Chronomorphic Programs: Runtime Diversity Prevents Exploits and Reconnaissance

Scott E. Friedman, David J. Musliner, and Peter K. Keller
Smart Information Flow Technologies (SIFT)
Minneapolis, USA
email: {sfriedman,dmusliner,pkeller}@sift.net

***Abstract*—In Return Oriented Programming (ROP) attacks, a cyber attacker crafts an exploit from instruction sequences already contained in a running binary. ROP attacks are now used widely, bypassing many cyber defense mechanisms. While previous research has investigated software diversity and dynamic binary instrumentation for defending against ROP, many of these approaches incur large performance costs or are susceptible to Blind ROP attacks. We present a new approach that automatically rewrites potentially-vulnerable software binaries into *chronomorphic* binaries that change their in-memory instructions and layout repeatedly, at runtime. We describe our proof of concept implementation of this approach, discuss its security and safety properties, provide statistical analyses of runtime diversity and reduced ROP attack likelihood, and present empirical results that demonstrate the low performance overhead of actual chronomorphic binaries.**

Keywords-cyber defense; software diversity; self-modifying code.

I. Introduction

In the old days, cyber attackers only needed to find a buffer overflow or other vulnerability and use it to upload their exploit instructions, then make the program execute those new instructions. To counter this broad vulnerability, modern operating systems enforce "write XOR execute" (W⊕X) defenses: that is, memory is marked as either writable or executable, but not both. So exploit code that is uploaded to writable memory cannot be executed. Not surprisingly, attackers then developed a more sophisticated exploit method.

Computer instruction sets are densely packed into a small number of bits, so accessing those bits in ways that a programmer did not originally intend can yield *gadgets*: groups of bits that form valid instructions that can be strung together by an attacker to execute arbitrary attack code from an otherwise harmless program. Known as *Return Oriented Programming* (ROP), these types of cyber exploits have been effective and commonplace since the widespread deployment of W⊕X defenses. This paper extends our previous research on runtime program security to prevent ROP exploits [1].

Software with a single small buffer-overflow vulnerability can be hijacked into performing arbitrary computations using ROP-like code-reuse attacks [2], [3]. Hackers have even developed *ROP compilers* that build the ROP exploits automatically, finding gadgets in the binary of a vulnerable target and stringing those gadgets together to implement the attacker's code [4], [5]. And to counteract various software diversity defenses that try to move the gadgets around, so that a previously-compiled ROP attack will fail, attackers have developed *Blind ROP* (BROP) attacks that perform automated reconnaissance to find the gadgets in a running program [6].

This paper presents a fully automated approach for transforming binaries into *chronomorphic* binaries that diversify themselves during runtime, throughout their execution, to offer strong statistical defenses against code reuse exploits such as ROP and BROP attacks. The idea is to modify the binary so that all of the potentially-dangerous gadgets are repeatedly changing or moving, so that even a BROP attack tool cannot accumulate enough information about the program's memory layout to succeed.

In the following sections, we discuss related research in this area (Section II), we outline the threat of ROP exploits (Section III), we describe how our prototype Chronomorph tool converts regular binaries into chronomorphic binaries (Section IV), and we review its present limitations. We then describe an analysis of the safety and security of the resulting chronomorphic binaries, and performance results on early examples (Section V). We conclude with several directions for future work, to harden the tool and broaden its applicability (Section VI).

II. Related Work

Various defense methods have been developed to try to foil code reuse exploits such as ROP and BROP. Some of defenses instrument binaries to change their execution semantics [7] or automatically filter program input to prevent exploits [8]; however, these approaches require process-level virtual machines or active monitoring by other processes. Other approaches separate and protect exploitable data (e.g., using shadow stacks [9]), but such approaches incur comparatively high overhead.

To reduce overhead and maintain compatibility with existing operating systems and software architectures, many researchers have focused on lightweight, diversity-based techniques to prevent code reuse exploits. For example, Address Space Layout Randomization (ASLR) is common in modern operating systems, and loads program modules into different locations each time the software is started. However, ASLR does not randomize the location of the instructions *within* loaded modules, so programs are still vulnerable to ROP attacks [10]. Some diversity techniques modify the binaries themselves to make them less predictable by an attacker. For example:

- Compile-time diversity (e.g., [11]) produces semantically equivalent binaries with different structures.

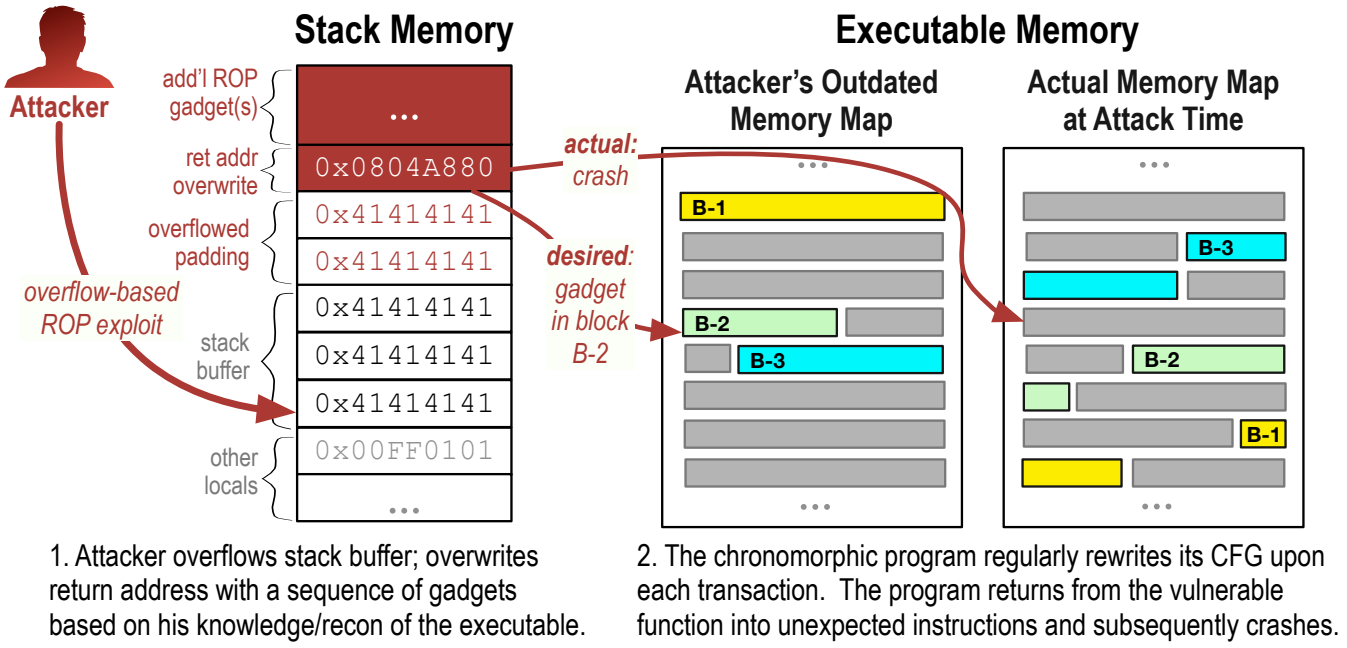

Figure 1. A traditional ROP attack– or blind ROP attack– is thwarted by the runtime diversity of a chronomorphic program.

- Offline code randomization (e.g., [12]) transforms a binary on disk into a functionally equivalent variant with different bytes loaded into memory.
- Load-time code randomization (e.g., [13], [14]) makes the binary load blocks of instructions at randomized addresses.

These diversity-based approaches incur comparatively lower overhead than other ROP defenses and they offer statistical guarantees against ROP attacks.

Unfortunately, these compile-time, offline, and load-time diversity defenses are still susceptible to BROP attacks that perform runtime reconnaissance to map the binary and find gadgets [6]. So, even with compile-time, offline, or load-time diversity, software that runs for a significant period of time without being reloaded (e.g., all modern server architectures) is vulnerable. Some ROP defenses modify the operating system to augment diversity [15], [16], but by nature they do not work on existing operating systems.

Another recent approach uses compile-time diversity in tandem with hardware-based enforcement mechanisms that prevents adversaries from reading any code [17]. This protects against memory disclosure— and thereby prevents ROP and BROP attacks— but like the above techniques, this requires modifying the underlying operating system or hardware.

Some runtime techniques clone executable elements in memory and toggle between them during runtime, so the attacker is unaware of which memory layout is executing; however, the diversity factor is not as high as the above approaches. Another recent approach combines execution-time switching with runtime diversification [18] by instrumenting all call, return, and jump instructions. On these instrumented instructions, execution may randomly switch between executable copies while the other copy is diversified by fine-grained ASLR. While this approach prevents varieties of ROP attacks, it incurs significant runtime overhead due to a dynamic binary instrumentation framework, and it doubles the size of the binary due to executable memory cloning.

Other tools such as DynInst[1] rewrite the binary to add runtime libraries and instrumentation, but this instrumentation consumes substantially higher disk space, memory footprint, or performance overhead.

Unlike the above diversity-based protection techniques, chronomorphic programs only utilize a single instance of the program in memory at any time, making them more suitable for embedded or memory-constrained systems. Chronomorphic programs will diversify themselves *throughout program execution* to statistically prevent code reuse attacks, even if the attacker knows the memory layout. The only runtime costs are incurred when actually morphing the program; when not morphing, the program executes (almost) its original instructions. Furthermore, the costs of the morphing behavior can be adjusted and controlled to achieve desired performance levels in the face of changing threat levels, rates of adversary-provided input, etc. Unlike other approaches, our prototyped chronomorphic programs run on existing hardware and operating systems, making them suitable for legacy systems.

III. Threat Model

Here we describe the setting for chronomorphic programs— including assumptions about the target program, the target system, and the attacker— and we illustrate this setting in

[1] http://www.dyninst.org/

Figure 1. The chronomorphic defense against code reuse attacks assumes the following of the targeted system and its adversaries:

- The target program has a vulnerability that permits the adversary to hijack the program's control flow.
- The target host uses write-xor-execute (W⊕X) permissions on its memory pages.
- The target operating system contains standard functions to modify W⊕X memory permissions (e.g., `mprotect` in Linux and `VirtualProtect` in Windows), which we describe later.
- The adversary cannot influence the offline operations that create the chronomorphic binary and its morph table from a standard executable (see Figure 2).
- The adversary may have access to the target program's source code and to the original (non-chronomorphic) off-the-shelf executable.
- The adversary may have *a priori* knowledge of the program's in-memory code layout at *any* point in execution, unlike other ROP defenses (e.g., [11], [12], [13], [14], [17]), due to cyber reconaissance (e.g., [6]) or *runtime information leaks* whereby timing data, cache data, and side channel data is available to the attacker.
- The adversary must interact with the target program (e.g., execute its in-memory code) in order to observe its in-memory code layout or conduct an exploit.

Under these assumptions, the adversary may successfully exploit the target program over multiple transactions (e.g., server requests), *provided the target program does not change its exploitable memory layout before or after those transactions*.

Chronomorphic programs foil cyber-attacks that rely on consistency of a program's memory layout— including code reuse attacks like ROP— since the memory layout changes during execution. As described below, these memory changes should occur early in the processing of each transaction. Figure 1 illustrates an attempt of a classic stack-overflow-based ROP exploit on a chronomorphic executable. The attacker overflows a stack buffer to overwrite the return address with one or more gadget addresses that he may have learned by analyzing the executable on disk or by exploiting the runtime information leaks mentioned above.

The chronomorphic program rewrites itself regularly throughout execution (Figure 1, right), so the attacker's knowledge of the program is outdated. When the program returns into the chain of gadgets written by the attacker, the program executes different instructions than those intended by the attacker. The program promptly crashes without executing the exploit.

The empty, unlabeled area (i.e., block relocation space) could be comprised of invalid, unexploitable instructions that quickly cause a crash, or alternatively, nop-slides into an invocation of alarms or forensic analysis functions that terminate with an error signal, consistent with other software "booby trapping" approaches [19].

Chronomorphic programs diversify themselves based on random selections from their *morph tables*, which describe how the program can be changed during execution. This means that if the morph table is aware to the attacker *and* the target host's `random` operation is perfectly predictable, then the attacker can predict the next configuration assumed by the chronomorphic program and thereby conduct a successful code reuse exploit. Consequently, even if the morph table is acquired and decrypted by the adversary, they must perfectly predict— or somehow influence— the host's randomization, which would already constitute a deep intrusion of the target system.

We next describe how we create chronomorphic programs from off-the-shelf executables, and how these programs diversify themselves throughout the course of their execution.

IV. Approach

The Chronomorph approach requires changing machine code at runtime, a technique known as *self-modifying code* (SMC). Using SMC, Chronomorph must preserve the functionality of the underlying program (i.e., maintain semantics), maximize diversity over time, and minimize performance costs.

Any SMC methodology requires a means to change the permissions of the program's memory (i.e., temporarily circumvent W⊕X defense) to modify the code and then resume its execution. Different operating systems utilize different memory protection functions: Linux's `mprotect` and Windows' `VirtualProtect` have different signatures, but both can temporarily change program memory permissions from executable to writable, and back again, during program execution. In this paper, we describe Chronomorph in a 32-bit Linux x86 setting.

Our approach automatically constructs chronomorphic binaries from normal third-party programs with the following enumerated steps, also illustrated in Figure 2:

Offline:

1) Transform the executable to inject the Chronomorph SMC runtime that invokes `mprotect` and rewrites portions of the binary during execution. This produces a *SMC binary* with SMC functions that are disconnected from the program's normal control flow.
2) Analyze the SMC binary to identify potentially-exploitable sequences of instructions (i.e., gadgets).
3) Identify relocatable gadgets and transform the SMC binary to make those gadgets relocatable.
4) Compute instruction-level, semantics-preserving transforms that denature non-relocatable gadgets and surrounding program code.
5) Write the relocations and transforms to a *morph table* outside the chronomorphic binary.
6) Inject *morph triggers* into the SMC binary so that the program will morph itself periodically. This produces the *chronomorphic binary*.

Online:

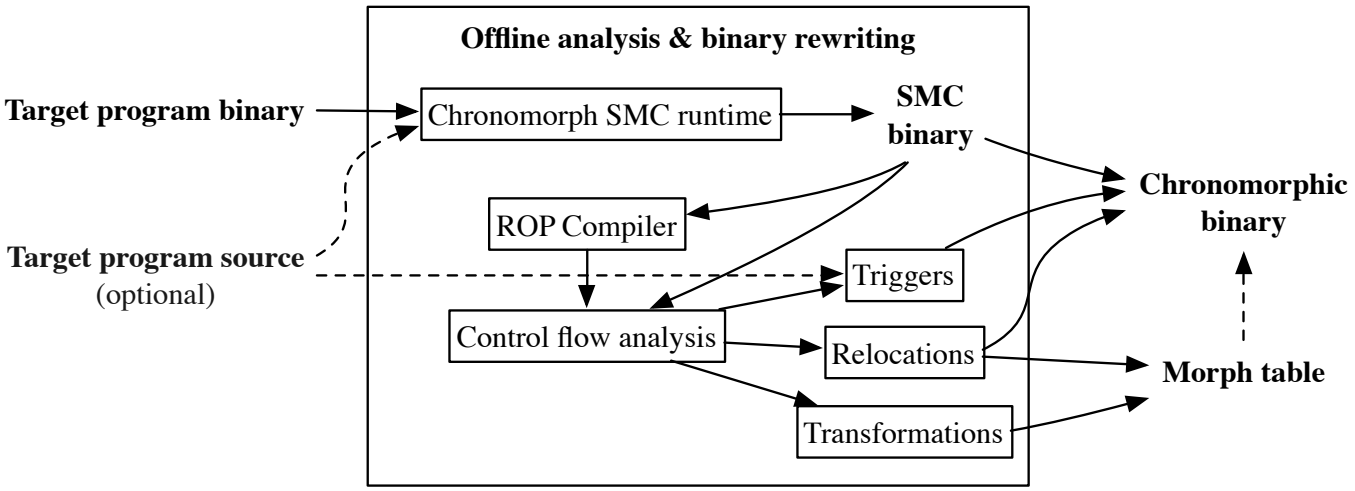

Figure 2. Chronomorph converts a third-party program into a chronomorphic binary.

7) During program runtime, diversify the chronomorphic binary's executable memory space by relocating and transforming instructions without hindering performance or functionality.

We have implemented each step in this process and integrated third-party tools including a ROP compiler [5], the Hydan tool for computing instruction-level transforms [20], and the open-source `objdump` disassembler. We next describe each of these steps in this process, including the research challenges and the strategy we employ in our Chronomorph prototype implementation. We note relevant simplifying assumptions in our prototype, and we address some remaining research challenges in Section VI.

A. Injecting SMC morphing functionality

Before the Chronomorph tool can analyze the binary and compute transformations, it must inject the Chronomorph SMC runtime, which contains functions for modifying memory protection (e.g., `mprotect`), writing byte sequences to specified addresses, and reading the morph table from outside the binary. These Chronomorph functions may *themselves* contain gadgets and have runtime diversification potential, so the SMC-capable binary that includes these functions is the input to the subsequent offline analyses, including ROP compilation.

We identified three ways of automatically injecting the Chronomorph runtime code, based on the format of the target program.

1) Link the target program's source code against the compiled Chronomorph runtime. This produces a dynamically- or statically-linked SMC executable. This is the simplest solution, and the one used in our experiments, but source code may not always be available.
2) Rewrite a statically-linked binary by extending its binary with a new loadable, executable segment containing the statically-linked Chronomorph runtime. This produces a statically-linked SMC executable.
3) Rewrite a dynamically-linked binary by adding Chronomorph procedures and objects to an alternative procedure linkage table (PLT) and global object table (GOT), respectively, and then extend the binary with a new loadable, executable segment containing the dynamically-linked Chronomorph runtime. This produces a dynamically-linked SMC executable.

All three of these approaches inject the self-modifying Chronomorph runtime, producing the *SMC binary* shown in Figure 2. At this point, the self-modification functions are not yet invoked from within the program's normal control flow, so we cannot yet call this a chronomorphic binary.

B. Identifying exploitable gadgets

As shown in Figure 2, the Chronomorph offline analysis tool includes a third-party ROP compiler [5] that automatically identifies available gadgets within a given binary and creates an exploit of the user's choice (e.g., execute an arbitrary shell command) by compiling a sequence of *attack gadgets* from the available gadgets, if possible. The Chronomorph analysis tool runs the ROP compiler against the SMC binary, finding gadgets that span the entire executable segment, including the Chronomorph SMC runtime.

The ROP compiler prioritizes Chronomorph's diversification efforts as follows, to allocate time and computing resources proportional to the various exploitation threats within the binary:

- Attack gadgets have the highest priority. The chronomorphic binary should address these with its highest-diversity transforms.
- Available gadgets (i.e., found by the ROP compiler but not present in an attack sequence) have medium priority. These too should be addressed by high-diversity transforms, within acceptable performance bounds.
- Instructions that have not been linked to an available gadget have the lowest priority, but should still be diversified

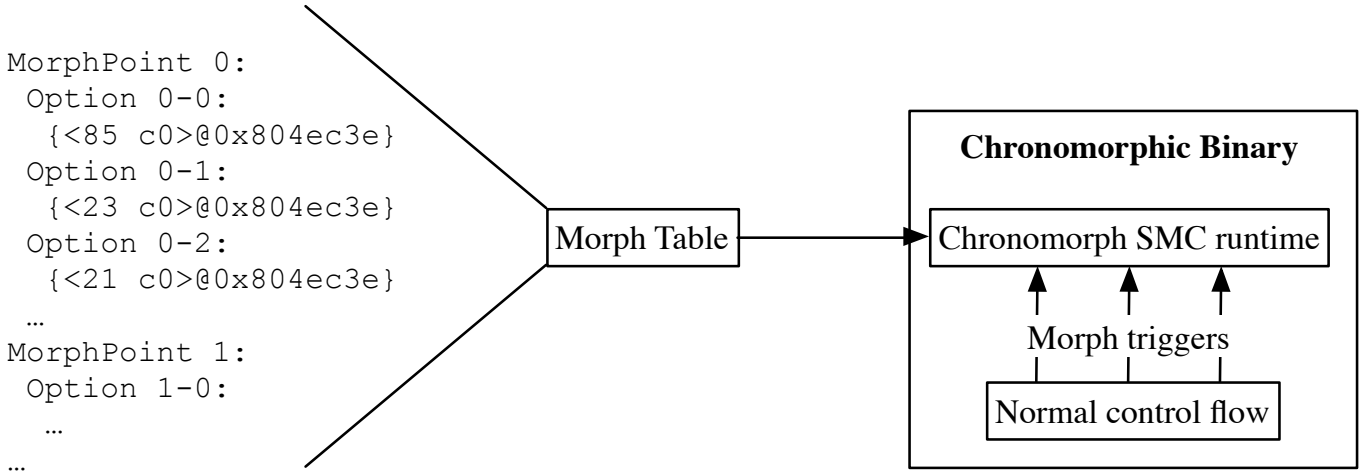

Figure 3. The resulting chronomorphic binary and its interaction with the morph table.

when feasible. Since zero-day gadgets and new code-reuse attack strategies may arise after transformation time, this diversification offers additional security.

Our approach attempts all transformations possible, saving more costly transformations, e.g., dynamic block relocations, for the high-risk attack gadgets. The ROPgadget compiler [5] currently used by Chronomorph may be easily replaced by newer, broader ROP compilers, provided the compiler still compiles attacks and reports all available gadgets. Also, a portfolio approach may be used, running a variety of ROP compilers and merging their lists of dangerous gadgets.

C. Diversity with relocation

We may not be able to remove a high-risk gadget entirely from the executable, since its instructions may be integral to the program's execution; however, the chronomorphic binary can relocate it with high frequency throughout execution, as long as it preserves the control flow.

Relocation is the highest-diversity strategy that Chronomorph offers. Chronomorph allocates an empty *block relocation space* in the binary, reserved for gadget relocation. Whenever the chronomorphic binary triggers a morph, it shuffles relocated blocks to random locations in the block relocation space and repairs previous control flow with recomputed `jmp` instructions to the corresponding location in the block relocation space.

For each high-risk attack gadget, Chronomorph performs the following steps to make it relocatable during runtime:

1) Compute the *basic block* (i.e., sequence of instructions with exactly one entry and exit point) that contains the gadget.
2) Relocate the byte sequence of the gadget's basic block to the first empty area in the block relocation space.
3) Write a `jmp` instruction from the head of the basic block to the new address in the block relocation space.
4) Write `nop` instructions over the remainder of the gadget's previous basic block, destroying the gadgets.
5) Write the block's byte sequence and the address of the new `jmp` instruction to the morph table.

The morph table now contains enough information to place the gadget-laden block anywhere in the block relocation space and recompute the corresponding `jmp` instruction accordingly.

Intuitively, diversity of the binary increases with the size of the block relocation space. For a single gadget block g with byte-size $|g|$, and block relocation space of size $|b|$, relocating g adds $V(g,b) = 1 + |b| - |g|$ additional program variants.

If we relocate multiple gadget blocks $G = \{g_0, ..., g_{|G|-1}\}$, then we add the following number of variants:

$$V(G,b) = |G| + \prod_{i=0}^{|G|-1} (|b| - \sum_{j=0}^{i} |g_j|). \quad (1)$$

The probability of guessing all of the relocated gadgets' addresses is therefore $1/V(G,b)$, which diminishes quickly as the chosen size of the block relocation space increases.

Our Chronomorph prototype has the following constraints for choosing gadget blocks for relocation:

- Relocated blocks cannot contain a `call` instruction. When a `call` instruction is executed, the subsequent instruction's address is pushed onto the stack, and if the calling block is then relocated, execution would return into an arbitrary (incorrect) spot in the block relocation space.
- Relocated blocks must be at least the size of the `jmp` to the block relocation space, so that Chronomorph has room to write the `jmp`.
- Relocated blocks must end in an indirect control flow (e.g., `ret`) instruction; otherwise, we would have to recompute the control flow instruction at the block's tail at every relocation. Empirically, the vast majority of these blocks end in `ret`.
- Relocated gadgets cannot span two blocks.

In the conclusion of this paper we discuss some potential improvements that would remove some of these constraints.

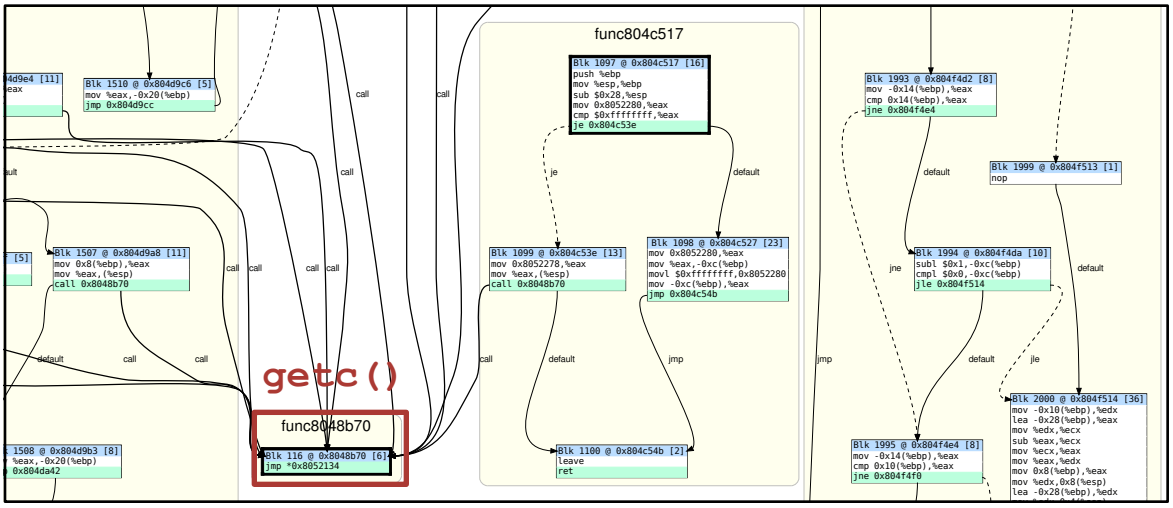

Figure 4. Small portion of a control flow graph (CFG) automatically created during Chronomorph's offline analysis. Shaded regions are functions, instruction listings are basic blocks, and edges are control flow edges.

D. Diversity with in-place code randomizations

Chronomorph uses *in-place code randomization* (IPCR) strategies to randomize non-relocated instructions [12]. IPCR performs narrow-scope transformations without changing the byte-length of instruction sequences.

At present, Chronomorph uses two IPCR strategies to compute transformations. The first, *instruction substitution* (IS), substitutes a single instruction for one or more alternatives. For example, comparisons can be performed in either order, `xor`'ing a register with itself is equivalent to `mov`'ing or `and`'ing zero, etc. These instructions have the same execution semantics, but they change the byte content of the instruction, so unintended control flow instructions (e.g., `0xC3` = `ret`) are potentially transformed or eliminated. A single IS adds as many program variants as there are instruction alternatives.

Another IPCR strategy, *register preservation code reordering* (RPCR) reorders the `pop` instructions before every `ret` instruction of a function, and also reorders the corresponding `push` instructions at the function head to maintain symmetry. A register preservation code reordering for a single function adds as many variants as there are permutations of `push` or `pop` instructions.

Importantly, RPCR changes the layout of a function's stack frame, which may render it non-continuable. For instance, if control flow enters the function and it preserves register values via `push`'ing, and then the chronomorphic binary runs RPCR on the function, it will likely `pop` values into unintended registers when it continues the function, which will adversely affect program functionality.

Any stack-frame diversity method such as RPCR should only be attempted at runtime if execution cannot *continue* or *re-enter* the function, e.g., from an internal `call`, after a SMC morph operation. We enforce this analytically with control flow graph (CFG) analysis: if execution can continue within a function f from the morph trigger (i.e., if the morph trigger is *reachable* from f in the CFG), the stack frame of f should not be diversified. Stack frame diversification is a valuable tool for ROP defense, but it requires these special considerations when invoked during program execution.

E. Writing and reading the morph data

The morph table is a compact binary file that accompanies the chronomorphic binary, as shown in Figure 3. The morph table binary represents packed structs: `MorphPoint` structs with internal `MorphOption` byte sequences. Each `MorphPoint` represents a decision point (i.e., an IS or RPCR opportunity) where any of the associated `MorphOption` structs will suffice. Each `MorphPoint` is stateless (i.e., does not depend on the last choice made for the `MorphPoint`), and independent of any other `MorphPoint`, so random choices are safe and ordering of the morph table is not important.

The relocation data is a separate portion of the morph table, containing the content of relocatable blocks alongside their corresponding `jmp` addresses. Like IPCR operations, relocations are stateless and independent, provided the Chronomorph runtime does not overlay them in the block relocation space.

Intuitively, the morph table cannot reside statically inside the binary as executable code, otherwise all of the gadgets would be accessible.

F. Injecting morph triggers

We have described how Chronomorph injects SMC capabilities into third-party executables and its diversification capabilities, but Chronomorph must also automatically connect the Chronomorph runtime into the program's control flow to induce diversification of executable memory during runtime.

The injection of these *morph triggers* presents a trade-off: morphing too frequently will unnecessarily degrade program performance; morphing too seldom will allow wide windows of attack. Ideally, morphing will happen at the speed of input,

e.g., once per server request or user input (or some modulo thereof). The location of the morph trigger(s) in the program's control flow ultimately determines morph frequency.

Figure 4 shows a portion of the CFG for the program used in our experiment, calling out the `getc()` input function. Chronomorph can inject calls to the SMC runtime at these input points, or at calling functions with stack-based buffers (which are more likely to contain vulnerabilities through which a ROP attack would begin).

Chronomorph also includes an interface for the application developer to add a specialized `MORPH` comment in the source code, which is replaced by a morph trigger during the rewriting phase.

G. Runtime diversification

A chronomorphic binary executes in the same manner as its former non-chronomorphic variant, except when the morph triggers are invoked.

When the first morph trigger is invoked, the Chronomorph runtime loads the morph table and seeds its random number generator. All morph triggers induce a complete SMC diversification of the in-process executable memory according to IPCR and relocation data in the morph table:

1) The block relocation space is made writable with `mprotect`.
2) Relocated basic blocks in the block relocation space are overwritten with `nop` instructions.
3) Each relocatable block is inserted to a random block relocation space address, and its `jmp` instruction (where the block used to be in the program's original CFG) is rewritten accordingly.
4) The block relocation space is made executable.
5) Each `MorphPoint` is traversed, and a corresponding `MorphOption` is chosen at random and written. Each operation is surrounded by `mprotect` calls to make the corresponding page writable and then executable. Future work will group `MorphPoints` by their address to reduce `mprotect` invocations, but our results demonstrate that the existing performance is acceptable.

We have described how to create chronomorphic binaries from off-the-shelf binaries, and we have described how chronomorphic binaries perform runtime diversification. Next we discuss important safety and correctness considerations for chronomorphic programs, since runtime rewriting has implications for concurrency, continuation, and reentrancy.

H. Multithreading

In a traditional multi-threaded setting, multiple threads use the same executable memory. This means that one thread could diversify the function, block, or even the instruction that another thread is executing.

Without additional protections such as thread synchronization, many runtime diversification strategies are unsafe in a multithreaded setting:

- Block relocation is unsafe if another thread is executing a relocatable block.
- Reordering a function's `push` and `pop` register preservation instructions is unsafe if another thread is executing the function, since the thread may `pop` values into the wrong registers.
- Reordering instructions *within* a block (e.g., [12]) is unsafe if another thread is executing the block.

Injecting thread synchronization around diversifiable regions in the chronomorphic program's morph table would prevent these unsafe operations, but this might be costly, since— as we demonstrate in our experiments— there are many diversifiable regions in even the smallest binaries.

For the above reasons, our prototype tool only supports creating chronomorphic binaries for single-threaded execution, but we discuss some possible avenues for multi-threaded chronomorphic programs in our discussion of future work.

I. Guaranteeing safe continuation

Runtime diversity can invalidate stack frame integrity and return address correctness if not properly constrained. These are important considerations for guaranteeing *function continuation* (i.e., resuming execution of a function after returning from another function) within a chronomorphic program. We discuss two continuation pitfalls and our approach for avoiding them.

Reordering `push` and `pop` instructions, are effective methods of foiling ROP attacks [12]; however, performing these operations at runtime will complicate continuation by changing the expected structure of the stack frame: if execution continues in (i.e., returns back into) a function whose `push` and `pop` differ from when execution initially entered, the program will pop the wrong data into the registers, causing a random fault in subsequent execution.

Another consideration is return address correctness. When a program executes a `call` instruction, it will write the address following the `call` onto the stack as the return address, and then transfer control flow to the called function. Suppose that after invoking a `call` instruction and writing the return address, the program triggers a downstream diversification that relocates this upstream `call` instruction. This changes the address that the program *should* return to; however, the old address is still written to the stack, and the program will ultimately return into an undesired location, causing a random fault.

Both of these continuation problems stem from changing executable memory in a way that is inconsistent with presently-written stack memory. There are three general strategies for preserving continuation in light of both of these problems:

1) *Avoidance*: Do not perform `push` and `pop` register preservation reordering, and do not relocate any block containing a `call` instruction.
2) *Rectification*: Rectify stack memory at diversification time by transposing register preservation data (to allow `push` and `pop` reordering) and rewriting return addresses (to allow `call` relocation) of continuable functions.

3) *Reachability*: Do not modify the stack frame (e.g., `push` and `pop` register preservation) or return addresses (e.g., location of `call` instructions) of functions that can plausibly be continued after runtime diversification, using a CFG graph-reachability criterion.

All of these strategies incur a cost. The avoidance strategy reduces diversity by disallowing `push` and `pop` register preservation and disallowing `call` instructions to be relocated. Other strategies such as instruction substitution and the relocation of other non-`call` blocks are still plausible, but this is an unnecessary loss.

The rectification strategy will incur memory overhead to store the data necessary to identify register preservation values and return addresses that have been pushed onto the stack. The time necessary to rectify the stack would be proportional to the depth of the stack, and it could take arbitrarily long to rectify all of the values, e.g., when modifying deep recursive functions.

The reachability strategy reduces diversity potential, but provides more diversity potential than the avoidance strategy. This approach also incurs substantially less overhead than the rectification strategy and permits more diversity than the avoidance strategy, so this is our preferred strategy for chronomorphic programs.

Our prototype tool implements the reachability strategy using a graph-theoretic reachability criteria in the program's CFG. From the CFG, we can infer the set of functions that could reach the runtime diversification function injected into the chronomorphic binary. Our approach recovers the CFG from the binary automatically, using mixed recursive/linear disassembly of the binary, static identification of jump tables, and dynamic tracing to identify indirect control flow (e.g., jump addresses stored as data) [21]. This over-approximates the CFG and ultimately over-approximates the set of functions that *will* be continued after runtime diversification. In a hypothetical worst case, where *every* function in the program can reach the runtime diversification function, the reachability strategy is equivalent to the avoidance strategy described above, which is still preferable to the rectification strategy.

We conducted an experiment with our Chronomorph prototype on a third-party Linux binary to characterize the diversity, ROP attack likelihood, and performance overhead of our Chronomorph approach. We discuss this experiment and its results in the next section.

V. EVALUATION

We tested our prototype tool on small Linux desktop applications, into which we deliberately injected vulnerabilities and gadgets (in the source code). Here, we discuss results for the `dc` (desktop calculator) program.

The original target program, with injected flaws, is easily compromised by our ROP compiler. We also inserted the special `MORPH` comment in the source code, to trigger morphing after each input line was read. After running the prototype Chronomorph system, the new binary operates as described in Figure 3, and cannot be defeated by the ROP compiler. The dynamically-linked version of the original binary is small (47KB), and after our tool has made it chronomorphic (with a block relocation space of 4KB) it is 62KB.

The rewritten binary is currently able to perform approximately 1000 changes to its own code in less than one millisecond, on a standard laptop. When the chronomorphic binary is not rewriting itself, it incurs no additional performance overhead, so the overhead is strictly the product of the time for a complete morph (e.g., one millisecond) and the frequency of morphs, as determined by the injected morph triggers. In our experiments with `dc` performing a short regression test, the chronomorphic version incurred an additional 2% overhead. However, this was an unoptimized version that reloaded the morph table on every morph trigger. For other binaries, overhead will depend heavily on morph trigger placement. Also, a more compute-intensive application might suffer a mild degradation due to cache-misses and branch prediction failures that might not occur in the non-morphing version.

Figure 5 shows two bitmaps illustrating how the binary instructions change in memory, as the program runs. Each pixel of each image represents a single byte of the program's executable code segment in memory. At the top of both images, the gray area is the `nop`-filled block relocation space, with colored segments representing the blocks moved there. Note that the colored segments are in different locations in the two images. Below the gray area, the original binary bits that are never changed remain black, while instructions that are rewritten are shown in different colors, where the red/green/blue values are computed from the byte values and `nop` instructions are gray. Comparison of the images will show that many of the colored areas are different between the images— this clearly shows the broad in-memory diversity induced by the chronomorphic behavior.

We assessed this example's morph table and estimate that it is capable of randomly assuming any one of approximately 10^{500} variants at any given time during execution. The chronomorphic version of the statically-linked target binary ($>$ 500KB) can assume any one of approximately 10^{8000} variants, using about 8ms to perform all of its rewrites. However, those variant counts do not really accurately characterize the probability that a ROP or BROP attack will succeed.

To do that, we must consider how many gadgets the attacker would need to locate, and how they are morphing. The dynamically linked target contains 250 indirect control flow instructions, and two thirds of those potentially risky elements are moved by the block-relocation phase. With the ROPgadget compiler we used for this evaluation, the original application yielded an exploit needing eight gadgets, of which six were subjected to morphing:

- `inc eax ; ret` – relocated.
- `int 0x80` – relocated.
- `pop edx ; ret` – relocated.
- `pop edx ; pop ecx ; pop ebx ; ret` – reordered (6 permutations).
- `pop ebx ; ret` – relocated.
- `xor eax,eax ; ret` – intact.

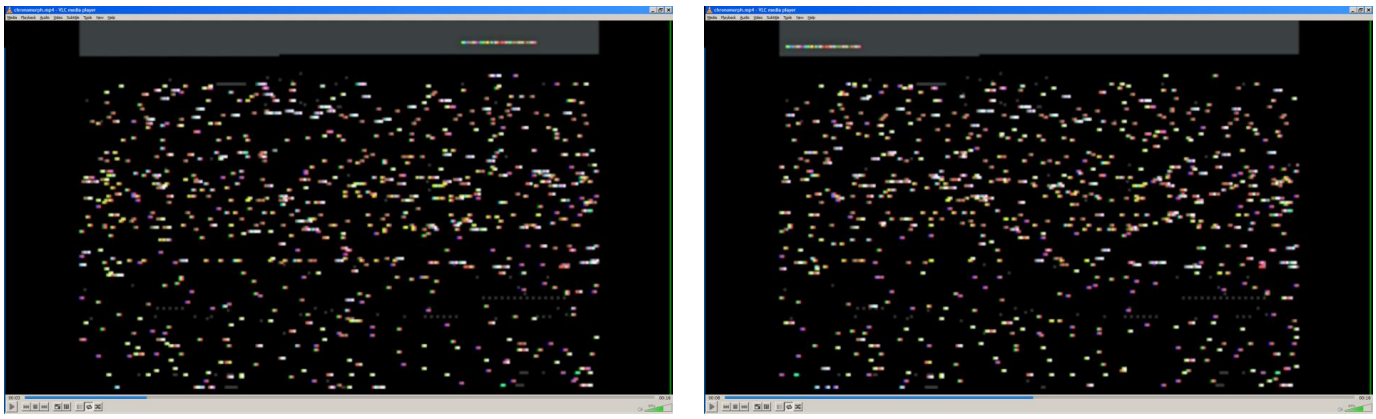

Figure 5. Example memory visualizations illustrating how the executable memory space of the binary changes at runtime.

- `pop eax ; ret` – relocated.
- `move [edx],eax ; ret` – intact.

Five of the gadgets are relocated dynamically within the block relocation space of size $|b|$, and a sixth gadget is rewritten to one of six permutations. As a result, to accurately locate all eight of those gadgets in the chronomorphed binary, a potential ROP attacker would have to pick correctly from approximately $6*|b|^5$ alternatives. For our $|b| = 4$KB example, the probability of a correct guess is approximately $1/10^{18}$, which is extremely unlikely. Needless to say, the ROPgadget exploit was unable to compromise the chronomorphic binary, in thousands of tests. Furthermore, a BROP attack will have no ability to accumulate information about gadget locations, because they change every time a new input is received.

VI. Conclusion and Future Work

We have implemented an initial version of an automatic Chronomorph tool and demonstrated that the resulting chronomorphic binaries are resistant to ROP and BROP attacks and retain their initial functionality. Our automatically-generated chronomorphic binary incurred no runtime overhead during normal operation, and only incurred one millisecond overhead to perform over 1000 sequential rewrites to executable memory during a morph operation.

Our initial version and experimental results demonstrate that chronomorphic programs are feasible: runtime diversity yields real security benefits on existing software running on existing operating systems and hardware. For chronomorphic programs to become widespread and practical, further research must characterize the effect on chronomorphic programs' error reporting capabilities, performance tuning, and reliable binary disassembly for safe code relocation. We prototyped our approach in a 32-bit x86 Linux setting, and scaling to x86_64 would increase the size of the morph table (due to 64-bit addresses), require a x86_64 ROP compiler, and require support for disassembly and binary rewriting to account for x86_64 argument-passing via registers.

Additional high-level research challenges remain for safety and scalability. The system call that allows the chronomorphing code to rewrite executable code is, of course, a dangerous call; if an attacker could locate it and exploit it, he could rewrite the code to do whatever he wants. Therefore, we would ideally like the rewriting/SMC code itself to relocate or transform at runtime; however, the code cannot rewrite itself. We can work around this limitation with a fairly simple trick: we can use two copies of the critical code to alternately rewrite or relocate each other throughout runtime.

Another security concern requiring additional research is ROP attacks *tailored* to chronomorphic programs. If the attacker has full reconnaissance to the program in memory, he can identify which blocks the program has replaced with a dynamic `jmp` instruction that points to the block's new (changing) address. This means that the ROP payload can (1) load the target address of one of said `jmp` instructions, (2) use the target address as an offset for the desired attack gadet, and (3) compute the location of the desired attack gadget. However, this assumes that (1) the attacker already access to gadgets that can load these addresses and perform the required arithmetic, and that (2) all of the desired gadgets are protected by relocation and not by the other code randomization techniques used in our prototype. Alternatively, chronomorphic binaries could compute their jumps to relocated blocks using control-flow-sensitive values, so that relocated block addresses cannot be trivially looked up. The performance overhead of this approach might be reasonable, based on the density of relocated blocks, but this is an empirical question. Even in light of chronomorph-tailored ROP attacks, chronomorphic programs offer significantly more protection than the other techniques reviewed in Section II in a fully-observable setting on legacy hardware and software.

We do not presently protect the morph table, which resides outside of the binary. While chronomorphic binaries do not rely on obscurity for security, an attacker's chances of success would be higher if he has access to the morph table describing how the binary can change itself. Straightforward encryption techniques should allow us to protect the morph table.

We can potentially support multi-threading— and also ensure safe function continuation— by automatically injecting control flow monitors into diversifiable functions. Control

flow monitors, similar to those used for runtime-injected aspect-oriented programming [22], can be certified by well-established model-checking techniques [23]. These control flow monitors can determine during runtime when threads actually have an active stack frame for a given function, so runtime diversification will not modify those functions. These control flow monitors could also implement thread synchronization to prevent threads from entering a function that is currently being rewritten. Despite their certifiability by model-checking, these thread monitoring and synchronization techniques may incur high performance overhead, so this remains an open empirical question.

These challenges represent areas of future research and development for chronomorphic programs. Our prototype tool and preliminary analyses demonstrate that chronomorphic binaries reduce the predictability of code reuse attacks for single-threaded programs, and we believe that these avenues of future work will improve the safety and robustness of chronomorphic binaries in complex multi-threaded applications.

ACKNOWLEDGMENTS

This work was supported by The Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory (AFRL) under contract FA8650-10-C-7087. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Approved for public release, distribution unlimited.

REFERENCES

[1] S. E. Friedman, D. J. Musliner, and P. K. Keller, "Chronomorphic programs: Using runtime diversity to prevent code reuse attacks," in Proceedings ICDS 2015: The 9th International Conference on Digital Society, Feb. 2015.

[2] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in Proceedings of the 14th ACM conference on Computer and communications security. ACM, 2007, pp. 552–561.

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 30–40.

[4] E. J. Schwartz, T. Avgerinos, and D. Brumley, "Q: Exploit hardening made easy." in USENIX Security Symposium, 2011, pp. 25–41.

[5] J. Salwan and A. Wirth, "Ropgadget," URL http://shell-storm.org/project/ROPgadget, 2011.

[6] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, "Hacking blind," in Proceedings of the 35th IEEE Symposium on Security and Privacy, 2014, pp. 227–242.

[7] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 571–585.

[8] S. E. Friedman, D. J. Musliner, and J. M. Rye, "Improving automated cybersecurity by generalizing faults and quantifying patch performance," International Journal on Advances in Security, vol. 7, no. 3–4, 2014, pp. 121–130.

[9] L. Davi, A.-R. Sadeghi, and M. Winandy, "Ropdefender: A detection tool to defend against return-oriented programming attacks," in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011, pp. 40–51.

[10] H. Shacham et al., "On the effectiveness of address-space randomization," in Proceedings of the 11th ACM conference on Computer and communications security. ACM, 2004, pp. 298–307.

[11] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in Proceedings of the 2010 workshop on New security paradigms. ACM, 2010, pp. 7–16.

[12] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in Security and Privacy (SP), 2012 IEEE Symposium on. IEEE, 2012, pp. 601–615.

[13] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012, pp. 157–168.

[14] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in Network and System Security. Springer, 2013, pp. 293–306.

[15] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization." in USENIX Security Symposium, 2012, pp. 475–490.

[16] M. Backes and S. Nürnberger, "Oxymoron: making fine-grained memory randomization practical by allowing code sharing," in Proceedings of the 23rd USENIX conference on Security Symposium. USENIX Association, 2014, pp. 433–447.

[17] S. Crane et al., "Readactor: Practical code randomization resilient to memory disclosure," 2015.

[18] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," Proc. 22nd Network and Distributed Systems Security Sym.(NDSS), 2015.

[19] S. Crane, P. Larsen, S. Brunthaler, and M. Franz, "Booby trapping software," in Proceedings of the 2013 workshop on New security paradigms workshop. ACM, 2013, pp. 95–106.

[20] R. El-Khalil and A. D. Keromytis, "Hydan: Hiding information in program binaries," in Information and Communications Security. Springer, 2004, pp. 187–199.

[21] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Toronto, ON, Canada, Jul. 2011.

[22] K. W. Hamlen and M. Jones, "Aspect-oriented in-lined reference monitors," in Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security. ACM, 2008, pp. 11–20.

[23] M. Sridhar and K. W. Hamlen, "Model-checking in-lined reference monitors," in Verification, Model Checking, and Abstract Interpretation. Springer, 2010, pp. 312–327.