflow in further detail. Our compiler is based on previous work by Dobis, and notably we extend her `btor2-opt` Python package [53]. We begin by providing a birds-eye view of the compilation process. Then, we dedicate the rest of the section to explaining the behavior of the compiler passes that transform a BTOR2 program into a BTORSEC file.

3.5.1 Overview

Before diving into the details of the compiler, we give a birds-eye view of how BTORSEC programs are compiled. To a user, the BTORSEC compiler can be seen as a black box that takes in a Verilog design and a SECSPEC and outputs a BTORSEC file. Thus, the only obligations on the user's part are to write the RTL in Verilog and to provide the security metadata of the RTL with a corresponding SECSPEC. The intended workflow is illustrated in Figure 3.7.

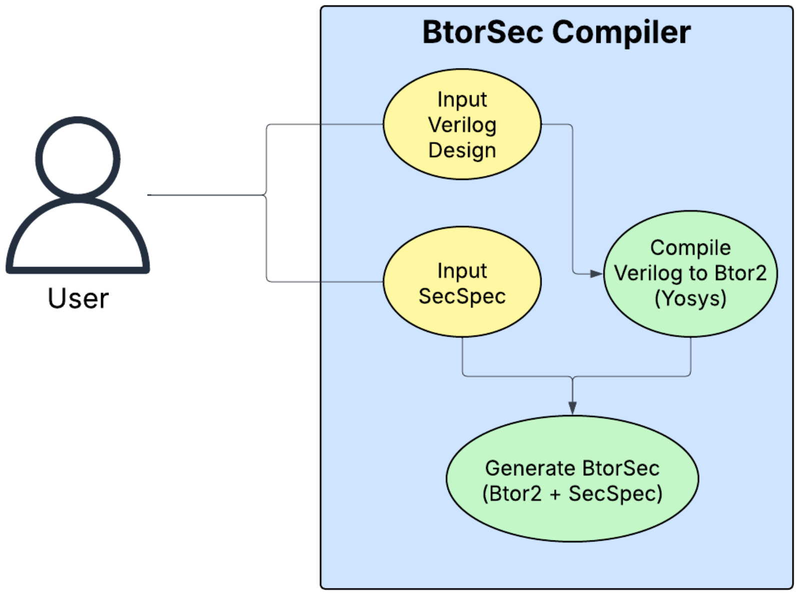
BtorSec Compiler
User
Input Verilog Design
Input SecSpec
Compile Verilog to Btor2 (Yosys)
Generate BtorSec (Btor2 + SecSpec)

Figure 3.7: The BTORSEC Compilation Pipeline for Users

As can be seen in the diagram, a user must provide the BTORSEC compiler with both the Verilog design of the hardware as well as an associated SECSPEC. Upon receiving these two inputs, the compiler will first convert the Verilog into its BTOR2 representation. Then, it will use the security information provided in the SECSPEC to transform the BTOR2 representation of the RTL into its final BTORSEC form.

3.5.2 Compiler Passes

We now proceed with a closer analysis of how our compiler transforms a BTOR2 representation of the RTL and a SECSPEC into a BTORSEC file. At a high level, the compiler achieves this by running through the program several times and transforming it little by little, until the desired result is achieved. We call each run of the compiler through the program a "pass."

Each compiler pass achieves a specific purpose. First, the AbstractModule pass uses the SECSPEC information to create a new cryptographic instruction that correctly abstracts the module's behavior. This pass also makes sure that all relevant signals use the correct AC Sort. Finally, the Cleanup pass removes any instructions that are no longer relevant to the design of the RTL. A diagram of the passes can be seen in Figure 3.8. In what follows, we will describe the functionality of each compiler pass in more detail.

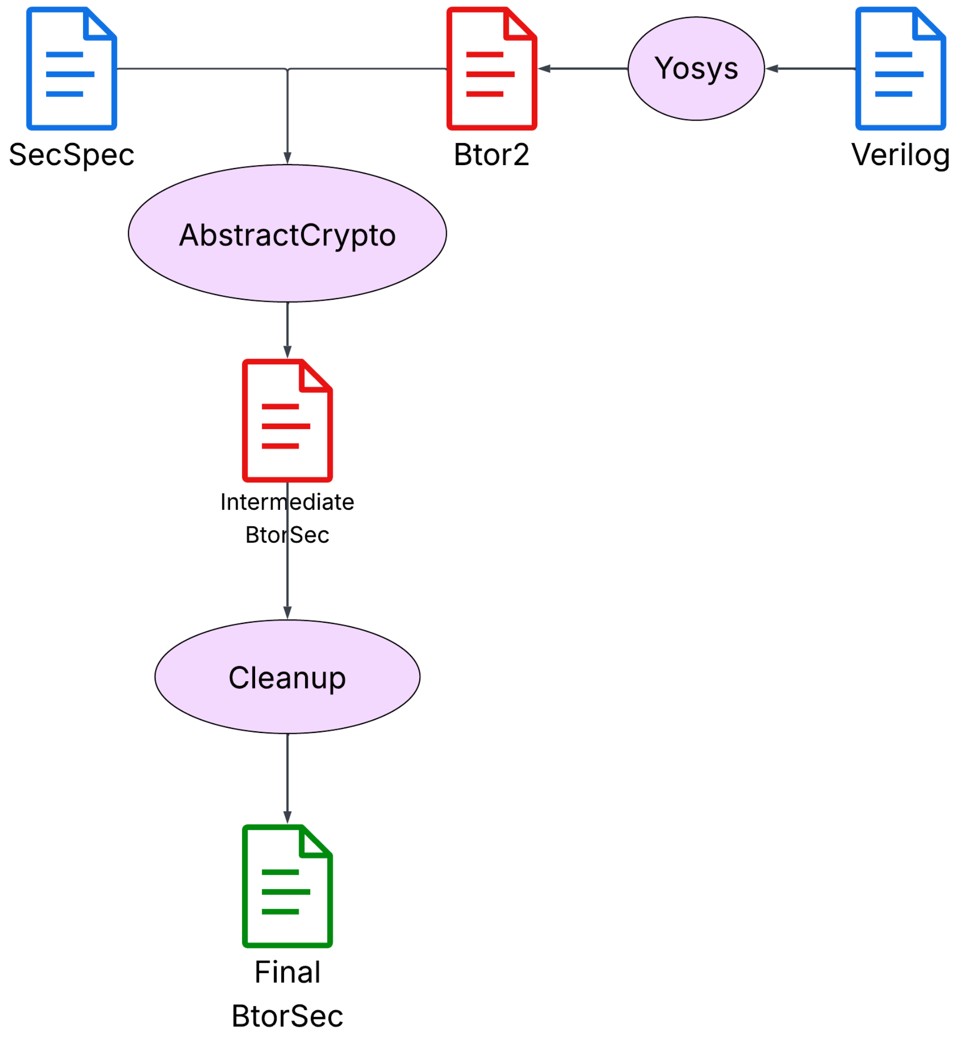

Figure 3.8: Overview of Compiler Passes

AbstractCrypto Pass

The AbstractCrypto pass takes a BtorSec program along with a collection of module specifications indicating which modules implement different cryptographic primitives. The pass then "replaces" the module with an abstract cryptographic instruction, such as `symenc` or `symdec`.

Let us illustrate this functionality with an example. Consider the Verilog implementation of a sequential encryptor design in Figure 3.9: Put briefly, this design will take some data (`din`) and a key (`k`), and compute the encryption of the data under the key. It will then will output the encrypted value the following cycle.

```
module top (
    input clk,
    input [31:0] din,
    input [31:0] k,
    output [31:0] dout
);

    reg [31:0] dout_reg;
    wire [31:0] enc_out;

    // Instantiation of encryptor
    encryptor enc (
        .din(din),
        .k(k),
        .dout(enc_out)
    );

    // Sequential logic
    always @(posedge clk) begin
        dout_reg <= enc_out;
    end

    assign dout = dout_reg;

endmodule


module encryptor (
    input [31:0] din,
    input [31:0] k,
    output [31:0] dout
);

    assign dout = din ^ k;

endmodule
```

Figure 3.9: A Sequential Encryptor Design

The actual encryption is performed by `enc`, which is an instantiation of the `encryptor` module. While the `encryptor` module is rather simple in this example, other designs may instead implement cryptographic primitives such as AES encryption with different stream cipher modes, which will inevitably have more complex circuitry. At any rate, regardless of *how* these modules achieve their task, they are all essentially trying to perform a *symmetric encryption* operation, a fact that we leverage when performing the abstraction.

The module above can be compiled into the following BTOR2 program (Figure 3.10 via Yosys:

```
; BTOR description generated by Yosys 0.50+56 (git sha1 176131b50, aarch64-apple-darwin23.5-clang++ 18.1.8 -fPIC -O3) for module top.
1 sort bitvec 1
2 input 1 clk ; sequential_encryptor/top.v:2.11-2.14
3 sort bitvec 32
4 input 3 din ; sequential_encryptor/top.v:3.18-3.21
5 input 3 k ; sequential_encryptor/top.v:4.18-4.19
6 state 3 dout_reg
7 output 6 dout ; sequential_encryptor/top.v:5.19-5.23
8 uext 3 4 0 enc.din ; sequential_encryptor/top.v:29.18-29.21
9 xor 3 4 5
10 uext 3 9 0 enc.dout ; sequential_encryptor/top.v:31.19-31.23
11 uext 3 5 0 enc.k ; sequential_encryptor/top.v:30.18-30.19
12 uext 3 9 0 enc_out ; sequential_encryptor/top.v:9.17-9.24
13 next 3 6 9
; end of yosys output
```

Figure 3.10: BTOR2 Representation of the Sequential Encryptor

We focus our attention on lines `8`, `10`, `11`, and `12`. The purpose of these `uext` instructions is to create aliases[4] of the signals that they are extending (`enc.din`, `enc.dout`, etc.). These aliases are crucial in the implementation of the AbstractCrypto pass, as they give us the `lids` of the relevant input and output signals of the different module instantiations present within the top-level modules. Indeed, the AbstractCrypto pass first collects these `lids` and then creates an abstract cryptographic instruction (in this case a `symenc`) using the relevant input `lids`. Simultaneously, this pass will replace any occurrences of the output `lid` with the `lid` of this new abstract cryptographic instruction. We can see the result of applying an AbstractCrypto pass in the BTORSEC program in Figure 3.11 below:

As the reader can appreciate, line `10` now contains a new instruction: `10 symenc 1 5 6`. This instruction does exactly what one expects: it encrypts the value of `5` (`din`) with the value of `6` (`k`). Moreover, we can see that any instructions in the BTOR2 program that used the value of `9` (the old output of the `enc` module instance—`enc.dout`) have now been replaced with `10`.

A particularly attentive reader has probably noticed that several instructions, not just the `symenc` instructions, are now of sort `acsort`. This is because the sorts of a BTORSEC program must be consistent, so the sorts of any dependents (and anti-dependents) of the `symenc` instruction must also change to `acsort`. Determining which instructions must have

[4]Upon closer inspection, we can see that these instructions are performing an unsigned extension of zero bits—meaning that they are simply copying the value of that signal.

```
1 sort acsort -1
2 sort bitvec 1
3 input 2 clk
4 sort bitvec 32
5 input 1 din
6 input 1 k
7 state 1 dout_reg
8 output 7
9 uext 1  5  0  enc.din
10 symenc 1  5  6
11 acnondet 1
12 uext 1  10  0  enc.dout
13 uext 1  6  0  enc.k
14 uext 1  10  0  enc_out
15 next 1  7  10
```

Figure 3.11: BTORSEC Program of the Sequential Encryptor

their sort changed is implemented in the MarkInstructions pass, which is called from within the AbstractCrypto pass.

MarkInstructions Pass

The MarkInstructions pass is a helper pass that is used by the AbstractCrypto pass when propagating the sorts across. The goal of this pass is to mark any dependents and anti-dependents of a given instruction. It achieves this by first iterating through the program (i.e. the list of instructions) and creating an undirected graph. In this graph, each instruction is represented as a node. Two nodes u and v share an edge if the instruction corresponding to u has the `lid` of the instruction corresponding to v as one of its operands, or vice versa. There are two exceptions to this rule, however. The first exception is that `sort` instructions are isolated no matter what. The second exception is that we do not add an edge between an `ite` instruction and the `lid` corresponding to its condition. The reason behind this is that we do not want to change the sort of a condition to an `acsort` unnecessarily, and it does not make sense for a conditional to be of a type different from 1-bit bit-vector.

To illustrate the functionality of the MarkInsts pass, consider the conditional encryptor design found in Figure 3.12. The design is almost identical to the sequential encryptor of Figure 3.9. The main differences are that this circuit is entirely combinational and that we have a new signal, `should_encrypt`, which decides whether `dout` takes on the value produced by the `enc` module or if it outputs `din` in plaintext.

The corresponding BTOR2 program can be seen in Figure 3.13, and the resulting graph created by the MarkInsts pass is shown in Figure 3.14. For better visual clarity, the graph is color-coded as follows: `input` instructions are marked as green, `sort` instructions are marked

```
module top (
    input clk,
    input should_encrypt,
    input [31:0] din,
    input [31:0] k,
    output [31:0] dout
);

    wire [31:0] enc_out;

    encryptor enc (
        .din(din),
        .k(k),
        .dout(enc_out)
    );

    assign dout = should_encrypt ? enc_out : din;

endmodule


module encryptor (
    input [31:0] din,
    input [31:0] k,
    output [31:0] dout
);

    assign dout = din ^ k;

endmodule
```

Figure 3.12: Conditional Encryptor

as light blue, `ite` instructions are marked as yellow, and `output` instructions are marked as pink.

As the reader can appreciate from the graph, the nodes in light blue, corresponding to `sort` instructions (`1` and `3`) are isolated. This is aligned with our first exception. Furthermore, node `6` is also isolated, despite being one of the operands to the instruction `ite 3 6 7 4`. This is because `6` is the condition of the `ite`, meaning that it cannot be part of this graph, in accordance to the second exception we outlined above.

Cleanup Pass

The final step in compiling a BTORSEC program is to eliminate all instructions that are no longer needed. Specifically, this includes any instructions that encode the internal logic of a module which has been replaced by an abstract cryptographic operation. We achieve this by removing any instructions found in the path from the inputs to the outputs of the replaced

```
; BTOR description generated by Yosys 0.50+56 (git sha1 176131b50,
   ↪ aarch64-apple-darwin23.5-clang++ 18.1.8 -fPIC -O3) for module
   ↪ top.
1 sort bitvec 1
2 input 1 clk ; enc_pln_mux/top.v:2.11-2.14
3 sort bitvec 32
4 input 3 din ; enc_pln_mux/top.v:4.18-4.21
5 input 3 k ; enc_pln_mux/top.v:5.18-5.19
6 input 1 should_encrypt ; enc_pln_mux/top.v:3.11-3.25
7 xor 3 4 5
8 ite 3 6 7 4
9 output 8 dout ; enc_pln_mux/top.v:6.19-6.23
10 uext 3 4 0 enc.din ; enc_pln_mux/top.v:23.18-23.21
11 uext 3 7 0 enc.dout ; enc_pln_mux/top.v:25.19-25.23
12 uext 3 5 0 enc.k ; enc_pln_mux/top.v:24.18-24.19
13 uext 3 7 0 enc_out ; enc_pln_mux/top.v:9.17-9.24
; end of yosys output
```

Figure 3.13: BTOR2 Program of Encryptor with MUX

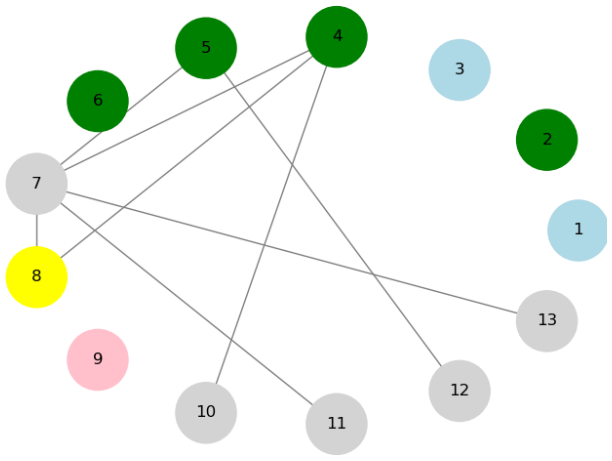

Figure 3.14: Dependency Graph Created by MarkInsts Pass

module.

3.6 Verification

So far, we have introduced the BTORSEC language and described how a user can compile their own BTORSEC programs. But a BTORSEC program on its own is useless; the whole point of BTORSEC was to provide a way to verify that our RTL is correctly employing cryptographic primitives. This section provides a brief introduction to the recursively-named SSV Security Verifier, our prototype bounded model checker for confidentiality queries in BtorSec programs. This section is meant to provide enough detail to make our approach clear and to give the reader an idea of how SSV works. Readers interested in the theoretical and mathematical foundations behind SSV are directed to Appendix C. Those interested in the implementation of SSV are encouraged to check out our GitHub repository [54].

3.6.1 Preliminaries

We now proceed to explain the basic formalisms required to understand the verification methodology of SSV.

Recall that one of the new additions of BtorSec over Btor2 is the Abstract Cryptography Sort (`acsort`). This sort was included in the language to make it easier for us to reason about symbolic terms that involve cryptographic operations. Let $\mathbb{A}$ denote the set of these abstract cryptographic values, and let $\mathbb{B}$ denote our base domain, consisting of bit-vectors, arrays, and other such concrete values. Then our universe, $\mathbb{U}$ will be $\mathbb{U} = \mathbb{A} \uplus \mathbb{B}$. Our verification methodology will consider hardware designs with variables over our universe. For any given hardware design $\mathcal{D}$, we denote its finite set of variables as $\mathcal{V}$.

Definition 3.1. State (σ): A state $\sigma : \mathcal{V} \to \mathbb{U}$ is a mapping[5] from the set of variables to values in our universe.

Definition 3.2. Trace (τ): A trace τ is a (possibly infinite) sequence of states $\langle \sigma_0, \sigma_1, \ldots \rangle$. τ is *finite* if $\tau = \langle \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \rangle$ for some n, in which case its length is n.

3.6.2 Threat Model

The threat model of BtorSec programs is intentionally left largely unspecified. Beyond allowing designers to label a subset of signals as `public` or `confidential`, the language makes no assumptions about the nature or capabilities of a potential adversary. This flexibility enables model-checking tools to define and enforce a wide range of adversaries and threat models. The design reflects the fact that different hardware designers face distinct security concerns. We now define the threat model assumed by SSV, our bounded model-checking tool.

Intuititon

Before moving on with formalisms, we provide some intuition on the threat model we consider. The SSV adversary can be thought of as a passive but curious observer of the system. By this, we mean that the adversary is allowed to observe certain parts of the hardware and later use their observations to try to deduce confidential values. However, they are not

[5]We assume that any state is well-typed, meaning that it only maps $\mathbb{A}$-typed variables to values in $\mathbb{A}$ and $\mathbb{B}$-typed variables to values in $\mathbb{B}$

allowed to interfere with the execution of the hardware. The process consists of two distinct phases: an observation phase and a computation phase.

Observation phase: This phase comes first, and it lasts for a predefined number n of cycles. During this phase, the adversary is allowed to see the values of the public variables of the system. However, they cannot observe any other parts of the system.

Computation phase: In this phase, the adversary is allowed to perform computation on the values that they observed during the observation phase. The goal is to deduce any confidential values. The adversary's computation is limited to using a the set of rules $\mathcal{R}$ with which it was parameterized. Moreover, it is allowed to apply these rules to their known terms a maximum of t times, where t is a user-specified bound.

Formalizing the Threat Model

We formalize the above intuition as follows. Let $\mathcal{A} = (Obs, \mathcal{R}, t)$ be an adversary. Let $\tau = \langle \sigma_0, \sigma_1, ..., \sigma_{n-1} \rangle$ be a trace of length n.

Definition 3.3. Adversary: We define an adversary as the tuple $\mathcal{A} = (Obs, \mathcal{R}, t)$, where $Obs : \Sigma \to 2^{\mathbb{U}}$ denotes the adversary's observation function, $\mathcal{R}$ is a set of rules of inference, and $t \in \mathbb{N}$.

Definition 3.4. Adversary Observations ($\mathcal{O}$): The set of values observed by $\mathcal{A}$ in a given trace τ is

$$\mathcal{O} = \{Obs(\sigma_i) \mid \sigma_i \in \tau\}.$$

Definition 3.5. Adversary Computation Function ($\mathsf{AdvComp}$): The adversary computation function, $\mathsf{AdvComp}$ takes in a set of adversary observations and returns a set of terms. Formally, $\mathsf{AdvComp}_{\mathcal{R}} : 2^{\mathbb{U}} \to 2^{\mathbb{U}}$. The knowledge computed in one round is given by:

$$\mathsf{AdvComp}_{\mathcal{R}}(\mathcal{O}) = \{\mathsf{head}(\mathsf{r}[\alpha]) \mid \mathsf{r} \in \mathcal{R},\ \alpha : \mathsf{vars}(\mathsf{r}) \to \mathbb{A},\ \mathsf{body}(\mathsf{r}[\alpha]) \subseteq \mathcal{K}\}.$$

Definition 3.6. Adversary Knowledge ($\mathsf{AK}_{\mathcal{R}}$): If $\mathcal{A}$ is allowed t rounds of computation, then the final set of adversary knowledge, $\mathsf{AK}_{\mathcal{R}}(\tau)$ is given by

$$\mathsf{AK}_{\mathcal{R}}(\tau) = \mathsf{AdvComp}^t_{\mathcal{R}}(\mathcal{O}),$$

where $\mathsf{AdvComp}^t_{\mathcal{R}}(\tau) = \underbrace{\mathsf{AdvComp}_{\mathcal{R}}(\mathsf{AdvComp}_{\mathcal{R}}(\cdots \mathsf{AdvComp}_{\mathcal{R}}(\mathcal{O}) \cdots))}_{t \text{ times}}$.

Problem Statement

With the above formalisms, we can state our definition of security as follows:

Definition 3.7. Security: Let $\mathcal{C}$ be a set of confidential values. We say that our design $\mathcal{D}$ is *secure* if

$$\mathcal{C} \cap \mathsf{AK}_{\mathcal{R}}(\tau) = \varnothing$$

for every feasible τ such that $\tau_0 \in \Sigma_0$.

A deeper dive into this threat model can be found in Section C.3.

3.6.3 Approach

Now that we have defined our threat model and our notion of security, we can proceed to describe how SSV conducts verification. To effectively model both the observation and computation phases, our solver uses a combination of two well-known techniques in security verification: symbolic simulation and fixed-point computation. In what follows, we explain how SSV uses symbolic simulation to simulate the observation phase and fixed-point computation to model the computation phase.

Observation Phase (Symbolic Simulation)

For the observation phase, we symbolically simulate the execution of the hardware for a fixed number t of cycles. Symbolic simulation allows us to consider many possible traces at once, drastically reducing the verification time when compared to concretely executing the design $\mathcal{D}$. In a nutshell, our approach to symbolic simulation is the following:

Definition 3.8. Symbolic Execution Context (γ): Let σ be a state, ϕ be a Boolean formula over variables and constants in $\mathbb{B}$, and $\mathcal{K} \subseteq \mathbb{A}$. Then $\gamma = (\sigma_a, \phi_b, \mathcal{K})$ is a symbolic execution context. σ_a denotes the assignments of $\mathbb{A}$-typed variables in this context, whereas ϕ_b denotes a path condition on the $\mathbb{B}$-typed variables. $\mathcal{K}$ is the set of all attacker-observed values in the symbolic execution thus far. We use the symbol Γ to denote the set of all such symbolic execution contexts.

The main advantage of using these contexts is that they allow us to consider many possible states as once, so long as they satisfy ϕ.

Definition 3.9. Symbolic Transition Relation: We define $\rightsquigarrow \subseteq \Gamma \times \Gamma$ as the symbolic transition relation of the circuit. $(\gamma_1, \gamma_2) \in \rightsquigarrow$ if we can transition from γ_1 to γ_2, as dictated by the semantics of the design $\mathcal{D}$. For readability, we use infix notation and write $\gamma_1 \rightsquigarrow \gamma_2$ instead.

Definition 3.10. Symbolic Trace (π): A symbolic trace π is a (possibly infinite) sequence of symbolic execution contexts $\langle \gamma_0, \gamma_1, \ldots \rangle$. We impose the constraint that $\forall i. \gamma_i \rightsquigarrow \gamma_{i+1}$. π is *finite* if $\pi = \langle \gamma_0, \gamma_1, \ldots, \gamma_{n-1} \rangle$ for some n, in which case its length is n.

Our approach to symbolic simulation is as follows. We begin with an initial set of symbolic contexts Γ_0. Then, for n cycles, we compute $\Gamma_{i+1} = \{\gamma_{i+1} \mid \gamma_i \in \Gamma_i \wedge \gamma_i \rightsquigarrow \gamma_{i+1} \wedge \mathsf{SAT}(\gamma_{i+1})\}$. This will give us a set of final symbolic contexts, which represents all the possible executions of our design.

Computation Phase (Fixed Point Computation)

Once we reach our final set of symbolic execution contexts, Γ_n, for each context $(\sigma_a, \phi, \mathcal{K}) \in \Gamma_n$, we will compute $\mathsf{AdvComp}^t_{\mathcal{R}}(\mathcal{K})$ and check whether $\mathsf{AdvComp}^t_{\mathcal{R}}(\mathcal{K}) \cap \mathcal{C} = \varnothing$.

3.6.4 Evaluation

We evaluate SSV on a number of hand-written hardware modules that use cryptographic primitives. We evaluate these with two different models of cryptography: the Perfect Encryption Model, which closely resembles the formal model by Dolev and Yao [55], and the XOR Model, in which all symmetric encryption and decryption primitives are simple XOR operations. The source code for these modules can be found in [54].

Module	**Cycles**	**Bound**	**Sim Time**	**FP Time**	**Result**
`comb_enc`	2	2	1.678ms	134.2ms	secure
`enc_pln_mux`	2	2	100.5ms	12.4ms	`din` deduced
`sequential_encryptor`	2	2	4.7ms	18.4ms	secure
`double_encryption`	2	2	6.4ms	143.5ms	secure
`perf_ok_xor_bad`	2	2	5.8ms	1846.8ms	secure

Table 3.2: Verification Results for Perfect Encryption Model

Module	**Cycles**	**Bound**	**Sim Time**	**FP Time**	**Result**
`comb_enc`	2	2	1.1ms	7.8s	secure
`enc_pln_mux`	2	2	91.1ms	9.2ms	`din` deduced
`sequential_encryptor`	2	2	4.6ms	23.6ms	secure
`double_encryption`	2	2	6.7ms	25.7ms	`din` deduced
`perf_ok_xor_bad`	2	2	6.5ms	32.5s	`din` deduced

Table 3.3: Verification Results for XOR Model

3.7 Related Work

This section reviews prior research and systems relevant to our study, providing the context for our approach and situating it within the broader literature.

Verification of Cryptographic Hardware

There is a rich literature in the verification of cryptographic hardware. Tools like Cryptol's allow users to specify cryptographic primitives in the Cryptol language, and then automatically synthesize an FPGA implementation [56], [57]. More recently, Cryptonite [58] allows users to synthesize hardware accelerators from straight-line C code.

In prior work, Dinesh et al. formulate the secure instruction set property (SISP) and develop ConjunCT [59] to verify whether hardware designs satisfy this property. The H-Houdini [60] algorithm and its associated tool, VeloCT, allow for push-button verification of large hardware designs. SynthCT [61] uses program synthesis to automatically translate unsafe instructions in a binary into safe instructions for a particular microarchitecture.

Secure language extensions

This project was also partly inspired by prior work concerning secure language extensions. In particular, our work borrows ideas from the extension of the P programming language [62] to its security-centric extension, Psec [63], [64]. This extension allows programmers to create distributed systems consisting of both trusted and untrusted machines.

3.8 Conclusion

In summary, in this chapter we presented BtorSec, a model-checking format for hardware security queries. In addition to the new format, we also presented a compiler and a proof-of-concept model checker that leverages the novel format to answer queries about the confidentiality of signals in a given circuit. Overall, the toolchain presented enables microarchitects to easily and seamlessly verify security properties about their designs with minimal engineering effort. We view this work as a starting point for integrating highly-automated security verification into the hardware design process.

3.8.1 Future Work

There are several directions in which the work presented in this chapter can be improved. We now give a few ideas on how future projects could build off the work done for BtorSec.

BtorSec

At the language level, we could include more sophisticated cryptographic primitives, such as pseudorandom number generators or more complex primitives like commitments for zero-knowledge proofs. With the rising popularity of hardware accelerators for cryptographic primitives for ZKPs, this could prove to be a valuable line of research.

SSV

At the solver level, there is much work left to do for SSV. Currently, our solver can only answer queries about confidentiality. Moreover, it can only perform bounded model checking, meaning that it can be useful to find bugs within a system, but is fundamentally unable to prove that a design is free of bugs altogether.

Integration

The ultimate vision for this report is to provide microarchitects and verification engineers with robust and easy-to-use tools for hardware verification. Following this vision, we could also explore integration of the BtorSec suite (i.e. the compiler and solver) with other existing toolkits. For instance, one could specify the high-level design of their system with UCLID5, verify that their BtorSec implementation indeed refines the high-level spec, and then verify that the primitives being abstracted do indeed meet all the security requirements with Cryptol and Cryptonite. Ideally, work in this direction would allow experts to seamlessly verify security properties of their designs at different levels of abstraction, at much lower engineering overheads.

Chapter 4

Conclusion

This project report has presented two approaches to the formal verification of hardware security. The goal of this work was to show how formal verification of security properties can be incorporated into the hardware design life cycle. We demonstrated that building formal models, such as the one presented in Chapter 2, can provide a high degree of confidence in a system's security. Furthermore, in Chapter 3, we showed that with the right tools, formal verification can be integrated seamlessly into hardware implementation workflows.

Looking ahead, there are several exciting directions future projects could take. One particularly promising direction is to unify various design and formal verification tools into a single, monolithic ecosystem. This integration would save both verification engineers and designers time, enabling them to specify and prove security properties at multiple levels of granularity in a cohesive way. Tightly coupling the processes of design and verification would also likely reduce design flaws and, as a result, lead to more secure hardware.

Appendix A

U^2 Types

We provide a description of the types used in the U^2 model. The types that make up our model are summarized in Table A.1.

Type	Description
A	Action; enum consisting of MALLOC, FREE, LOAD, STORE
$\mathbb{B}$	Boolean.
$\mathbb{B}_n$	n-bit bit-vector.
$\mathcal{E}$	Encryption state type.
$\mathcal{G}$	Ghost memory record.
I	Register index.
K	Uninterpreted key type.
$\mathcal{O}$	Observation record.
O	Uninterpreted opcode type.
W	Machine words.

Table A.1: Types of U^2

State var.	Description
`value`	Real value stored in memory
`h_base`	Base of the handle
`h_length`	Length of the handle
`h_offset`	Offset of the handle
`enc_state`	State of the encryption

Table A.2: Ghost Memory Record

State var.	**Description**
`valid`	Boolean value denoting whether this is a valid allocation.
`ghost_data`	Observable ghost data (of type $\mathcal{G}$).

Table A.3: Observation Record

Appendix B

UCLID5 Model of U^2

This appendix includes the source code of our UCLID5 model of the U^2 system.

B.1 Common Module

```


module common {

    type word_t = bv2;
    type size_t = word_t;

    type opcode_t;

    type lmapind_t = bv3;

    // Attacker, and Victim keys
    type key_t = enum { AtKey, ViKey };

    type nonce_t;

    type value_t = enum { RAW, HAN };

    datatype enc_state_t = Pln()
        | DEnc(DEnc_val: enc_state_t, DEnc_key: key_t, DEnc_nonce: nonce_t
    ↪ , intcheck: boolean)
        | DDec(DDec_val: enc_state_t, DDec_key: key_t, DDec_nonce: nonce_t
    ↪ )
        | PEnc(PEnc_val: enc_state_t, PEnc_key: key_t)
        | PDec(PDec_val: enc_state_t, PDec_key: key_t);

    type data_t = word_t;

    // Operation semantics should be independent of this
    type ghost_data_t = record {
        // Raw value
        value: word_t,

        // Value state (i.e. is this a handle or a raw value?)
        vtype: value_t,

        // Handle elements
        // Base address (inclusive)
        h_base: word_t,
        // This is the current handle offset
        h_offset: word_t,
        // Allocation is [h_base, h_base+h_length)
        h_length: word_t,

        // Only modified by the encryption function
        // Encryption state
        enc_state: enc_state_t,
        // True nonce for that allocation (used for authentication)
        ca_nonce_true: nonce_t
    };

    type alloc_data_t = boolean;


    type observable_t = record {
        valid : boolean,
        data_g : ghost_data_t
    };

    // Data map
    type mem_t = [word_t]data_t;
    // Local variable map
    type lmap_t = [lmapind_t]data_t;
    // Is this current cell allocated?
    type alloc_map_t = [word_t]alloc_data_t;

    // Ghost memory
    type ghost_mem_t = [word_t]ghost_data_t;
    // Ghost local variable map
    type ghost_lmap_t = [lmapind_t]ghost_data_t;

    type alloc_struct_t = record {
        valid   : boolean,
        allocg  : ghost_data_t,

        newgmem : ghost_mem_t,
        newamap : alloc_map_t
    };

    // Action performed by the process
    type action_t = enum {
        MALLOC, FREE, LOAD, STORE, HANOP, ARITHOP
    };


    // Constants
    const zero  : word_t = 0bv2;
    const one   : word_t = 1bv2;
    const wmax  : word_t = 3bv2;

    define between (a: word_t, b : word_t, c: word_t) : boolean = ((b <=_u
   ↪  a) && (a <_u c));

}
```

Listing B.1: Common Module

B.2 U^2 Module

```


module u2 {

    type * = common.*;

    const * = common.*;

    define * = common.*;

    // Ghost state
    var ghost_mem   : ghost_mem_t;
    var ghost_lmap  : ghost_lmap_t;
    // Allocated map
    var alloc_map: alloc_map_t;


    // Current operation was a memory safety vulnerability
    var msv_curr : boolean;
    // Architectural observable
    var observable : observable_t;

    // Debugging
    var addr_probe       : data_t;
    var addr_probe_g     : ghost_data_t;
    var addr_probe_true  : word_t;
    var data_probe       : data_t;
    var data_probe_g     : ghost_data_t;
    var data_probe_pre   : data_t;
    var data_probe_pre_g : ghost_data_t;

    var alloc_probe      : alloc_struct_t;
    var src1_probe       : data_t;
    var src1_probe_g     : ghost_data_t;
    var src2_probe       : data_t;
    var src2_probe_g     : ghost_data_t;


    // Temporary local variable indices
    var opcode : opcode_t;
    var l1, l2, l3: lmapind_t;
    var action: action_t;

    init {
        // Initially all cells are unallocated
        assume (forall (a: word_t) :: (!alloc_map[a] && !shadow_mem[a].
   ↪ alloc));
```

```

        // Initially all values (in lmap and memory) hold raw data (not
    ↪ pointers)
        assume (forall (a: lmapind_t) :: ghost_lmap[a].vtype == RAW);
        assume (forall (a: word_t) :: ghost_mem[a].vtype == RAW);

        observable.valid = false;
        msv_curr = false;
    }

    axiom slot_to_nonce_disjointness :
        (forall (i1: word_t, i2: word_t, i3: word_t, i4: word_t) :: (
            non_overlapping_ranges(i1, i2, i3, i4) ==> (slot_to_nonce(i1,
    ↪ i2) != slot_to_nonce(i3, i4))
        ))
    ;

    next {
        havoc l1;
        havoc l2;
        havoc l3;
        havoc opcode;
        havoc action;

        case
            (action == LOAD) : { call load(l1, l2, ViKey, ViKey); }
            (action == STORE) : { call store(l1, l2, ViKey, ViKey); }
            (action == MALLOC) : { call malloc(l1, ViKey, ViKey); }
            (action == HANOP) : { call hanop(opcode, l1, l2, l3); }
            (action == ARITHOP) : { call arithop(opcode, l1, l2, l3); }
        esac
    }

}
```

Listing B.2: U^2 Module

B.3 Main Module

```
module main
{
    type * = common.*;
    const * = common.*;

    define * = u2.*;
    function * = u2.*;

    instance u2_instance : u2();

    var v: boolean;

    // init
    init {
        v = false;
    }

    next {
        if (v) {
            next(u2_instance);
        }
        v' = true;
    }

    // This the global confidentiality property
    property load_msv_implies_ni : u2_instance.msv_curr ==> (
        !u2_instance.observable.valid
        || (u2_instance.observable.data_g.enc_state != Pln())
    );

    // This is the integrity property
    property store_msv_implies_integrityviolation : (
        forall (a: word_t) :: ((!u2_instance.alloc_map[a]) || (
            is_DEnc(u2_instance.ghost_mem[a].enc_state) &&
            (u2_instance.ghost_mem[a].ca_nonce_true
                == u2_instance.ghost_mem[a].enc_state.DEnc_nonce)
        ) || (
            is_garbled(u2_instance.ghost_mem[a].enc_state)
        ))
    );

    property shadow_state_alloc_matches_alloc : (
        forall (a: word_t) :: (
            u2_instance.alloc_map[a] == u2_instance.shadow_mem[a].alloc
        )
    );

    property shadow_state_interval_matches_ca_nonce_true : (
        forall (a: word_t) :: ((!u2_instance.shadow_mem[a].alloc) || (
            slot_to_nonce(u2_instance.shadow_mem[a].lowend, u2_instance.shadow_mem[a].highend)
                == u2_instance.ghost_mem[a].ca_nonce_true
        ))
    );

    property shadow_state_interval_is_memory_safe : (
        forall (a: word_t) :: ((!u2_instance.shadow_mem[a].alloc) || (
            between (a, u2_instance.shadow_mem[a].lowend, u2_instance.shadow_mem[a].highend)
        ))
    );

    property handle_sizes_are_nonzero : (
        (forall (a: word_t) :: (handles_are_valid(u2_instance.ghost_mem[a])))
        && (forall (l: lmapind_t) :: (handles_are_valid(u2_instance.ghost_lmap[l])))
    );

    property mem_handles_are_summarized_in_shadow_state : (
        forall (a: word_t) :: (
            (u2_instance.alloc_map[a] && u2_instance.ghost_mem[a].vtype == HAN)
                ==> ((u2_instance.shadow_mem[u2_instance.ghost_mem[a].h_base].lowend
                        == u2_instance.ghost_mem[a].h_base)
                    && (u2_instance.shadow_mem[u2_instance.ghost_mem[a].h_base].highend
                        == (u2_instance.ghost_mem[a].h_base + u2_instance.ghost_mem[a].h_length))
                    && u2_instance.shadow_mem[u2_instance.ghost_mem[a].h_base].alloc
                )
        )
    );

    property lmap_handles_are_summarized_in_shadow_state : (
        forall (l: lmapind_t) :: (
            (u2_instance.ghost_lmap[l].vtype == HAN)
                ==> ((u2_instance.shadow_mem[u2_instance.ghost_lmap[l].h_base].lowend
                        == u2_instance.ghost_lmap[l].h_base)
                    && (u2_instance.shadow_mem[u2_instance.ghost_lmap[l].h_base].highend
                        == (u2_instance.ghost_lmap[l].h_base + u2_instance.ghost_lmap[l].h_length))
                    && u2_instance.shadow_mem[u2_instance.ghost_lmap[l].h_base].alloc
                )
        )
    );

    property shadow_state_non_overlapping_intervals : (
        forall (a1: word_t) :: (
            forall (a2: word_t) :: ((
                    u2_instance.shadow_mem[a1].alloc && u2_instance.
↪ shadow_mem[a2].alloc
                ) ==> (
                    non_overlapping_ranges (
                        u2_instance.shadow_mem[a1].lowend,
                        u2_instance.shadow_mem[a1].highend,
                        u2_instance.shadow_mem[a2].lowend,
                        u2_instance.shadow_mem[a2].highend
                    ) || (
                        u2_instance.shadow_mem[a1].lowend == u2_instance.
↪ shadow_mem[a2].lowend
                        && u2_instance.shadow_mem[a1].highend ==
↪ u2_instance.shadow_mem[a2].highend
                    )
                )
            )
        )
    );

    property shadow_state_consistent_intervals : (
        forall (a1: word_t) :: (
            forall (a2: word_t) :: ((
                    u2_instance.shadow_mem[a1].alloc &&
                    between (a2, u2_instance.shadow_mem[a1].lowend,
↪ u2_instance.shadow_mem[a1].highend)
                ) ==> (
                    u2_instance.shadow_mem[a1].lowend == u2_instance.
↪ shadow_mem[a2].lowend
                    && u2_instance.shadow_mem[a1].highend == u2_instance.
↪ shadow_mem[a2].highend
                    && u2_instance.shadow_mem[a2].alloc
                )
            )
        )
    );

    // For valid handles, raw value is the same as handle value
    property mem_value_matches_handles : (
        forall (a: word_t) :: (
            (u2_instance.ghost_mem[a].vtype == HAN) ==>
            (
                u2_instance.ghost_mem[a].value == (
                    u2_instance.ghost_mem[a].h_base + u2_instance.
    ↪ ghost_mem[a].h_offset
                )
            )
        )
    );

    // For valid handles, raw value is the same as handle value
    property lmap_value_matches_handles : (
        forall (a: lmapind_t) :: (
            (u2_instance.ghost_lmap[a].vtype == HAN) ==>
            (
                u2_instance.ghost_lmap[a].value == (
                    u2_instance.ghost_lmap[a].h_base + u2_instance.
    ↪ ghost_lmap[a].h_offset
                )
            )
        )
    );


    // spec
    control {
        v = induction;
        check;
        print_results;
        v.print_cex_json();
    }
}
```

Listing B.3: Main Module

B.4 Operations Module

```


module u2 {

    function hanop_fun (oldoff: word_t, operand: word_t) : word_t;


    procedure [noinline] malloc_helper (s: size_t, pkey: key_t, dkey:
   ↪ key_t)
        returns (as: alloc_struct_t)
        ensures ((
                s == zero ==> (!as.valid)
            ) && ((
                // was able to perform the allocation
                as.valid
                // returned value is a handle type (raw value does not
   ↪ matter)
                && (as.allocg.vtype == HAN)
                // allocation has size s
                && (as.allocg.h_length == s)
                // allocation is in a valid interval
                && (no_addr_overflow(as.allocg.h_base, s))
                // current offset is zero
                && (as.allocg.h_offset == zero)
                // generated pointer is p_encrypted
                && (as.allocg.enc_state == PEnc(Pln(), pkey))
                // Underlying data value matches the handle
                && (as.allocg.value == as.allocg.h_base)
                // so ca nonce does not matter (since no data encryption)
                && true
                // allocation interval was free originally
                && (forall (a: word_t) :: (
                    (between(a, as.allocg.h_base, (as.allocg.h_base + s)))
                    ==> !alloc_map[a]
                ))
                // allocation interval is now claimed
                && (forall (a: word_t) :: (
                    (between(a, as.allocg.h_base, (as.allocg.h_base + s)))
                    ==> as.newamap[a]
                ))
                // mem outside allocation interval is the same
                // and ghost_mem
                && (forall (a: word_t) :: (
                    (!(between (a, as.allocg.h_base, (as.allocg.h_base + s
   ↪ ))))
                    ==> as.newgmem[a] == ghost_mem[a]
                ))
                // mem in allocation interval is protected
                //  and is not a pointer type
                && (forall (a: word_t) :: (
                    (between(a, as.allocg.h_base, (as.allocg.h_base + s)))
                    ==> (as.newgmem[a].enc_state == DEnc(
                            Pln(),
                            dkey,
                            slot_to_nonce(as.allocg.h_base, as.allocg.
   ↪ h_base + s),
                            true)
                        && as.newgmem[a].ca_nonce_true
                            == slot_to_nonce(as.allocg.h_base, as.allocg.
   ↪ h_base + s)
                        && as.newgmem[a].vtype == RAW
                    )
                ))

                // and allocation map
                && (forall (a: word_t) :: (
                    (!(between (a, as.allocg.h_base, (as.allocg.h_base + s
   ↪ ))))
                    ==> as.newamap[a] == alloc_map[a]
                )))
                // was not able to perform allocation
                || !as.valid
            )
        );
    { }


    procedure malloc (lind: lmapind_t, pkey: key_t, dkey: key_t)
        modifies ghost_mem, ghost_lmap;
        // Shadow state
        modifies shadow_mem;
        modifies alloc_map;
        // Debugging
        modifies data_probe, data_probe_g, alloc_probe;
    {
        var as : alloc_struct_t;

        var nondet_size: word_t;

        // Call the helper function
        call (as) = malloc_helper(nondet_size, pkey, dkey);

        // If the allocation was successful, update the local variable map
        if (as.valid) {
            // Update the shadow state
            call shadow_malloc(as);

            ghost_lmap[lind] = as.allocg;
```

```
         ghost_mem = as.newgmem;
         alloc_map = as.newamap;
     }

     data_probe_g = ghost_lmap[lind];
     alloc_probe = as;

 }


 procedure load (lind_dest: lmapind_t, lind_addr: lmapind_t, pkey:
↪ key_t, dkey: key_t)
     modifies msv_curr, observable;
     modifies ghost_lmap;
     // Debugging
     modifies addr_probe, data_probe, data_probe_pre;
     modifies addr_probe_g, data_probe_g, data_probe_pre_g,
↪ addr_probe_true;
 {
     var addr_bundle     : data_t;
     var addr_bundle_g   : ghost_data_t;
     var h_base          : word_t;
     var h_offset        : word_t;
     var h_length        : word_t;
     var addr            : word_t;
     var cell_data       : data_t;
     var cell_data_g     : ghost_data_t;
     var cell_alloc      : alloc_data_t;
     var dec_cell_data   : data_t;
     var dec_cell_data_g : ghost_data_t;

     // Get the linear address bundle and value
     call (addr_bundle_g) = ghost_ca_decrypt(ghost_lmap[lind_addr],
↪ pkey);


     // extract fields
     h_base = addr_bundle_g.h_base;
     h_offset = addr_bundle_g.h_offset;
     h_length = addr_bundle_g.h_length;
     // compute the address
     // INFO: ghost_to_nonce is the more precise
     addr = ghost_to_addr_value(addr_bundle_g);

     // address value does not wrap around heap boundary
     assume(no_addr_overflow(h_base, h_offset));

     // definition of a spatial memory safety vulnerability
     if (!between(h_offset, zero, h_length) || addr_bundle_g.vtype !=
↪ HAN) {
```

```
            msv_curr = true;
        } else {
            msv_curr = false;
        }

        // get memory cell at the address
        cell_data_g = ghost_mem[addr];
        cell_alloc = alloc_map[addr];

        if (!cell_alloc) {
            // in this case, read is invalid, and there is a fault
            observable.valid = false;
        } else {

            // INFO: ghost_to_nonce below is the accurate model
            assume(value_matches_handle_iff_valid(addr, addr_bundle_g));

            call (dec_cell_data_g) = ghost_data_decrypt(cell_data_g, dkey,
                ghost_to_nonce(addr_bundle_g)
            );


            // observable is the decrypted value of the cell
            observable.valid = true;
            observable.data_g = dec_cell_data_g;

            // decrypted data is placed as in in the local variable map
            ghost_lmap[lind_dest] = dec_cell_data_g;
            // Debugging probe
            data_probe_pre = cell_data;
            data_probe_pre_g = cell_data_g;
            data_probe = dec_cell_data;
            data_probe_g = dec_cell_data_g;
        }
        addr_probe = addr_bundle;
        addr_probe_g = addr_bundle_g;
        addr_probe_true = addr;
    }


    procedure store (lind_addr: lmapind_t, lind_val: lmapind_t, pkey:
    ↪ key_t, dkey: key_t)
        modifies ghost_mem;
        modifies msv_curr, observable;
        // Debugging
        modifies addr_probe, addr_probe_g, data_probe, data_probe_g;
    {
        var addr_bundle      : data_t;
        var addr_bundle_g    : ghost_data_t;
        var h_base           : word_t;
        var h_offset         : word_t;
        var h_length         : word_t;
        var addr             : word_t;
        var oldcell_data     : data_t;
        var oldcell_data_g   : ghost_data_t;
        var oldcell_alloc    : alloc_data_t;
        var cell_data        : data_t;
        var cell_data_g      : ghost_data_t;
        var cell_alloc       : alloc_data_t;
        var enc_cell_data    : data_t;
        var enc_cell_data_g  : ghost_data_t;

        // Get the linear address bundle and value
        call (addr_bundle_g) = ghost_ca_decrypt(ghost_lmap[lind_addr],
    ↪ pkey);


        // extract fields
        h_base = addr_bundle_g.h_base;
        h_offset = addr_bundle_g.h_offset;
        h_length = addr_bundle_g.h_length;
        // compute the address
        // INFO: ghost_to_addr_value is the precise model
        addr = ghost_to_addr_value(addr_bundle_g);

        // no address overflow
        assume(no_addr_overflow(h_base, h_offset));
        // definition of a memory safety vulnerability
        if (!between(h_offset, zero, h_length)) {
            msv_curr = true;
        } else {
            msv_curr = false;
        }

        // Old cell location
        oldcell_data_g = ghost_mem[addr];
        oldcell_alloc = alloc_map[addr];

        // Plaintext value of the store
        cell_data_g = ghost_lmap[lind_val];
        // Encrypted value of the store

        // INFO: ghost_to_nonce below is the accurate model
        call (enc_cell_data_g) = ghost_data_encrypt(cell_data_g, dkey,
            ghost_to_nonce(addr_bundle_g),
            oldcell_data_g
        );


        observable.valid = false;

        if (oldcell_alloc) {
            // in this case, write is invalid, and there is a fault
        } else {
            // Decrypted data is placed as in in the local variable map
            ghost_mem[addr] = enc_cell_data_g;
            // Debugging probe
            data_probe = enc_cell_data;
            data_probe_g = enc_cell_data_g;
        }
        addr_probe = addr_bundle;
        addr_probe_g = addr_bundle_g;
    }


    procedure hanop (op: opcode_t, lind_dest: lmapind_t, lind_src1:
   ↪ lmapind_t, lind_src2: lmapind_t)
        modifies ghost_lmap;
        // Debugging
        modifies data_probe, src1_probe, src2_probe;
        modifies data_probe_g, src1_probe_g, src2_probe_g;
    {
        var newoffset: word_t;
        var newdata: ghost_data_t;

        // Abstraction for the hanop
        var nondet_arg : word_t;


        // First operand must be a handle
        src1_probe_g = ghost_lmap[lind_src1];

        if (ghost_lmap[lind_src1].vtype == HAN) {
            // Both the value and the handle offset are modulated by the
   ↪ same amount
            newoffset = ghost_lmap[lind_src1].h_offset + nondet_arg;
            assume (no_addr_overflow(ghost_lmap[lind_src1].h_base,
   ↪ newoffset));

            newdata = ghost_lmap[lind_src1];
            // With new offset defined by hanop_fun
            newdata.h_offset = newoffset;

            // Also shift the raw value by nondet_arg
            newdata.value = newdata.value + nondet_arg;
            ghost_lmap[lind_dest] = newdata;

        } else {
            // Unhandle
            assume (newdata.vtype == RAW);

            // INFO: completely unconstrained
            ghost_lmap[lind_dest] = newdata;
        }
        // Debugging probes
        data_probe_g = newdata;
    }


    procedure arithop (op: opcode_t, lind_dest: lmapind_t, lind_src1:
    ↪ lmapind_t, lind_src2: lmapind_t)
        modifies ghost_lmap;
        // Debugging
        modifies data_probe, data_probe_g;
    {
        var newdata : data_t;
        var newdata_g : ghost_data_t;

        // Result from expression evaluation is a non-handle
        // The result can be arbitrary
        assume (newdata_g.vtype == RAW);

        ghost_lmap[lind_dest] = newdata_g;
        // Debugging probe
        data_probe = newdata;
        data_probe_g = newdata_g;
    }
}
```

Listing B.4: Operations Module

B.5 Cryptography Module

```


module u2 {

    // Encryption and decryption functions: these are left uninterpreted
    function cdecrypt (addr_in: word_t, key: key_t) : word_t;
    function cencrypt (addr_in: word_t, key: key_t) : word_t;
    // Data encryption and decryption
    function dencrypt (value_in: word_t, key: key_t, ca: nonce_t) : word_t
    ↪ ;
    function ddecrypt (value_in: word_t, key: key_t, ca: nonce_t) : word_t
    ↪ ;

    // For a given range on the slot, what is the CA
    // that maps to this slot
    // This is the power-of-2 assumption from the C3 paper
    function slot_to_nonce (s1: word_t, s2: word_t) : nonce_t;

    function ghost_to_nonce (val: ghost_data_t) : nonce_t;

    function ghost_to_addr_value (ghost_data: ghost_data_t) : data_t;


    // Given an address what is the corresponding nonce?
    function ca_to_nonce (addr: word_t) : nonce_t;

    define is_garbled (enc_state : enc_state_t) : boolean = (
        (is_DEnc(enc_state) && !is_Pln(enc_state.DEnc_val))
        || is_DDec(enc_state)
    );

    // Operation over the RAW value of the data
    procedure ca_decrypt (addr_in: data_t, key: key_t)
        returns (addr_out: data_t)
    {
        addr_out = cdecrypt(addr_in, key);
    }

    // All cryptographic operations preserve the non-crypto fields
    // and only operate on the crypto ADT elements
    procedure ghost_ca_decrypt (ghost_addr_in: ghost_data_t, key: key_t)
        returns (ghost_addr_out: ghost_data_t)
    {
        // Core operation
        // Preserves the internal values
        ghost_addr_out.value = ghost_addr_in.value;
        ghost_addr_out.vtype = ghost_addr_in.vtype;
        ghost_addr_out.h_base = ghost_addr_in.h_base;
        ghost_addr_out.h_offset = ghost_addr_in.h_offset;
        ghost_addr_out.h_length = ghost_addr_in.h_length;

        // Operation over the ADT-based abstraction
        if (is_Pln(ghost_addr_in.enc_state) || is_PDec(ghost_addr_in.
    ↪ enc_state)
            || is_DEnc(ghost_addr_in.enc_state) || is_DDec(ghost_addr_in.
    ↪ enc_state)) {
            ghost_addr_out.enc_state = PDec(ghost_addr_in.enc_state, key);
        } else {
            if (is_PEnc(ghost_addr_in.enc_state)
                && ghost_addr_in.enc_state.PEnc_key == key) {
                ghost_addr_out.enc_state = ghost_addr_in.enc_state.
    ↪ PEnc_val;
            } else {
                // Unreachable
                // assert(false);
            }
        }
        ghost_addr_out.ca_nonce_true = ghost_addr_in.ca_nonce_true;
    }


    procedure ca_encrypt (addr_in: data_t, key: key_t)
        returns (addr_out: data_t)
    {
        addr_out = cencrypt(addr_in, key);
    }

    procedure ghost_ca_encrypt (ghost_addr_in: ghost_data_t, key: key_t)
        returns (ghost_addr_out: ghost_data_t)
    {
        // Preserves the internal values
        ghost_addr_out.value = ghost_addr_in.value;
        ghost_addr_out.vtype = ghost_addr_in.vtype;
        ghost_addr_out.h_base = ghost_addr_in.h_base;
        ghost_addr_out.h_offset = ghost_addr_in.h_offset;
        ghost_addr_out.h_length = ghost_addr_in.h_length;

        ghost_addr_out.enc_state = PEnc(ghost_addr_in.enc_state, key);
        ghost_addr_out.ca_nonce_true = ghost_addr_in.ca_nonce_true;
    }


    procedure data_encrypt (data_in: data_t, key: key_t, ca: nonce_t)
        returns (data_out: data_t)
    {
        data_out = dencrypt(data_in, key, ca);
    }

    procedure ghost_data_encrypt (ghost_data_in: ghost_data_t, key: key_t, ca: nonce_t, old_ghost_data: ghost_data_t)
        returns (ghost_data_out: ghost_data_t)
    {
        // Core operation (kept uninterpreted)
        ghost_data_out.value = ghost_data_in.value;
        ghost_data_out.vtype     = ghost_data_in.vtype;
        ghost_data_out.h_base     = ghost_data_in.h_base;
        ghost_data_out.h_offset   = ghost_data_in.h_offset;
        ghost_data_out.h_length   = ghost_data_in.h_length;

        // Operation over the abstraction
        // Integrity violation
        if (old_ghost_data.ca_nonce_true != ca) {
            ghost_data_out.enc_state = DEnc(ghost_data_in.enc_state, key, ca, false);
        } else {
            ghost_data_out.enc_state = DEnc(ghost_data_in.enc_state, key, ca, true);
        }
        ghost_data_out.ca_nonce_true = old_ghost_data.ca_nonce_true;
    }

    procedure data_decrypt (data_in: data_t, key: key_t, ca: nonce_t)
        returns (data_out: data_t)
    {
        data_out = ddecrypt(data_in, key, ca);
    }

    procedure ghost_data_decrypt (ghost_data_in: ghost_data_t, key: key_t, ca: nonce_t)
        returns (ghost_data_out: ghost_data_t)
    {
        // Preserves the internal values
        ghost_data_out.value      = ghost_data_in.value;
        ghost_data_out.vtype     = ghost_data_in.vtype;
        ghost_data_out.h_base     = ghost_data_in.h_base;
        ghost_data_out.h_offset   = ghost_data_in.h_offset;
        ghost_data_out.h_length   = ghost_data_in.h_length;


        if (is_DEnc(ghost_data_in.enc_state)
            && ghost_data_in.enc_state.intcheck
            && ghost_data_in.enc_state.DEnc_key == key
            && ghost_data_in.enc_state.DEnc_nonce == ca) {
            ghost_data_out.enc_state = ghost_data_in.enc_state.DEnc_val;
        } else {
            ghost_data_out.enc_state = DDec(ghost_data_in.enc_state, key, ca);
        }

        ghost_data_out.ca_nonce_true  = ghost_data_in.ca_nonce_true;
    }

}
```

Listing B.5: Operations Module

B.6 Contracts Module

```


module u2 {


    define addr_must_be_handle (ghost_addr_bundle : ghost_data_t) : boolean =
        (ghost_addr_bundle.vtype == HAN);

    define no_addr_overflow (h_base : word_t, h_offset : word_t) : boolean =
        (h_base <=_u wmax - h_offset);

    define non_overlapping_ranges (b1: word_t, l1: word_t, b2: word_t, l2: word_t) : boolean =
        // Both of these work and result in convergence with upto bv4
        ((b1 <_u l1) && (b2 <_u l2) && !between (b1, b2, l2) && !between (b2, b1, l1));
        // (b1 <_u l1) && (b2 <_u l2) && !((b1 <=_u (l2-one)) && (b2 <=_u (l1-one)));

    define handles_are_valid (ghost_addr_bundle: ghost_data_t) : boolean =
        (ghost_addr_bundle.vtype == HAN) ==> (zero <_u ghost_addr_bundle.h_length);

    define value_matches_handle_iff_valid (addr: word_t, ghost_addr_bundle : ghost_data_t) : boolean =
        // Non-plain or non-handles do not match any-slot ("diffusion property")
        ((ghost_addr_bundle.vtype != HAN || !is_Pln(ghost_addr_bundle.enc_state)) ==> (
            forall (a1: word_t) :: (
                forall (a2 : word_t) :: (
                    (a1 <_u a2) ==> (ghost_to_nonce(ghost_addr_bundle) != slot_to_nonce(a1, a2))
                )
            )
        )) && ((ghost_addr_bundle.vtype == HAN && is_Pln(ghost_addr_bundle.enc_state)) ==> (
            ghost_to_nonce(ghost_addr_bundle) ==
                slot_to_nonce(ghost_addr_bundle.h_base, ghost_addr_bundle.h_base+ghost_addr_bundle.h_length)
            && ghost_addr_bundle.value == ghost_to_addr_value(ghost_addr_bundle)
        ))
    ;


}
```

Listing B.6: Operations Module

B.7 Shadow U^2 Module

```


module u2 {

    // Ghost memory with slots information
    var shadow_mem: shadow_mem_t;

}
```

Listing B.7: Shadow U^2 Module

B.8 Shadow Memory Module

```


module u2 {


    procedure [noinline] shadow_malloc_helper (as: alloc_struct_t)
        returns (nm: shadow_mem_t)
        ensures (
            // New memory is allocated in the ghost state
            //  with the correct allocation interval information
            (forall (a: word_t) :: (
                (between (a, as.allocg.h_base, (as.allocg.h_base + as.allocg.h_length)))
                ==> (
                    nm[a].alloc
                    && nm[a].lowend == as.allocg.h_base
                    && nm[a].highend == as.allocg.h_base + as.allocg.h_length
                    // True nonce (currently needs to be non-shadow)
                    // && nm[a].nonce_true ==
                    //     slot_to_nonce(as.alloca.h_base, as.alloca.h_base + as.alloca.h_length)
                )
            ))
            // Originally shadow_mem is unoccupied
            && (forall (a: word_t) :: (
                (between (a, as.allocg.h_base, (as.allocg.h_base + as.allocg.h_length)))
                ==> (!shadow_mem[a].alloc)
            ))
            // And memory outside this interval stays the same
            && (forall (a: word_t) :: (
                (!(between (a, as.allocg.h_base, (as.allocg.h_base + as.allocg.h_length))))
                ==> nm[a] == shadow_mem[a]
            ))
        );
    { }

    procedure shadow_malloc (as: alloc_struct_t)
        modifies shadow_mem;
    {
        // Call the helper function
        call (shadow_mem) = shadow_malloc_helper(as);
    }

}
```

Listing B.8: Shadow Operations Module

Appendix C

Formal Foundations of SSV

C.1 Introduction

At a high level, the purpose of SSV is to verify that a system uses cryptographic primitives effectively to protect confidential data against an adversary observing its execution. From this informal description of our goal, many questions naturally arise: What does it mean to protect confidential data? How exactly does the adversary observe the system while it executes? Most importantly, how do we go about verifying that this security property holds?

These are all important questions that must be answered precisely if we wish to make SSV work. Hence, the focus of this chapter is to build the formal framework required to set up and solve the security verification problem. We begin by presenting a formal model of the system, specifying both its syntax and semantics. After that, we give a formal description of our threat model and the security problem that we wish to verify. Next, we present the verification problem, and show how we can Finally, we conclude the chapter with an overview of ProtoSL, our Python implementation of this formal framework.

C.2 System Model

We now introduce our system model. Intuitively, we consider a system in which imperative programs are executed over a set of typed variables, which take on familiar values such as booleans, integers, bit-vectors, etc, in addition to an abstract "message type". A key aspect of our system model is the threat model, which consists of a passive but curious adversary; the adversary comes equipped with the ability to *observe* a subset of the variables in the program and perform some computation on the values that those variables take on. They are not, however, allowed to interfere with or otherwise modify the program. We now proceed to formalize this intuitive notion.

C.2.1 System State

Abstract and Base Domains

Given that the typed variables are instrumental to our system model, we begin by introducing what values they can take on. We consider two domains: The abstract message domain and the base domain. The first is the abstract domain, $\mathbb{A}$, which can be conceptualized as the domain of all messages. Let $\mathbb{M}$ denote the concrete domain of message types. Then we define the abstract domain $\mathbb{A}$ by the following grammar:

$$\mathbb{A} = \mathbb{M} \mid \text{SymEnc}(d:\mathbb{A}, k:\mathbb{A}) \mid \text{SymDec}(d:\mathbb{A}, k:\mathbb{A})$$

On the other hand, the base domain, $\mathbb{B}$, is the union of integers, Booleans, bit-vectors, etc. Morally, we think of $\mathbb{A}$ as the message domain (augmented to support cryptographic operations) and $\mathbb{B}$ as the non-message domain. The system will consist of a set of variables $\mathcal{V}$, which includes message-typed variables $\mathcal{V}_a$ that take values from the domain $\mathbb{A}$, and non-message-typed variables $\mathcal{V}_b$ that take values from the domain $\mathbb{B}$. Naturally, $\mathcal{V} = \mathcal{V}_a \cup \mathcal{V}_b$.

States

Let $\mathbb{U} = \mathbb{A} \cup \mathbb{B}$ denote our universe. Then an abstract state σ is given by a $\mathbb{U}$-valued assignment of the variables in $\mathcal{V}$; that is, $\sigma : \mathcal{V} \to \mathbb{U}$. We assume that all states that are well-typed. To be more precise, any state σ can only assign a variable $v_a \in \mathcal{V}_a$ a value from $\mathbb{A}$; similarly, variables $v_b \in \mathcal{V}_b$ only get assigned values from $\mathbb{B}$. We denote the set of all states as Σ.

On occasion, we will need to refer to only the $\mathbb{A}$-valued or $\mathbb{B}$-valued assignments of a state σ. In these cases, we will denote the restriction of σ to $\mathbb{A}$-valued assignments as σ_a and the restriction of σ to $\mathbb{B}$-valued assignments as σ_b. Similarly, we will let Σ_a denote the set of all $\mathbb{A}$-valued restrictions and Σ_b denote the set of all $\mathbb{B}$-valued restrictions.

C.2.2 Programs

Intuitively, our system admits loop-free programs that allow for conditional branching. This subsection is dedicated to showing how these programs are constructed, and how their execution changes the system state. In other words, we now proceed to define a program's syntax and semantics.

Syntax

We now lay out the syntax of ProtoSL programs. Programs are executed over the previously defined variables and are very simple in nature. Our programs consist of a sequence of parallel assignments of variables to expressions, allowing branching based on Boolean conditions.

We allow a fairly permissive space of $\mathbb{U}$-valued expressions defined over a wide range of operations, the details of which we leave up to the implementation of the system. Much like we have done before, we define $\mathcal{E}$ as the set of all $\mathbb{U}$-valued expressions. The only assumption that ProtoSL makes is that any expression $e \in \mathcal{E}$ is well-typed, meaning that the value that e ultimately evaluates to matches the type of the variable that e is being assigned to.

We now formulate the statements that can be executed in the form of a grammar. Let $x \in \mathcal{V}$ be a variable and $\overline{x} \in \mathcal{V}^n$ be a collection of n variables: $\overline{x} = (x_1, x_2, ..., x_n)$. Additionally, let ψ be a Boolean constraint defined over the base domain; formally, $\psi : \sigma_b \to \{\texttt{true}, \texttt{false}\}$. Lastly, let $e \in \mathcal{E}$ be a single expression and $\overline{e} \in \mathcal{E}^n$ be a collection of n expressions: $\overline{e} = (e_1, e_2, ..., e_n)$.

$$\begin{array}{rlr}
\langle \mathsf{st} \rangle ::= & x \leftarrow e & \text{[assign]} \\
& \mid \overline{x} \leftarrow \overline{e} & \text{[multiassign]} \\
& \mid \text{if } \psi \text{ then } \langle \mathsf{st} \rangle \text{ else } \langle \mathsf{st} \rangle & \text{[ite]} \\
& \mid \langle \mathsf{st} \rangle \cdot \langle \mathsf{st} \rangle & \text{[composition]}
\end{array}$$

Semantics

In this section, we provide the semantics of ProtoSL programs, beginning by explaining how we interpret expressions. For every expression e, we define $[\![e]\!] : \Sigma \to \mathbb{A}$ as the interpretation of the expression. As a quick example, if our expression e is $x + y$ and we have that $\sigma(x) = 1$ and $\sigma(y) = 2$, then

$$[\![e]\!](\sigma) = [\![x + y]\!](\sigma) = 1 + 2 = 3,$$

as one would naturally expect.

We extend the above definition to vectors of expressions $\overline{e}$, where $[\![\overline{e}]\!] : \Sigma \to \mathbb{A}^n$. Continuing the above example, suppose $\overline{e} = (x + x, y + y)$. Then

$$[\![\overline{e}]\!](\sigma) = [\![(x + x, y + y)]\!](\sigma) = (1 + 1, 2 + 2) = (2, 4).$$

Next, we provide the semantics of our grammar in the form of an interpretation function $[\![\mathsf{st}]\!] : \Sigma \to \Sigma$ that returns a new state σ' after executing the statement st on the current state σ. This effectively defines how each ProtoSL statement is to be executed.

$$[\![x \leftarrow e]\!](\sigma) = \sigma[x \mapsto [\![e]\!](\sigma)]$$

$$[\![\overline{x} \leftarrow \overline{e}]\!](\sigma) = \sigma[\overline{x} \mapsto [\![\overline{e}(\sigma)]\!]]$$

$$[\![\text{if } \psi \text{ then } \mathsf{st}_A \text{ else } \mathsf{st}_B]\!](\sigma) = \begin{cases} [\![\mathsf{st}_A]\!](\sigma) & \text{if } \psi(\sigma) \\ [\![\mathsf{st}_B]\!](\sigma) & \text{otherwise} \end{cases}$$

$$[\![\mathsf{st}_A \cdot \mathsf{st}_B]\!] = [\![\mathsf{st}_B]\!]([\![\mathsf{st}_A]\!](\sigma))$$

Now that we have defined how statements are executed, we formalize a program P as a sequence of statements, $\mathsf{st}_1 \cdots \mathsf{st}_n$. If $\mathcal{P}$ has n statements, we say that it is a program of length n. We also allow indexing into the program: $\mathcal{P}_i = \mathsf{st}_i$. As an additional remark, if $\mathcal{P}$ has no statements, then we say that the length of $\mathcal{P}$ is 0, or that $\mathcal{P}$ is *empty*. We denote this special case as $\mathcal{P} = \varnothing$, and we define $[\![\varnothing]\!](\sigma_0) = \langle \sigma_0 \rangle$ for every $\sigma_0 \in \Sigma_0$.

Starting from an initial state $\sigma_0 \in \Sigma$, we define an execution of the program P as a sequence of states $[\![P]\!](\sigma_0) = \langle \sigma_0, \sigma_1, ..., \sigma_n \rangle$ where $\sigma_{i+1} = [\![\mathsf{st}_{i+1}]\!](\sigma_i)$. Often, we will refer to such a sequence of states as a *trace*, and represent it with the symbol τ. For convenience, we will allow indexing into a trace: if $\tau = \langle \sigma_0, \sigma_1, ..., \sigma_n \rangle$, then $\tau_i = \sigma_i$.

C.3 Threat Model

As we stated at the beginning of this appendix, we consider a threat model where the adversary is allowed to observe certain variables of the system. In addition, the adversary is allowed to perform computation on the values they observed, as dictated by a set of *rules*. We can thus conceptualize "security" as the adversary being unable to *deduce* any secret values, given their observations. In this section, we make the effort to formally specify this threat model, including the cryptographic assumptions that under which our threat model operates.

C.3.1 Abstract Terms and Rules

The terms in the abstract domain are given by the following grammar over a set of variables X:

$$T = \mathsf{Enc}(T, T) \mid \mathsf{Dec}(T, T) \mid x \in X$$

We refer to the variables in mentioned in a term as $\mathsf{vars}(T)$.

We define a rule r to be an object containing a head and a body: $\mathsf{head}(\mathsf{r})$ and $\mathsf{body}(\mathsf{r})$, respectively. The head is a term, and the body is a set of terms. For example, consider the rule r below detailing how one may derive a plaintext message (m) given the message's encryption ($\mathsf{Enc}(m, k)$) and the key used to encrypt it (k):

$$\mathsf{r} := m \leftarrow \{\mathsf{Enc}(m, k), k\}$$

Here, $\mathsf{head}(\mathsf{r}) = x$ and $\mathsf{body}(\mathsf{r}) = \{\mathsf{Enc}(x, k), k\}$. Just as we did with terms, we let $\mathsf{vars}(\mathsf{r})$ denote the set of variables mentioned in r.

Models of Cryptography

For our purposes, we will consider two models of cryptography, each defined as a set of rules: the perfect cryptography model and the XOR model. We define the set of rules for the perfect cryptography model, $\mathcal{R}_p$, as:

$$\begin{aligned}\mathcal{R}_p = \{&\mathsf{Enc}(m,k) \leftarrow \{m,k\},\\ &\mathsf{Dec}(c,k) \leftarrow \{c,k\},\\ &m \leftarrow \{\mathsf{Dec}(\mathsf{Enc}(m,k),k)\}\}.\end{aligned}$$

The first two rules give the adversary the ability to encrypt and decrypt arbitrary messages (terms) under some key (also a term). The third rule specifies how Enc and Dec interact. Namely, we can decrypt an encryption of a message m to recover it, provided we have the key k.

Next, we have the XOR model, which introduces a more sophisticated algebraic reasoning system. This is useful in analyzing a number of protocols that make use of certain symmetric key operations (e.g., one-time pad, stream ciphers). For this model, we first define a new constant symbol 0, representing the identity element. Then, we define $\mathcal{R}_x$ as:

$$\begin{aligned}\mathcal{R}_x = \{&a \oplus b \ \leftarrow \{a,b\},\\ &a \leftarrow \{a \oplus 0\},\\ &0 \leftarrow \{a \oplus a\},\\ &b \oplus a \leftarrow \{a \oplus b\},\\ &a \oplus (b \oplus c) \leftarrow \{(a \oplus b) \oplus c\},\\ &(a \oplus b) \oplus c \leftarrow \{a \oplus (b \oplus c)\}\}.\end{aligned}$$

These rules should align with the reader's existing understanding of the XOR ($\oplus$) operation.

Grounding

Given a rule r and an assignment of variables $\alpha : \mathsf{vars}(\mathsf{r}) \to \mathbb{A}$, we ground a rule by replacing the variables in the head and body with their respective assignment from α. Such a grounding produces a rule where the head is a message type and the body is also a message type. We denote the grounding of r with assignment α as $\mathsf{r}[\alpha]$. Likewise, we let $\mathsf{head}(\mathsf{r}[\alpha]) \in \mathbb{A}$ and $\mathsf{body}(\mathsf{r}[\alpha]) \subseteq \mathbb{A}$.

Adversary Computability

So far, we have been building the tools required to give our adversary the ability play around with the values they have observed during the execution of a program. This aims to model an adversary with the capability to perform cryptographic operations. We now make precise this notion of computability by defining the Adversary Computability relation $\mathsf{AdvComp}_{\mathcal{R}} : 2^{\mathbb{A}} \to 2^{\mathbb{A}}$, which is parameterized by a set of rules $\mathcal{R}$. Given a set of abstract values $\mathcal{K} \subseteq \mathbb{A}$, we have that

$$\mathsf{AdvComp}_{\mathcal{R}}(\mathcal{K}) = \{\mathsf{head}(\mathsf{r}[\alpha]) \mid \mathsf{r} \in \mathcal{R},\ \alpha : \mathsf{vars}(\mathsf{r}) \to \mathbb{A},\ \mathsf{body}(\mathsf{r}[\alpha]) \subseteq \mathcal{K}\}.$$

Intuitively, this function represents the possible inferences the adversary can make, making use of the rules specified by $\mathcal{R}$. We define $\mathsf{AdvComp}^*_{\mathcal{R}}(S)$ as the closure of the $\mathsf{AdvComp}_{\mathcal{R}}$ relation.

C.3.2 Adversary Knowledge

Our threat model consists of an adversary that can passively observe messages that are assigned to public set of variables $\mathcal{V}_p \subseteq \mathcal{V}_a$. Given an abstract state σ_a, we let $\mathsf{AK}(\sigma_a) = \{\sigma_a(v) \mid v \in \mathcal{V}_p\}$. We refer to this set $\mathsf{AK}_i(\sigma_a)$ as the adversary's knowledge.

Furthermore, for a fixed set of rules $\mathcal{R}$ and given a trace of abstract states $\tau = \langle \sigma_0, ..., \sigma_n \rangle$, let the set of all observed values, $\mathcal{S}_\tau$, be $\mathcal{S}_\tau = \bigcup\{\{\sigma_i(v) \mid v \in \mathcal{V}_p\} \mid \sigma_i \in \tau\}$. We make the assumption that the adversary has perfect memory, meaning that they retain all the values that they have observed over the execution of a program forever.

Then, we define the adversary's knowledge, $\mathsf{AK}_{\mathcal{R}}(\tau)$, as

$$\mathsf{AK}_{\mathcal{R}}(\tau) = \mathsf{AdvComp}^*_{\mathcal{R}}(\mathcal{S}_\tau).$$

Secret Inference Problem

Now that we have defined the observability and the computability of the adversary, we can go ahead and define the *secret inference problem.* As the name suggests, this problem asks whether an adversary can reason about the values they have observed during the execution of a program in order to infer some secret value(s). Intuitively, if an adversary is indeed able to infer any secrets, then the system is insecure. A keen reader will notice that we have not yet defined what it means for a system to be insecure. We will formally define our notion of security in the next subsection, then show that an adversary is able to solve the secret inference problem if and only if the system is insecure.

Now we proceed to state the secret inference problem formally. Let $\mathcal{P}$ be a program, $\mathcal{R}$ be a set of rules, and Σ_0 be a set of initial states. We will use the symbol $\mathcal{Q}$ to denote a set of *query values*, meaning that $\mathcal{Q} \subseteq \mathbb{A}$. Conceptually, $\mathcal{Q}$ contains all the secret values that we do not want the adversary to observe. Now, given the above parameters, we define a *configuration* to be the tuple $(\mathcal{P}, \mathcal{R}, \mathcal{Q}, \Sigma_0)$.

Next, we define the secret inference problem, $\mathsf{SIP}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$, as

$$\mathsf{SIP}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0) \Leftrightarrow \exists \sigma_0 \in \Sigma_0,\ \exists q \in \mathcal{Q}.\ q \in \mathsf{AK}_{\mathcal{R}}(\llbracket \mathcal{P} \rrbracket(\sigma_0)).$$

Essentially, $\mathsf{SIP}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$ asks the following question: "In a given configuration $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$, does there exist some execution of $\mathcal{P}$ starting from an initial state $\sigma_0 \in \Sigma_0$ such that a secret value $q \in \mathcal{Q}$ is deducible by the adversary using the rules $\mathcal{R}$?" If so, we say that the adversary has solved the secret inference problem, or that they have inferred a secret.

Note that the SIP is defined relative to a configuration $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$. This is because any change to the given system parameters potentially changes whether an adversary can solve the secret inference problem or not.

Now that we have formally defined the SIP, we must also develop a way to verify it. Thus, the aim of the next section is to develop a methodology for efficient verification of this problem. To this end, we will show how to symbolically execute programs, which will allow us to consider multiple executions of the program at once. Then, we will define what it means for a configuration $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$ to be secure. Finally, we will connect the secret inference problem and our notion of security, showing that a system is secure if and only if the adversary is unable to solve the secret inference problem.

C.4 Verification

We have dedicated the past two sections to building the machinery required for this section. In addition to executing ProtoSL programs, we also want to be able to efficiently verify them. To do this, we develop a methodology to symbolically simulate the aforementioned programs, a technique that allows us to consider many possible executions of a program at once without explicitly executing them.

After defining how we symbolically execute ProtoSL programs, we show how to verify them. In particular, we will show how we can leverage symbolic executions to determine whether an adversary can solve the SIP.

Symbolic Execution

For a given state σ, let σ_a denote only its $\mathbb{A}$-valued assignments and let σ_b denote only its $\mathbb{B}$-valued assignments. That is, $\sigma_a : \mathcal{V}_a \to \mathbb{A}$ and $\sigma_b : \mathcal{V} \to \mathbb{B}$.

Furthermore, for σ_b, we let ϕ denote its characteristic function. That is, $\phi(\sigma_b)$ is true if and only if $\sigma_b \vDash \phi$. We also overload the symbol ϕ to represent the set of all such states:

$$\phi = \{\sigma_b \in \Sigma_b \mid \sigma_b \vDash \phi\}$$

We refer to this formula ϕ as the *path condition*. While we represent this path condition as an explicit set of states here, in the actual execution implementation, this would be represented by a symbolic formula, following standard symbolic model checking conventions [48]. Furthermore, we refer to a triple $(\sigma_a, \phi, K) \in \Sigma_a \times 2^{\Sigma_b} \times 2^{\mathbb{A}}$ as a *symbolic execution context*. We will refer to the set of all symbolic execution contexts, $\Sigma_a \times 2^{\Sigma_b} \times 2^{\mathbb{A}}$, as Γ.

We provide the semantics of our execution in the form of an interpretation function $\langle\!\langle \mathsf{st} \rangle\!\rangle : \Gamma \to 2^{\Gamma}$ that returns a set of new execution contexts after executing the statement st. We lift the semantics of $\langle\!\langle \mathsf{st} \rangle\!\rangle$ to a set of symbolic execution contexts in the obvious way: if $C \in 2^{\Gamma}$, then $\langle\!\langle \mathsf{st} \rangle\!\rangle(C) = \bigcup\{\langle\!\langle \mathsf{st} \rangle\!\rangle(c) \mid c \in C\}$.

For clarity, we will make a distinction between the different kinds of variables contained in a collection of variables $\overline{x}$. In particular, we will use the subscript a to refer to $\mathbb{A}$-typed variables, the subscript b to refer to $\mathbb{B}$-typed variables, and the subscript p to refer to public $\mathbb{A}$-typed variables. Thus, we say that $\overline{x}_a = \{x \in \overline{x} \mid x \in \mathcal{V}_a\}$, that $\overline{x}_b = \{x \in \overline{x}_b \mid x \in \mathcal{V}_b\}$, and, naturally, $\overline{x}_p = \{x \in \overline{x} \mid x \in \mathcal{V}_p\}$. Similarly, we will define $\overline{e}_a$ as the set of expressions assigned to a variable in $\overline{x}_a$, $\overline{e}_b$ as the set of expressions assigned to a variable in $\overline{x}_b$, and $\overline{e}_p$ as the set of expressions assigned to a variable in $\overline{x}_p$.

For readability, we will let $w = [\![e]\!](\sigma)$ for a given state σ. In other words, w refers to the interpretation of e in state σ. We extend this definition to tuples: $\overline{w} = [\![\overline{e}]\!](\sigma)$. We will also use the following conventions, which are hopefully clear from context:

- $\overline{w}_a = [\![\overline{e}_a]\!](\sigma)$,
- $\overline{w}_b = [\![\overline{e}_b]\!](\sigma)$, and
- $\overline{w}_p = [\![\overline{e}_p]\!](\sigma)$.

With all that said, we formally define our symbolic executions below:

$$\langle\!\langle x \leftarrow e \rangle\!\rangle(\sigma_a, \phi, K) = \begin{cases} \{(\sigma_a[x \mapsto w], \phi, K \cup \{w\})\} & \text{if } x \in \mathcal{V}_p \\ \{(\sigma_a[x \mapsto w], \phi, K\})\} & \text{if } x \in \mathcal{V}_a \setminus \mathcal{V}_p \\ \{(\sigma_a, \{\sigma_b[x \mapsto w] \mid \sigma_b \in \phi\}, K)\} & \text{if } x \in \mathcal{V}_b \end{cases}$$

$$\langle\!\langle \overline{x} \leftarrow \overline{e} \rangle\!\rangle(\sigma_a, \phi, \mathcal{K}) = \{(\sigma_a[\overline{x}_a \mapsto \overline{w}_a], \{\sigma_b[\overline{x}_b \mapsto \overline{w}_b] \mid \sigma_b \in \phi\}, \mathcal{K} \cup \overline{w}_p)\}$$

$$\langle\!\langle \text{if } \psi \text{ then } \mathsf{st}_A \text{ else } \mathsf{st}_B \rangle\!\rangle(\sigma_a, \phi, K) = \langle\!\langle \mathsf{st}_A \rangle\!\rangle(\sigma_a, \phi \wedge \psi, K) \cup \langle\!\langle \mathsf{st}_B \rangle\!\rangle(\sigma_a, \phi \wedge \neg\psi, K)$$

$$\langle\!\langle \mathsf{st}_A \cdot \mathsf{st}_B \rangle\!\rangle(\sigma_a, \phi, K) = \langle\!\langle \mathsf{st}_B \rangle\!\rangle(\langle\!\langle \mathsf{st}_A \rangle\!\rangle(\sigma_a, \phi, K))$$

Starting from an initial context $\gamma_0 = (\sigma_{a0}, \phi_0, K_0) \in \Gamma$, we define a symbolic execution of the program $\mathcal{P} = \mathsf{st}_1 \cdots \mathsf{st}_n$ as a sequence of sets of contexts $\langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0) = \langle \gamma_0, \gamma_1, ..., \gamma_n \rangle$ where $\gamma_{i+1} = \langle\!\langle \mathsf{st}_{i+1} \rangle\!\rangle(\gamma_i)$. We will often write $\pi = \langle \gamma_0, \gamma_1, ..., \gamma_n \rangle$ to refer to a given symbolic execution. Additionally, we allow indexing for convenience: if $\pi = \langle \gamma_0, \gamma_1, ..., \gamma_n \rangle$, then $\pi_i = \gamma_i$.

Without loss of generality , we define $\mathcal{K}_0 = \{\sigma_{a0}(v) \mid v \in \mathcal{V}_p\}$. Moreover, we assume that any initial context must be satisfiable; that is $\forall (\sigma_{a0}, \phi_0, \mathcal{K}_0) \in \Gamma_0. \ \mathsf{SAT}(\phi_0)$. Finally, we say that if $\mathcal{P} = \varnothing$, then $\langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0) = \gamma_0$.

Security Verification Condition

Recall that we initially wanted to keep certain values secret from the adversary. We call these values "queries," and denote the set of all such values as $\mathcal{Q}$. Now we can finally state our definition of security as follows:

Let $\mathcal{P}$ be a program and $\Sigma_0 \subseteq \Sigma$ be a set of initial states. Let Γ_0 be the set of initial execution contexts obtained from Σ_0. Then we use the symbol $\mathcal{F}$ to denote the set of final execution contexts:

$$\mathcal{F} = \{\langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0)_n \mid \gamma_0 \in \Gamma_0\}.$$

Now that we have all the necessary definitions at hand, we can formally define what it means for a given system $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$ to be *secure*:

Definition C.1. System Security: Let $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$ be a system. Then we say that

$$\mathsf{Secure}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0) \Leftrightarrow \forall(\sigma_a, \phi, \mathcal{K}) \in \mathcal{F}.\ \neg\mathsf{SAT}(\phi) \vee (\mathsf{AdvComp}^*_{\mathcal{R}}(\mathcal{K}) \cap \mathcal{Q}) = \varnothing.$$

Intuitively, $\mathsf{Secure}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$ indicates that in any symbolic execution path in $\mathcal{F}$, either that path is not reachable (this is what we mean by $\neg\mathsf{SAT}(\phi)$ above), or the adversary is unable to deduce any of the secret values in $\mathcal{Q}$ (this is what we mean when we say that $(\mathsf{AdvComp}^*_{\mathcal{R}}(\mathcal{K}) \cap \mathcal{Q}) = \varnothing$)).

Equivalence of SIP and Security Verification Condition

It is finally time to put all the pieces of the puzzle together. Recall that our security goal was to prevent an adversary from being able to infer any sensitive values from the data they were able to see. Now that we have a precise formulation of what exactly it means for an adversary to be able to infer a secret, as well as what it means for the system to be secure, we can show that our symbolic execution is sound and complete.

Theorem 1. Correctness: Given a program $\mathcal{P}$, a set of rules $\mathcal{R}$, a set of initial states $\Sigma_0 \subseteq \Sigma$, and a set of query values $\mathcal{Q}$, we have that

$$\mathsf{SIP}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0) \Leftrightarrow \neg\mathsf{Secure}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0).$$

We prove this equivalence in Appendix D. This effectively gives us a characterization of correctness of SSV. What this does is guarantee that our approach is sound, and that we can trust the results produced by our solver.

Appendix D

Proof of Equivalence of the Secret Inference Problem and the Security Verification Condition

In this appendix, we prove the following equivalence:

$$\mathsf{SIP}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0) \Leftrightarrow \neg\mathsf{Secure}(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$$

for an arbitrary system specification $(\mathcal{P}, \mathcal{Q}, \mathcal{R}, \Sigma_0)$.

The first step required to prove this property is to show that the knowledge an adversary obtains through concrete and symbolic executions of $\mathcal{P}$ are effectively the same. From this fact, the equivalence easily follows. We formulate this "knowledge equivalence" as soundness and completeness lemmas, with respect to the knowledge obtained by an adversary. Intuitively, the soundness lemma tells us that any reachable symbolic execution context must correspond to a concrete trace of the program. Conversely, our completeness lemma states that every concrete trace of our program can be symbolically simulated. To reiterate, in both lemmas, the main property we care about is that the knowledge obtained by the adversary at any point of the execution is the same on the concrete and symbolic side.

Before we state and prove our lemmas, let us define some notational conventions, starting with those for a concrete execution trace τ. Let $\mathcal{P}$ be a program of length $n \in \mathbb{N}$ and $\Sigma_0 \subseteq \Sigma$ be a set of initial states. For any choice of $\sigma_0 \in \Sigma_0$, we have that $\tau = [\![\mathcal{P}]\!](\sigma_0) = \langle \sigma_0, \sigma_1, ..., \sigma_n \rangle$, where the ith state in the trace τ is $\sigma_i = \tau_i$. As a reminder, we have that the set of observed values in a trace τ is defined as

$$\mathcal{S}_\tau = \bigcup \{ \{ \sigma_i(v) \mid v \in \mathcal{V}_p \} \mid \sigma_i \in \tau \}.$$

On the side of symbolic executions, we begin with the initial symbolic execution context $\gamma_0 = \{(\sigma_{a0}, \phi_0, \mathcal{K}_0)\}$, where

- σ_{a0} is the restriction of σ_0 to $\mathbb{A}$-typed variables,

- σ_{0b} is the restriction of σ_0 to $\mathbb{B}$-typed variables,
- ϕ_0 is the characteristic function of σ_{0b}, and
- $\mathcal{K}_0 = \{\sigma_{a0}(v) \mid v \in \mathcal{V}_p\}$.

Analogous to a concrete trace τ, we have that a symbolic execution trace is $\pi = \langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0) = \langle \gamma_0, \gamma_1, ..., \gamma_n \rangle$, where $\gamma_i = \pi_i$. Each γ_i is a set of symbolic execution contexts, and each one of its elements is of the form $(\sigma_{ai}, \phi_i, \mathcal{K}_i)$. Again, as a reminder:

- σ_{ai} symbolizes the current assignment of $\mathbb{A}$-typed variables,
- ϕ_i is the path condition of the symbolic execution context, and
- $\mathcal{K}_i$ is the set of values observed by the adversary up to and including the current symbolic execution context.

Definition D.1. Characteristic $\mathbb{B}$-Function: Let $\sigma \in \Sigma$ be a variable assignment. Then we define σ's characteristic $\mathbb{B}$-formula, β_σ, such that:

$$\beta_\sigma \Leftrightarrow \bigwedge_{v \in \mathcal{V}_b} v = \sigma(v).$$

Definition D.2. Path Condition: Let α be a formula and β_σ be the characteristic $\mathbb{B}$-function of some $\sigma \in \Sigma$. Then we say that a formula ϕ is a path condition if

$$\phi \Leftrightarrow \alpha \wedge \beta_\sigma.$$

Definition D.3. Reassignment 1: Let β_σ be a characteristic $\mathbb{B}$-function for some $\sigma \in \Sigma$. Next, let $x \in \mathcal{V}_b$ and $w \in \mathbb{B}$. Then we define the formula $\beta_\sigma[x \mapsto w]$ as

$$\beta_\sigma[x \mapsto w] \Leftrightarrow x = w \wedge \bigwedge_{v \in \mathcal{V}_b \setminus \{x\}} v = \sigma(v).$$

Remark D.1. *If $\sigma \in \Sigma$ and β_σ is its characteristic $\mathbb{B}$-function, then $\beta_\sigma[x \mapsto w]$ is the characteristic $\mathbb{B}$-function of $\sigma[x \mapsto w]$.*

Definition D.4. Reassignment 2: Let ϕ be a path condition; that is, $\phi \Leftrightarrow \alpha \wedge \beta_\sigma$ for some formula α and some characteristic $\mathbb{B}$-function β_σ. Let $x \in \mathcal{V}_b$ and $w \in \mathbb{B}$. Then we define a reassignment on the path condition, $\phi[x \mapsto w]$, as:

$$\phi[x \mapsto w] \Leftrightarrow \alpha \wedge \beta_\sigma[x \mapsto w].$$

Remark D.2. *If θ is not a path condition or a characteristic $\mathbb{B}$-function, then $\theta[x \mapsto w] \Leftrightarrow \theta$.*

Definition D.5. Containment: Let $\gamma \in \Gamma$ and $\sigma \in \Sigma$. We say that γ contains σ if there exists $(\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma$ such that

$$\sigma_a = \sigma_a^* \wedge \sigma_b \vDash \phi^*$$

Definition D.6. Equivalence: Σ and Γ are equivalent when they 'contain' each other: every $\sigma \in \Sigma$ is contained in some $\gamma \in \Gamma$ and every $\gamma \in \Gamma$ is such that every σ contained in γ is an element of Σ.

With these definitions in mind, we can finally state our lemmas, beginning with our soundness lemma:

Lemma D.1 (Soundness)**.** *Suppose $\mathcal{P}$ is a program and Σ_0 is a set of initial states. Let Γ_0 be a set of contexts that contains Σ_0. For all $\sigma_0 \in \Sigma_0$, suppose $\tau = \langle \sigma_0, ..., \sigma_n \rangle = [\![\mathcal{P}]\!](\sigma_0)$. Then there exists, $\gamma_0 \in \Gamma_0$ with $\pi = \langle \gamma_0, ..., \gamma_n \rangle = \langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0)$ and $\gamma_n = (\sigma_a^*, \phi^*, K^*)$ such that:*

(a) γ_n contains σ_n and

(b) $S_\tau = \mathcal{K}^*$.

Proof. Fix some arbitrary $\Sigma_0 \subseteq \Sigma$ and some Γ_0 that contains Σ_0. Additionally, fix some arbitrary $\sigma_0 \in \Sigma_0$. We proceed by induction on the length of the program $\mathcal{P}$.
<u>Base Case:</u> For our base case, we must check that the property holds for programs of length 0 (i.e. when $\mathcal{P} = \varnothing$). First and foremost, we have that $\tau = [\![\mathcal{P}]\!](\sigma_0) = \langle \sigma_0 \rangle$, meaning that $\sigma_n = \sigma_0$.

Consider $(\sigma_a^*, \phi^*, \mathcal{K}^*)$, where

$$\begin{aligned} &\sigma_a^* = (\sigma_0)_a, \\ &\phi^* \Leftrightarrow \texttt{true}, \text{ and} \\ &\mathcal{K}^* = \{\sigma_a^*(v) \mid v \in \mathcal{V}_p\}. \end{aligned}$$

Clearly, $\sigma_0 \vDash \texttt{true}$. So $(\sigma_0)_b \vDash \phi^*$. This means that

$$(\sigma_0)_a = \sigma_a^* \wedge (\sigma_0)_b \vDash \phi^*.$$

So γ_0 contains σ_0. Thus, we can deduce that that the set $\gamma_0 = \{(\sigma_a^*, \phi^*, \mathcal{K}^*)\}$ is in Γ_0.

Now, we have that $\pi = \langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0) = \langle \gamma_0 \rangle$. In other words, $\gamma_n = \gamma_0$.

Furthermore, we have that by definition:

$$\begin{aligned} \mathcal{S}_\tau &= \mathcal{S}_{\langle \sigma_0 \rangle} \\ &= \bigcup \{\{\sigma_i(v) \mid v \in \mathcal{V}_p\} \mid \sigma_i \in \langle \sigma_0 \rangle\} \\ &= \{\sigma_0(v) \mid v \in \mathcal{V}_p\} \\ &= \{(\sigma_0)_a(v) \mid v \in \mathcal{V}_p\} \\ &= \{\sigma_a^*(v) \mid v \in \mathcal{V}_p\} \\ &= \mathcal{K}^*. \end{aligned}$$

Thus our base case holds.
<u>Inductive Hypothesis:</u> For our inductive hypothesis, we will assume that for any program $\mathcal{P}$ of length n, there exists some $\gamma_0 \in \Gamma_0$ with $\pi = \langle \gamma_0, ..., \gamma_n \rangle = \langle\!\langle \mathcal{P} \rangle\!\rangle(\gamma_0)$ such that there is some $(\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n$ so that:

(a) γ_n contains σ_n and

(b) $\mathcal{S}_\tau = \mathcal{K}^*$.

Inductive Step: Let $\mathcal{P}$ be a program of length $n+1$. More explicitly, $\mathcal{P} = \mathsf{st}_1 \cdots \mathsf{st}_n \cdot \mathsf{st}_{n+1}$. As usual, we will let $\tau = [\![\mathcal{P}]\!](\sigma_0)$. Let $\tau^- = [\![\mathsf{st}_1 \cdots \mathsf{st}_n]\!](\sigma_0) = \langle \sigma_0, ..., \sigma_n \rangle$, or the execution of only the first n statements of $\mathcal{P}$. By our inductive hypothesis, there exists some $\gamma_0 \in \Gamma_0$ such that $\pi^- = \langle\!\langle \mathsf{st}_1 \cdots \mathsf{st}_n \rangle\!\rangle(\gamma_0) = \langle \gamma_0, ..., \gamma_n \rangle$, and, furthermore, there exists some $(\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n$ so that

(a) γ_n contains σ_n and

(b) $\mathcal{S}_{\tau^-} = \mathcal{K}^*$.

Let $\sigma_{n+1} = [\![\mathsf{st}_{n+1}]\!](\sigma_n)$ and $\gamma_{n+1} = \langle\!\langle \mathsf{st}_{n+1} \rangle\!\rangle(\sigma_n)$. For visual clarity, we will use the notation $(\sigma_a^*, \phi^*, \mathcal{K}^*)$ for elements of γ_n and $(\sigma_a', \phi', \mathcal{K}')$ for elements of γ_{n+1}. With that said, our goal is to show that there exists some $(\sigma_a', \phi', \mathcal{K}') \in \gamma_{n+1}$ such that

(a) γ_{n+1} contains σ_{n+1} and

(b) $\mathcal{S}_\tau = \mathcal{K}'$.

We will show that this is true by structural induction. Our base cases will be those where $\mathsf{st}_{n+1} = x \leftarrow e$ and $\mathsf{st}_{n+1} = \overline{x} \leftarrow \overline{e}$. Our inductive cases will be where $\mathsf{st}_{n+1} =$ if ψ then st_A else st_B and $\mathsf{st}_{n+1} = \mathsf{st}_A \cdot \mathsf{st}_B$. Let us begin.

$\mathsf{st}_{n+1} = x \leftarrow e$: For this statement, we have that $\sigma_{n+1} = \sigma_n[x \mapsto e_{\sigma_n}]$.

There are three disjoint cases[1] to consider: $x \in \mathcal{V}_a \setminus \mathcal{V}_p$, $x \in \mathcal{V}_p$, and $x \in \mathcal{V}_b$. Let us now show what happens in each case.

(i) $x \in \mathcal{V}_a \setminus \mathcal{V}_p$: By our inductive hypothesis, we know that there is some $(\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n$ such that

$$(\sigma_n)_a = \sigma_a^* \wedge (\sigma_n)_b \vDash \phi^*.$$

Recall that in the case that $x \in \mathcal{V}_a \setminus \mathcal{V}_p$, we have that

$$\gamma_{n+1} = \{(\sigma_a^*[x \mapsto e_{\sigma_a}], \phi^*, \mathcal{K}^*) \mid (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n\}.$$

This means that for this $(\sigma_a^*, \phi^*, \mathcal{K}^*)$, there must exist some $(\sigma_a', \phi', \mathcal{K}')$ so that

$$\begin{aligned} &\sigma_a' = \sigma_a^*[x \mapsto e_{\sigma_n}], \\ &\phi' \Leftrightarrow \phi^*, \text{ and} \\ &\mathcal{K}' = \mathcal{K}^*. \end{aligned}$$

The above facts imply that there exists some $(\sigma_a', \phi', \mathcal{K}') \in \gamma_{n+1}$ such that

[1]This follows from the fact that $\mathcal{V} = \mathcal{V}_a \uplus \mathcal{V}_b$ and $\mathcal{V}_p \subseteq \mathcal{V}_a$. This implies that $\mathcal{V}_a = \mathcal{V}_p \uplus (\mathcal{V}_a \setminus \mathcal{V}_p)$. This means that $\mathcal{V} = \mathcal{V}_p \uplus (\mathcal{V}_a \setminus \mathcal{V}_p) \uplus \mathcal{V}_b$

So we have that

$$\begin{aligned}(\sigma_{n+1})_a &= (\sigma_n)_a[x \mapsto e_{\sigma_n}]\\ &= \sigma_a^*[x \mapsto e_{\sigma_n}]\\ &= \sigma_a'.\end{aligned}$$

Note that $(\sigma_{n+1})_b = (\sigma_n)_b$ and $\phi' \Leftrightarrow \phi^*$. This means that

$$\begin{aligned}&(\sigma_n)_b \vDash \phi^*\\ \Leftrightarrow\ &(\sigma_{n+1})_b \vDash \phi^*\\ \Leftrightarrow\ &(\sigma_{n+1})_b \vDash \phi'.\end{aligned}$$

Finally, note that

$$\begin{aligned}\mathcal{S}_\tau &= \{\{\sigma_i(v) \mid v \in \mathcal{V}_p\} \mid \sigma_i \in \tau\}\\ &= \{\{\sigma_i(v) \mid v \in \mathcal{V}_p\} \mid \sigma_i \in \tau^-\} \cup \{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p\}\\ &= \mathcal{S}_{\tau^-} \cup \{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p\}.\end{aligned}$$

Since $x \notin \mathcal{V}_p$, we know that $(\sigma_{n+1})_p = (\sigma_n)_p$. It follows that

$$\begin{aligned}\{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p\} &= \{(\sigma_{n+1})_p(v) \mid v \in \mathcal{V}_p\}\\ &= \{(\sigma_n)_p(v) \mid v \in \mathcal{V}_p\}\\ &= \{\sigma_n(v) \mid v \in \mathcal{V}_p\}.\end{aligned}$$

So it must be true that

$$\begin{aligned}\mathcal{S}_\tau &= \mathcal{S}_{\tau^-} \cup \{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p\}\\ &= \mathcal{S}_{\tau^-} \cup \{\sigma_n(v) \mid v \in \mathcal{V}_p\}\\ &= \mathcal{S}_{\tau^-}.\end{aligned}$$

The main consequence of this fact is that

$$\begin{aligned}\mathcal{S}_\tau &= \mathcal{S}_{\tau^-}\\ &= \mathcal{K}^*\\ &= \mathcal{K}'.\end{aligned}$$

Putting all the information together, we now know that there is some $(\sigma_a', \phi', \mathcal{K}') \in \gamma_{n+1}$ such that

(a) $(\sigma_{n+1})_a = \sigma_a' \wedge (\sigma_{n+1})_b \vDash \phi'$, meaning that γ_{n+1} contains σ_{n+1}.

(b) $\mathcal{S}_\tau = \mathcal{K}'$.

Therefore, our soundness property holds in this case.

(ii) $\underline{x \in \mathcal{V}_p}$: Given that $\mathcal{V}_p \subseteq \mathcal{V}_a$, this case is handled almost exactly as the previous one. For $x \in \mathcal{V}_p$, we have that

$$\gamma_{n+1} = \{(\sigma_a^*[x \mapsto e_{\sigma_n}], \phi^*, \mathcal{K}^* \cup \{e_{\sigma_n}\}) \mid (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n\}.$$

We can use the same reasoning as above to show that γ_{n+1} contains σ_{n+1}. Therefore, we will now focus on showing that $\mathcal{S}_\tau = \mathcal{K}'$.

By our inductive hypothesis, there exists some $(\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n$ such that $\mathcal{S}_{\tau^-} = \mathcal{K}^*$. Earlier, we showed that $\mathcal{S}_\tau = \mathcal{S}_{\tau-} \cup \{\sigma_{n+1} \mid v \in \mathcal{V}_p\}$. We can go a step further and observe that

$$\begin{aligned}\{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p\} &= \{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p \setminus \{x\}\} \cup \{\sigma_{n+1}(x)\} \\ &= \{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p \setminus \{x\}\} \cup \{e_{\sigma_n}\}\end{aligned}$$

Note that since $\sigma_{n+1} = \sigma_n[x \mapsto e_{\sigma_n}]$, we have that

$$\{\sigma_{n+1}(v) \mid v \in \mathcal{V}_p \setminus \{x\}\} = \{\sigma_n(v) \mid v \in \mathcal{V}_p \setminus \{x\}\}.$$

Since $\sigma_n \in \tau^-$, we also have that $\{\sigma_n(v) \mid v \in \mathcal{V}_p \setminus \{x\}\} \subseteq \mathcal{S}_\tau^-$. This implies that

$$\begin{aligned}\mathcal{S}_\tau &= \mathcal{S}_{\tau^-} \cup \{\sigma_n(v) \mid v \in \mathcal{V}_p \setminus \{x\}\} \cup \{e_{\sigma_n}\} \\ &= \mathcal{S}_{\tau^-} \cup \{e_{\sigma_n}\} \\ &= \mathcal{K}'.\end{aligned}$$

So indeed, we have that

(a) γ_{n+1} contains σ_{n+1}, and
(b) $\mathcal{S}_\tau = \mathcal{K}'$.

Hence, soundness holds in this case as well.

(iii) $\underline{x \in \mathcal{V}_b}$: Again, by our inductive hypothesis, there is some $(\sigma_a^*.\phi^*, \mathcal{K}^*) \in \gamma_n$ so that

$$(\sigma_n)_a = \sigma_a^* \wedge (\sigma_n)_b \vDash \phi^*.$$

Recall that when $x \in \mathcal{V}_b$, we define γ_{n+1} as

$$\gamma_{n+1} = \{(\sigma_a^*, \phi^*[x \mapsto e_{\sigma_n}], \mathcal{K}^*) \mid (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n\}.$$

So there must be some corresponding $(\sigma_a', \phi', \mathcal{K}') \in \gamma_{n+1}$ so that

$$\begin{aligned}&\sigma_a' = \sigma_a^*, \\ &\phi' \Leftrightarrow \phi^*[x \mapsto e_{\sigma_n}], \text{ and} \\ &\mathcal{K}' = \mathcal{K}^*.\end{aligned}$$

By definition of a path condition, $\phi^* \Leftrightarrow \alpha \wedge \beta_\sigma$ for some $\sigma \in \Sigma$. So

$$\sigma_n \vDash \phi^* \Leftrightarrow \sigma_n \vDash \alpha \wedge \beta_\sigma$$

Since $\sigma_n \vDash \beta_\sigma$, it must be true that

$$\begin{aligned}\beta_\sigma &\Leftrightarrow \beta_{\sigma_n} \\ &\Leftrightarrow \bigwedge_{v \in \mathcal{V}_b} v = \sigma_n(v)\end{aligned}$$

This means that we can express our new path condition, ϕ', as

$$\begin{aligned}\phi' &\Leftrightarrow \phi^*[x \mapsto e_{\sigma_n}] \\ &\Leftrightarrow (\alpha \wedge \beta_{\sigma_n})[x \mapsto e_{\sigma_n}] \\ &\Leftrightarrow \alpha \wedge \beta_{\sigma_n}[x \mapsto e_{\sigma_n}] \\ &\Leftrightarrow \alpha \wedge (x = e_{\sigma_n} \wedge \bigwedge_{x \in \mathcal{V}_b \setminus \{x\}} v = \sigma_n(v)).\end{aligned}$$

We'll define $\beta_{\sigma_{n+1}}$ as

$$(x = e_{\sigma_n} \wedge \bigwedge_{x \in \mathcal{V}_b \setminus \{x\}} v = \sigma_n(v)).$$

Now given that $\sigma_{n+1} = \sigma_n[x \mapsto e_{\sigma_n}]$, we have that

$$\begin{aligned}&\sigma_{n+1} \vDash x = e_{\sigma_n} \\ \wedge\ &\sigma_{n+1} \vDash \bigwedge_{v \in \mathcal{V}_b \setminus \{x\}} v = \sigma_n.\end{aligned}$$

So it follows that

$$\begin{aligned}&\sigma_{n+1} \vDash x = e_{\sigma_n} \bigwedge_{v \in \mathcal{V}_b \setminus \{x\}} v = \sigma_n \\ \Leftrightarrow\ &\sigma_{n+1} \vDash \beta_{\sigma_{n+1}}.\end{aligned}$$

Next, since α is simply a constraint over the symbolic inputs, and $\sigma_n \vDash \alpha$, it must be true that $\sigma_{n+1} \vDash \alpha$. So $\sigma_{n+1} \vDash \alpha \wedge \beta_{\sigma_{n+1}}$, which implies that $\sigma_{n+1} \vDash \phi'$.

Moreover, since $(\sigma_{n+1})_a = (\sigma_n)_a$, it is easy to see that $\mathcal{S}_\tau = \mathcal{S}_{\tau^-}$.

Thus, we have that there exists some $(\sigma'_a, \phi', \mathcal{K}') \in \gamma_{n+1}$ such that

(a) $\sigma_{n+1} = \sigma_a \wedge \sigma_{n+1} \vDash \phi$, and

(b) $\mathcal{S}_\tau = \mathcal{K}'$.

Therefore, we can say that soundness holds for the case that $x \in \mathcal{V}_b$.

$\underline{\mathsf{st}_{n+1} = \overline{x} \leftarrow \overline{e}}$: This case is similar to the one in which $\mathsf{st}_{n+1} = x \leftarrow e$. The only difference is that everything happens simultaneously. Naturally, we have that $\sigma_{n+1} = \sigma_n[\overline{x} \leftarrow \overline{e}_{\sigma_n}]$. Let $\overline{x}_a \subseteq \overline{x}$, $\overline{x}_b \subseteq \overline{x}$, and $\overline{x}_p \subseteq \overline{x}_a$ be the natural projections of $\overline{x}$ into the appropriate types. Then we have that γ_{n+1} is the set

$$\gamma_{n+1} = \{(\sigma_a^*[\overline{x}_a \mapsto (\overline{e}_{\sigma_n})_a], \phi^*[\overline{x}_b \mapsto (\overline{e}_{\sigma_n})_b], \mathcal{K}^* \cup (\overline{e}_{\sigma_n})_p) \mid (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n\}.$$

By our inductive assumption, there is some $(\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n$ so that σ_n is contained in $(\sigma_a^*, \phi^*, \mathcal{K}^*)$ and $\mathcal{S}_{\tau^-} = \mathcal{K}^*$. So there must be some corresponding $(\sigma_a', \phi', \mathcal{K}') \in \gamma_{n+1}$ such that

$$\begin{aligned}\sigma_a' &= \sigma_a^*[\overline{x} \mapsto \overline{e}_{\sigma_n}],\\ \phi' &\Leftrightarrow \phi^*[\overline{x} \mapsto \overline{e}_{\sigma_n}], \text{ and}\\ \mathcal{K}' &= \mathcal{K}^* \cup (\overline{e}_{\sigma_n})_p.\end{aligned}$$

By the same reasoning as above, we have that

$$\begin{aligned}(\sigma_{n+1})_a &= (\sigma_n[\overline{x} \mapsto \overline{e}_{\sigma_n}])_a\\ &= \sigma_a^*[\overline{x} \mapsto \overline{e}_{\sigma_n}]\\ &= \sigma_a'.\end{aligned}$$

By our inductive hypothesis, $\sigma_n \vDash \alpha^*$, and we know that by construction of γ_{n+1}, $\alpha' \Leftrightarrow \alpha^*$. So we have that $\sigma_{n+1} \vDash \alpha'$. Our inductive hypothesis also tells us that $(\sigma_n)_b = \beta^*$. So we have that

$$\begin{aligned}(\sigma_{n+1})_b &= (\sigma_n[\overline{x} \mapsto \overline{e}_{\sigma_n}])_b\\ &= \beta^*[\overline{x} \mapsto \overline{e}_{\sigma_n}]\\ &= \beta'.\end{aligned}$$

Ultimately, this means that $\sigma_{n+1} \vDash \phi'$. This information tells us that we meet the criterion (a) for soundness: $(\sigma_a', \phi', \mathcal{K}')$ contains σ_{n+1}.

Next, we want to show that criterion (b) holds: that $\mathcal{S}_\tau = \mathcal{K}'$. By our inductive hypothesis and our prior reasoning steps, we can show that

$$\begin{aligned}\mathcal{S}_\tau &= \mathcal{S}_{\tau^-} \cup \overline{e}_{\sigma_n}\\ &= \mathcal{K}^* \cup \overline{e}_{\sigma_n}\\ &= \mathcal{K}'.\end{aligned}$$

This shows us that, indeed, there is some $(\sigma_a', \phi', \mathcal{K}') \in \gamma_{n+1}$ such that

(a) g_{n+1} contains σ_{n+1}.

(b) $\mathcal{S}_\tau = \mathcal{K}'$.

$\underline{\mathsf{st}_{n+1} = \text{if } \psi \text{ then } \mathsf{st}_A \text{ else } \mathsf{st}_B}$: Before we begin, it will be useful to define some notation. Recall that for this kind of statement, we have that

$$\sigma_{n+1} = \begin{cases} [\![\mathsf{st}_A]\!](\sigma_n) & \text{if } \psi(\sigma_n) \\ [\![\mathsf{st}_B]\!](\sigma_n) & \text{otherwise.} \end{cases}$$

For convenience, let σ_{n+1}^A be shorthand for $[\![\mathsf{st}_A]\!](\sigma_n)$. Similarly, let σ_{n+1}^B be shorthand for $[\![\mathsf{st}_B]\!](\sigma_n)$. Also, recall that by definition,

$$\gamma_{n+1} = \langle\!\langle \mathsf{st}_A \rangle\!\rangle(\gamma_n^{\psi}) \cup \langle\!\langle \mathsf{st}_B \rangle\!\rangle(\gamma_n^{\neg\psi}).$$

As before, we have that

$$\gamma_n^{\psi} = \{(\sigma_a^*, \phi^* \wedge \psi, \mathcal{K}^*) \mid (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n\}, \text{ and}$$
$$\gamma_n^{\neg\psi} = \{(\sigma_a^*, \phi^* \wedge \neg\psi, \mathcal{K}^*) \mid (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n\}.$$

Consistent with our notation above, we can define

$$\gamma_{n+1}^A = \langle\!\langle \mathsf{st}_A \rangle\!\rangle(\gamma_n^{\psi}), \text{ and}$$
$$\gamma_{n+1}^B = \langle\!\langle \mathsf{st}_B \rangle\!\rangle(\gamma_n^{\neg\psi}).$$

So we can rewrite γ_{n+1} as

$$\gamma_{n+1} = \gamma_{n+1}^A \cup \gamma_{n+1}^B.$$

With the notation above defined, we can proceed with the rest of the proof. Note that for this kind of statement, there are two possibilities that we must consider. Either

(i) $\psi(\sigma_n)$ is true, or

(ii) $\neg\psi(\sigma_n)$ is true.

In the case that (i) is true, we execute st_A, and in the case that (ii) is true, we must execute st_B. Let us now show that soundness holds in both of these cases.

Case (i): By our inductive hypothesis, there must be some $g_n = (\sigma_a^*, \phi^*, \mathcal{K}^*) \in \gamma_n$ such that g_n contains σ_n. Since g_n contains σ_n, it must be true that $\sigma_n \vDash \phi$. This, along with the fact that $\psi(\sigma_n)$ is true, tells us that $\sigma_n \vDash \phi \wedge \psi$. It follows that $g_n^{\psi} = (\sigma_a^*, \phi^* \wedge \psi, \mathcal{K}^*)$ contains σ_n. Note that $g_n^{\psi} \in \gamma_n^{\psi}$.

By our inductive assumption on st_A, since g_n^{ψ} contains σ_n and $\mathcal{S}_{\tau^-} = \mathcal{K}^*$, there must exist some $g_{n+1}^A = (\sigma_a^A, \phi^A, \mathcal{K}^A) \in \gamma_{n+1}^A$ such that

(a) g_{n+1}^A contains σ_{n+1}^A,

(b) and $\mathcal{S}_{\tau^A} = \mathcal{K}^A$.

Since clearly $\gamma^A_{n+1} \subseteq \gamma_{n+1}$, we have that $g^A_{n+1} \in \gamma_{n+1}$.

Next, since

$$\begin{aligned}\tau &= \tau^- \cdot \sigma_{n+1}\\ &= \tau^- \cdot \sigma^A_{n+1}\\ &= \tau^A,\end{aligned}$$

it must be true that $\mathcal{S}_\tau = \mathcal{S}_{\tau^A}$. So

$$\begin{aligned}\mathcal{S}_\tau &= \mathcal{S}_{\tau^A}\\ &= \mathcal{K}^A.\end{aligned}$$

This all tells us that there exists some $g^A_{n+1} = (\sigma^A_a, \phi^A, \mathcal{K}^A) \in \gamma_{n+1}$ such that

(a) g^A_{n+1} contains σ_{n+1}, and

(b) $\mathcal{S}_\tau = \mathcal{K}^A$.

This proves that soundness holds when $\psi(\sigma_n)$ is true.

Case (ii): The proof for the case where $\neg\psi(\sigma_n)$ is true is nearly identical to the one we just did. Systematically replacing every occurrence of ψ above with $\neg\psi$ and replacing A with B yields the desired soundness result for Case (ii). For the sake of brevity, we omit the full details and conclude simply that there must exist some $g^B_{n+1} = (\sigma^B_a, \phi^B, \mathcal{K}^B) \in \gamma_{n+1}$ such that

(a) g^B_{n+1} contains σ_{n+1}, and

(b) $\mathcal{S}_\tau = \mathcal{K}^B$.

$\mathsf{st}_{n+1} = \mathsf{st}_A \cdot \mathsf{st}_B$: In this final case, we once again assume inductively that everything works as expected with st_A and st_B. Recall that

$$\begin{aligned}\sigma_{n+1} &= [\![\mathsf{st}_B]\!]([\![\mathsf{st}_A]\!](\sigma_n)), \text{ and}\\ \gamma_{n+1} &= \langle\!\langle \mathsf{st}_B \rangle\!\rangle(\langle\!\langle \mathsf{st}_A \rangle\!\rangle(\gamma_n)).\end{aligned}$$

First, let $\sigma^A_{n+1} = [\![\mathsf{st}_A]\!](\sigma_n)$ and let $\gamma^A_{n+1} = \langle\!\langle \mathsf{st}_A \rangle\!\rangle(\gamma_n)$.

By almost the same reasoning as in the previous statement, the following must hold: there exists some $g^A_{n+1} = (\sigma^A_a, \phi^A, \mathcal{K}^A)$ such that

(a) g^A_{n+1} contains σ^A_{n+1}, and

(b) $\mathcal{S}_{\tau^A} = \mathcal{K}^A$.

Next, let $\sigma^B_{n+1} = [\![\mathsf{st}_B]\!](\sigma^A_{n+1})$. Clearly, $\sigma_{n+1} = \sigma^B_{n+1}$. Furthermore, let $\gamma^B_{n+1} = \langle\!\langle \mathsf{st}_B \rangle\!\rangle(\gamma^A_{n+1})$. It is easy to see that there must exist some $g^B_{n+1} = (\sigma^B_a, \phi^B, \mathcal{K}^B) \in \gamma_{n+1}$ such that

(a) g^B_{n+1} contains σ^B_{n+1}, and

(b) $\mathcal{S}_{\tau^{AB}} = \mathcal{K}^B$.

Thus, since $\tau \cong \tau^{AB}$ and $\gamma_{n+1}^B = \gamma_{n+1}$, we must have that soundness holds for the composition of statements st_A and st_B, too.

This completes the induction, effectively proving that the soundness property must hold for any program $\mathcal{P}$.

□

The proof of completeness proceeds in an analogous fashion. To avoid redundancy, we omit it here and leave its reconstruction as an exercise for the reader.

Appendix E

BtorSec Language Details

In this appendix, we dive deeper into the details of the BtorSec language. We first present a more detailed treatment of the syntax of BtorSec programs in Section E.1. We present a grammar and outline the supported operations of the language. Subsequently, in Section E.2 we present the structure of SecSpecs in the form of a JSON Schema.

E.1 The Syntax of BTORSEC

```
    ⟨num⟩ ::= positive unsigned integer (greater than zero)
   ⟨uint⟩ ::= unsigned integer (including zero)
 ⟨string⟩ ::= sequence of whitespace and printable characters without [newline]
 ⟨symbol⟩ ::= sequence of printable characters without [newline]
⟨comment⟩ ::= ‘;’⟨ string⟩
    ⟨nid⟩ ::= ⟨num⟩
    ⟨sid⟩ ::= ⟨num⟩
  ⟨const⟩ ::= ‘const’⟨sid⟩[0-1]+
 ⟨constd⟩ ::= ‘constd’⟨sid⟩[‘-’]⟨uint⟩
  ⟨const⟩ ::= ‘consth’⟨sid⟩[0-9a-fA-F]+
  ⟨input⟩ ::= (‘input’|‘one’|‘ones’|‘zero’) ⟨sid⟩ | ⟨const⟩ | ⟨constd⟩ | ⟨consth⟩
  ⟨state⟩ ::= ‘state′ ⟨sid⟩
 ⟨bitvec⟩ ::= ‘bitvec’ ⟨num⟩
  ⟨array⟩ ::= ‘array’ ⟨sid⟩ ⟨sid⟩
   ⟨node⟩ ::= ⟨nid⟩‘sort’(⟨array⟩|⟨bitvec⟩)
              |⟨nid⟩ (⟨input⟩|⟨state⟩)
              |⟨nid⟩ ⟨opidx⟩ ⟨sid⟩ ⟨nid⟩ ⟨uint⟩ [⟨uint⟩]
              |⟨nid⟩ ⟨op⟩ ⟨sid⟩ ⟨nid⟩ [⟨nid⟩ [⟨nid⟩]]
              |⟨nid⟩ (‘init’|‘next’) ⟨sid⟩ ⟨nid⟩ ⟨nid⟩
   ⟨line⟩ ::= ⟨comment⟩|⟨node⟩ [⟨symbol⟩[⟨comment⟩]]
⟨btorsec⟩ ::= (⟨line⟩[newline])+
```

Figure E.1: Syntax of BTOR2

E.2 A JSON Schema for SECSPECS

Recall that SECSPECs are simply JSON files that follow a specific structure. In this section, we precisely outline the structure that well-formatted SECSPECs must follow in the form of a JSON Schema. In a sense, this Schema serves as a meta-specification for a SECSPEC.

```
{
    "$schema": "https://json-schema.org/draft/2020-12/schema",
    "title": "SecSpec",
    "description": "The Schema for a BtorSec Security Specification file",
    "type": "object",

    "properties": {
        "modules": {
            "type": "array",
            "minItems": 1,
            "description": "An array of cryptographic module specifications",
            "items": {
                "oneOf": [
                    {
                        "type": "object",
                        "description": "A top-level module",
                        "properties": {
                            "name": {"type": "string"},
                            "modtype": {"type": "string", "const": "top"},
                            "query": {"type": "array", "items": {"type": "string"}},
                            "public": {"type": "array", "items": {"type": "string"}}
                        },
                        "required": ["name", "modtype", "query", "public"],
                        "additionalProperties": false
                    },
                    {
                        "type": "object",
                        "description": "A key generation module",
                        "properties": {
                            "name": {"type": "string"},
                            "modtype": {"type": "string", "const": "keygen"},
                            "delay": {"type": "integer", "minimum": 0}
                        },
                        "required": ["name", "modtype"],
                        "additionalProperties": false
                    },
                    {
                        "type": "object",
                        "description": "A key derivation module",
                        "properties": {
                            "name": {"type": "string"},
                            "modtype": {"type": "string", "const": "keydrv"},
                            "delay": {"type": "integer", "minimum": 0},
                        "sources": {"type": "array", "minItems": 1, "items": {"type": "string"}}
                    },
                    "required": ["name", "modtype"],
                    "additionalProperties": false
                },
                {
                    "type": "object",
                    "description": "A symmetric encryption module",
                    "properties": {
                        "name": {"type": "string"},
                        "modtype": {"type": "string", "const": "symenc"},
                        "delay": {"type": "integer", "minimum": 0},
                        "plaintext": {"type": "string"},
                        "key": {"type": "string"},
                        "ciphertext": {"type": "string"}
                    },
                    "required": ["name", "modtype", "plaintext", "key", "ciphertext"],
                    "additionalProperties": false
                },
                {
                    "type": "object",
                    "description": "A symmetric decryption module",
                    "properties": {
                        "name": {"type": "string"},
                        "modtype": {"type": "string", "const": "symdec"},
                        "delay": {"type": "integer", "minimum": 0},
                        "ciphertext": {"type": "string"},
                        "key": {"type": "string"},
                        "plaintext": {"type": "string"}
                    },
                    "required": ["name", "modtype", "ciphertext", "key", "plaintext"],
                    "additionalProperties": false
                },
                {
                    "type": "object",
                    "description": "An asymmetric encryption module",
                    "properties": {
                        "name": {"type": "string"},
                        "modtype": {"type": "string", "const": "asymenc"},
                        "delay": {"type": "integer", "minimum": 0},
                        "plaintext": {"type": "string"},
                        "publickey": {"type": "string"},
                        "ciphertext": {"type": "string"}
                    },
                    "required": ["name", "modtype", "plaintext", "publickey", "ciphertext"],
                    "additionalProperties": false

                    },
                    {
                        "type": "object",
                        "description": "An asymmetric decryption module",
                        "properties": {
                            "name": {"type": "string"},
                            "modtype": {"type": "string", "const": "asymdec"},
                            "delay": {"type": "integer", "minimum": 0},
                            "plaintext": {"type": "string"},
                            "privatekey": {"type": "string"},
                            "ciphertext": {"type": "string"}
                        },
                        "required": ["name", "modtype", "plaintext", "privatekey", "
    ↪ ciphertext"],
                        "additionalProperties": false
                    },
                    {
                        "type": "object",
                        "description": "A hashing module",
                        "properties": {
                            "name": {"type": "string"},
                            "modtype": {"type": "string", "const": "hash"},
                            "delay": {"type": "integer", "minimum": 0},
                            "sources": {"type": "array", "minItems": 1, "items": {"type":
    ↪  "string"}},
                            "computedhash" : {"type": "string"}
                        },
                        "required": ["name", "modtype", "computedhash", "sources"],
                        "additionalProperties": false
                    }
                ]
            }
        }
    }
}
```

Bibliography

[1] S. A. Seshia and P. Subramanyan, "Uclid5: Integrating modeling, verification, synthesis and learning," in *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2018, pp. 1–10. DOI: 10.1109/MEMCOD.2018.8556946.

[2] E. Polgreen, K. Cheang, P. Gaddamadugu, *et al.*, *Uclid5: Multi-modal formal modeling, verification, and synthesis*, 2022. arXiv: 2208.03699 [cs.LO]. [Online]. Available: https://arxiv.org/abs/2208.03699.

[3] A. Niemetz, M. Preiner, C. Wolf, and A. Biere, "Btor2 , btormc and boolector 3.0," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds., Cham: Springer International Publishing, 2018, pp. 587–595, ISBN: 978-3-319-96145-3.

[4] M. Shetty, N. Jain, A. Godbole, S. A. Seshia, and K. Sen, *Syzygy: Dual code-test c to (safe) rust translation using llms and dynamic analysis*, 2024. arXiv: 2412.14234 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2412.14234.

[5] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2000.

[6] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, May 2005, ISSN: 0164-0925. DOI: 10.1145/1065887.1065892. [Online]. Available: https://doi.org/10.1145/1065887.1065892.

[7] O. Tripp, M. Pistoia, S. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009, pp. 87–97.

[8] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[9] F. B. Schneider, "Enforceable security policies," in *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, 2000, pp. 30–50.

[10] A. Limited, "Armv8.5-a memory tagging extension (mte)," Arm Ltd., White Paper, 2019, Introduces MTE to help detect spatial and temporal memory-safety violations without requiring source changes.

[11] O. /. S. Microsystems, "Application data integrity (adi) for sparc m7/m8," Oracle / Solaris Documentation, Technical Documentation, 2018, Describes hardware tagging of memory and pointers via version numbers to detect memory corruption in SPARC systems.

[12] R. N. M. Watson, D. Chisnall, J. Clarke, *et al.*, "Cheri: Hardware-enabled c/c++ memory protection at scale," *IEEE Security & Privacy*, vol. 22, no. 4, pp. 50–61, 2024. DOI: 10.1109/MSEC.2024.3396701.

[13] M. LeMay, J. Rakshit, S. Deutsch, *et al.*, "Cryptographic capability computing," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 253–267, ISBN: 9781450385572. DOI: 10.1145/3466752.3480076. [Online]. Available: https://doi.org/10.1145/3466752.3480076.

[14] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[15] Mohamed Tarek Ibn Ziad, "Hardware-software co-design for practical memory safety," Ph.D. dissertation, Columbia University, Apr. 2022, ISBN: 9798426814110. [Online]. Available: https://www.proquest.com/docview/2656610453/fulltextPDF/A52BE1A4F143C2PQ/1?accountid=10226.

[16] M. Kounavis, S. Deutsch, S. Ghosh, and D. Durham, "K-cipher: A low latency, bit length parameterizable cipher," in *2020 IEEE Symposium on Computers and Communications (ISCC)*, 2020, pp. 1–7. DOI: 10.1109/ISCC50000.2020.9219582.

[17] A. S. Ocegueda and A. Godbole. "U2," GitHub. (2025), [Online]. Available: https://github.com/sanchezocegueda/u2 (visited on 08/15/2025).

[18] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, "P: Safe asynchronous event-driven programming," *SIGPLAN Not.*, vol. 48, no. 6, pp. 321–332, Jun. 2013, ISSN: 0362-1340. DOI: 10.1145/2499370.2462184. [Online]. Available: https://doi.org/10.1145/2499370.2462184.

[19] D. Lee, K. Cheang, A. Thomas, *et al.*, "Cerberus: A formal approach to secure and efficient enclave memory sharing," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1871–1885, ISBN: 9781450394505. DOI: 10.1145/3548606.3560595. [Online]. Available: https://doi.org/10.1145/3548606.3560595.

[20] P. Gaddamadugu, "Formally verifying trusted execution environments with uclid5," M.S. thesis, EECS Department, University of California, Berkeley, May 2021. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-200.html.

[21] P. Subramanyan, R. Sinha, I. Lebedev, S. Devadas, and S. A. Seshia, "A formal foundation for secure remote execution of enclaves," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2435–2450, ISBN: 9781450349468. DOI: 10.1145/3133956.3134098. [Online]. Available: https://doi.org/10.1145/3133956.3134098.

[22] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019, pp. 288–28 815. DOI: 10.1109/CSF.2019.00027.

[23] C. Rasmussen, "Secure speculation: From vulnerability to assurances with uclid5," M.S. thesis, EECS Department, University of California, Berkeley, May 2019. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-95.html.

[24] K. Cheang, C. Rasmussen, D. Lee, D. W. Kohlbrenner, K. Asanović, and S. A. Seshia, *Verifying risc-v physical memory protection*, 2022. arXiv: 2211.02179 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2211.02179.

[25] M. Barbosa, G. Barthe, K. Bhargavan, *et al.*, "Sok: Computer-aided cryptography," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008.

[26] D. Basin, C. Cremers, J. Dreier, and R. Sasse, " Tamarin: Verification of Large-Scale, Real-World, Cryptographic Protocols," *IEEE Security & Privacy*, vol. 20, no. 03, pp. 24–32, May 2022, ISSN: 1558-4046. DOI: 10.1109/MSEC.2022.3154689. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSEC.2022.3154689.

[27] C. Cremers, M. Horvat, J. Hoyland, S. Scott, and T. van der Merwe, "A comprehensive symbolic analysis of tls 1.3," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17, Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1773–1788, ISBN: 9781450349468. DOI: 10.1145/3133956.3134063. [Online]. Available: https://doi.org/10.1145/3133956.3134063.

[28] D. Basin, J. Dreier, L. Hirschi, S. Radomirovic, R. Sasse, and V. Stettler, "A formal analysis of 5g authentication," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18, Toronto, Canada: Association for Computing Machinery, 2018, pp. 1383–1396, ISBN: 9781450356930. DOI: 10.1145/3243734.3243846. [Online]. Available: https://doi.org/10.1145/3243734.3243846.

[29] B. Blanchet, “The security protocol verifier proverif and its horn clause resolution algorithm,” *Electronic Proceedings in Theoretical Computer Science*, vol. 373, pp. 14–22, Nov. 2022, ISSN: 2075-2180. DOI: 10.4204/eptcs.373.2. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.373.2.

[30] B. Blanchet and C. Jacomme, “CryptoVerif: a Computationally-Sound Security Protocol Verifier,” Inria, Tech. Rep. RR-9526, Oct. 2023, p. 194. [Online]. Available: https://inria.hal.science/hal-04253820.

[31] M. Abadi, B. Blanchet, and C. Fournet, “The applied pi calculus: Mobile values, new names, and secure communication,” *J. ACM*, vol. 65, no. 1, Oct. 2017, ISSN: 0004-5411. DOI: 10.1145/3127586. [Online]. Available: https://doi.org/10.1145/3127586.

[32] V. Cheval, V. Cortier, and A. Debant, “Election verifiability with ProVerif,” in *Proceedings of the 36th IEEE Computer Security Foundations Symposium (CSF’23)*, Dubrovnik, Croatia: IEEE, Jul. 2023, pp. 43–58.

[33] N. Kobeissi, K. Bhargavan, and B. Blanchet, “Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach,” in *2nd IEEE European Symposium on Security and Privacy (EuroS&P’17)*, Paris, France: IEEE, Apr. 2017, pp. 435–450.

[34] V. Cheval, C. Cremers, A. Dax, L. Hirschi, C. Jacomme, and S. Kremer, “Hash gone bad: Automated discovery of protocol attacks that exploit hash function weaknesses,” in *32nd USENIX Security Symposium (USENIX Security’23)*, Anaheim, CA, USA: USENIX Association, Aug. 2023.

[35] B. Blanchet and C. Jacomme, “Post-quantum sound CryptoVerif and verification of hybrid TLS and SSH key-exchanges,” in *Proceedings of the 37th IEEE Computer Security Foundations Symposium (CSF’24)*, Enschede, The Netherlands: IEEE, Jul. 2024, pp. 543–556.

[36] B. Blanchet, “Dealing with dynamic key compromise in CryptoVerif,” in *Proceedings of the 37th IEEE Computer Security Foundations Symposium (CSF’24)*, Enschede, The Netherlands: IEEE, Jul. 2024, pp. 495–510.

[37] “Ieee standard for verilog hardware description language,” *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006. DOI: 10.1109/IEEESTD.2006.99495.

[38] “Ieee standard for systemverilog–unified hardware design, specification, and verification language,” *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, pp. 1–1354, 2024. DOI: 10.1109/IEEESTD.2024.10458102.

[39] M. J. H. Heule, O. Kullmann, and V. W. Marek, “Solving and verifying the boolean pythagorean triples problem via cube-and-conquer,” in *Theory and Applications of Satisfiability Testing – SAT 2016*, N. Creignou and D. Le Berre, Eds., Cham: Springer International Publishing, 2016, pp. 228–245, ISBN: 978-3-319-40970-2.

[40] A. Biere and D. Kröning, "Sat-based model checking," in *Handbook of Model Checking*, E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, Eds. Cham: Springer International Publishing, 2018, pp. 277–303, ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_10. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_10.

[41] K. L. McMillan, "Interpolation and sat-based model checking," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13, ISBN: 978-3-540-45069-6.

[42] M. J. Heule and O. Kullman, "The science of brute force," *Communications of the ACM*, vol. 60, no. 8, pp. 70–79, Jul. 2017. DOI: 10.1145/3107239. [Online]. Available: https://cacm.acm.org/research/the-science-of-brute-force/.

[43] S. Malik and L. Zhang, "Boolean satisfiability from theoretical hardness to practical success," *Commun. ACM*, vol. 52, no. 8, pp. 76–82, Aug. 2009, ISSN: 0001-0782. DOI: 10.1145/1536616.1536637. [Online]. Available: https://doi.org/10.1145/1536616.1536637.

[44] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., second, vol. 336, IOS Press, 2021, ch. 33, pp. 1267–1329, ISBN: 978-1-64368-161-0.

[45] D. Marker, *Model Theory: An Introduction* (Graduate Texts in Mathematics). Springer, 2002, vol. 217, ISBN: 978-0-387-98760-6. DOI: 10.1007/b98860.

[46] C. Barrett, P. Fontaine, and C. Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, www.SMT-LIB.org, 2016.

[47] L. de Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340, ISBN: 978-3-540-78800-3.

[48] E. M. C. Jr., O. Grumberg, D. Kroening, D. Peled, and H. Veith, *Model Checking*, 2nd ed. Cambridge, MA: MIT Press, 2018, ISBN: 978-0-262-03883-6.

[49] R. E. Bryant, "Symbolic simulation—techniques and applications," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, ser. DAC '90, Orlando, Florida, USA: Association for Computing Machinery, 1991, pp. 517–521, ISBN: 0897913639. DOI: 10.1145/123186.128296. [Online]. Available: https://doi.org/10.1145/123186.128296.

[50] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, "Horn clause solvers for program verification," in *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds. Cham: Springer International Publishing, 2015, pp. 24–51, ISBN: 978-3-319-23534-9. DOI: `10.1007/978-3-319-23534-9_2`. [Online]. Available: `https://doi.org/10.1007/978-3-319-23534-9_2`.

[51] A. Gurfinkel, "Program verification with constrained horn clauses (invited paper)," in *Computer Aided Verification*, S. Shoham and Y. Vizel, Eds., Cham: Springer International Publishing, 2022, pp. 19–29, ISBN: 978-3-031-13185-1.

[52] Y. Team, *Oss cad suite*, 2025. [Online]. Available: `https://github.com/YosysHQ/oss-cad-suite-build`.

[53] A. Dobis, "Formal verification of hardware using mlir," en, Master Thesis, ETH Zurich, Zurich, 2024. DOI: `10.3929/ethz-b-000668906`.

[54] A. S. Ocegueda, A. Godbole, and A. Dobis. "Btorsec," GitHub. (2025), [Online]. Available: `https://github.com/sanchezocegueda/btor2-opt` (visited on 05/11/2025).

[55] D. Dolev and A. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–208, 1983. DOI: `10.1109/TIT.1983.1056650`.

[56] L. Erkök, M. Carlsson, and A. Wick, "Hardware/software co-verification of cryptographic algorithms using cryptol," *2009 Formal Methods in Computer-Aided Design*, pp. 188–191, 2009. [Online]. Available: `https://api.semanticscholar.org/CorpusID:23306298`.

[57] S. Browning, M. Carlsson, L. Erkök, J. Matthews, B. Martin, and S. A. Weaver, "Empowering the experts : High-assurance , high-performance , high-level design with cryptol," 2011. [Online]. Available: `https://api.semanticscholar.org/CorpusID:13259826`.

[58] K. S. Maheswaran, C. Bossut, A. Wanna, Q. Zhang, and C. Hao, *Cryptonite: Scalable accelerator design for cryptographic primitives and algorithms*, 2025. arXiv: `2505.14657 [cs.AR]`. [Online]. Available: `https://arxiv.org/abs/2505.14657`.

[59] S. Dinesh, M. Parthasarathy, and C. W. Fletcher, "Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 3735–3753. DOI: `10.1109/SP54263.2024.00180`.

[60] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-HOUDINI: Scalable invariant learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '25, New York, NY, USA: Association for Computing Machinery, 2025. DOI: `10.1145/3669940.3707263`. [Online]. Available: `https://doi.org/10.1145/3669940.3707263`.

[61] S. Dinesh, G. Garrett-Grossman, and C. W. Fletcher, “Synthct: Towards portable constant-time code,” in *NDSS*, The Internet Society, 2022.

[62] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey, “P: Safe asynchronous event-driven programming,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 321–332, ISBN: 9781450320146. DOI: 10.1145/2491956.2462184. [Online]. Available: https://doi.org/10.1145/2491956.2462184.

[63] S. Kushwah, “Psec: A programming language for secure distributed computing,” M.S. thesis, EECS Department, University of California, Berkeley, May 2020. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-83.html.

[64] S. Kushwah, A. Desai, P. Subramanyan, and S. A. Seshia, “Psec: Programming secure distributed systems using enclaves,” in *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’21, Virtual Event, Hong Kong: Association for Computing Machinery, 2021, pp. 802–816, ISBN: 9781450382878. DOI: 10.1145/3433210.3453113. [Online]. Available: https://doi.org/10.1145/3433210.3453113.